JavaScript

для профессиональных веб-разработчиков

Николас Закас

JavaScript
for Web Developers
Third Edition

Nicholas C. Zakas

John Wiley & Sons, Inc.

JavaScript
для профессиональных веб-разработчиков

Николас Закас

ПИТЕР®

Москва · Санкт-Петербург · Нижний Новгород · Воронеж
Ростов-на-Дону · Екатеринбург · Самара · Новосибирск
Киев · Харьков · Минск
2015

ББК 32.988.02-018
УДК 004.738.5
З-18

Закас Н.

З-18 JavaScript для профессиональных веб-разработчиков / [Пер. с англ. А. Лютича]. — СПб.: Питер, 2015. — 960 с.: ил. — (Серия «Для профессионалов»).

ISBN 978-5-496-01325-3

Если вы хотите полностью реализовать потенциал JavaScript, то крайне важно понять саму природу этого языка, его фундаментальные возможности и ограничения. Перед вами — обновленный вариант бестселлера, написанного гуру JavaScript Николасом Закасом. Автор показывает, как применять этот мощный инструмент для решения конкретных задач по созданию динамических пользовательских интерфейсов, которые стирают грань между настольными и веб-приложениями. Книга удачно сочетает в себе лучшие качества понятного учебного пособия, адресованного разработчикам, и всеобъемлющего руководства, которой всегда должно быть под рукой даже у профессионала.

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

ББК 32.988.02-018
УДК 004.738.5

ISBN 978-1118026694 англ.
ISBN 978-5-496-01325-3

Краткое содержание

Благодарности 23

Предисловие 24

Введение 26

Глава 1. Что такое JavaScript? 34

Глава 2. JavaScript в HTML 46

Глава 3. Основы языка 57

Глава 4. Переменные, область видимости и память 117

Глава 5. Ссылочные типы 135

Глава 6. Объектно-ориентированное программирование 203

Глава 7. Функции-выражения 247

Глава 8. Объектная модель браузера 269

Глава 9. Распознавание клиента 300

Глава 10. Объектная модель документа 338

Глава 11. Расширения DOM 386

Глава 12. DOM Level 2 и 3 410

Глава 13. События 461

Глава 14. Работа с формами 545

Глава 15. Рисование на холсте 587

Глава 16. HTML5 631

Глава 17. Обработка ошибок и отладка 648

Глава 18. XML в JavaScript 684

Глава 19. ECMAScript для XML 714

Глава 20. JSON 735

Глава 21. Ajax и Comet 746

Глава 22. Более сложные приемы 779

Глава 23. Автономный режим и клиентское хранилище 817

Глава 24. Наилучшие методики 856

Глава 25. Перспективные API 894

Приложение А. ECMAScript Harmony 919

Приложение Б. Строгий режим 940

Приложение В. JavaScript-библиотеки 947

Приложение Г. JavaScript-инструменты 953

Оглавление

Об авторе ... 22
О научном редакторе ... 22

Благодарности ... 23

Предисловие ... 24

Введение ... 26

Целевая аудитория ... 26
Темы, рассматриваемые в книге ... 27
Структура книги ... 27
Что нужно для эффективной работы с книгой ... 30
Принятые соглашения ... 31
Исходный код ... 31
Ошибки ... 32
Страница p2p.wrox.com ... 32

Глава 1. Что такое JavaScript? ... 34

Краткая история JavaScript ... 35
Реализации JavaScript ... 36
    ECMAScript ... 36
    Объектная модель документа ... 40
    Объектная модель браузера ... 43
Версии JavaScript ... 44
Резюме ... 45

Глава 2. JavaScript в HTML ... 46

Элемент <script> ... 46
    Расположение тегов ... 49
    Отложенные сценарии ... 50
    Асинхронные сценарии ... 51
    Изменения в XHTML ... 51
    Устаревший синтаксис ... 53
Встроенный код или внешние файлы? ... 53
Режимы документа ... 54
Элемент <noscript> ... 55
Резюме ... 56

Глава 3. Основы языка 57

Синтаксис 57
Чувствительность к регистру 57
Идентификаторы 58
Комментарии 58
Строгий режим 59
Инструкции 59
Ключевые и зарезервированные слова 60
Переменные 61
Типы данных 63
Оператор typeof 63
Тип Undefined 64
Тип Null 65
Тип boolean 66
Тип number 67
Тип string 74
Тип Object 77
Операторы 78
Унарные операторы 78
Поразрядные операторы 82
Логические операторы 88
Мультипликативные операторы 91
Операторы сложения и вычитания 93
Операторы отношений 96
Операторы эквивалентности 97
Условный оператор 100
Операторы присваивания 100
Оператор «запятая» 101
Инструкции 101
Инструкция if 101
Инструкция do-while 102
Инструкция while 103
Инструкция for 103
Инструкция for-in 105
Метки инструкций 105
Инструкции break и continue 106
Инструкция with 107
Инструкция switch 108
Функции 111
Аргументы функций 112
Никакой перегрузки 115
Резюме 115

Глава 4. Переменные, область видимости и память 117

Примитивные и ссылочные значения 117
Динамические свойства 118
Копирование значений 118
Передача аргументов 120
Проверка типа 122
Контекст выполнения и область видимости 123
Приращение цепочки областей видимости 125
Отсутствие блочных областей видимости 126
Сборка мусора 129
Отслеживание и очистка 130
Подсчет ссылок 130
Производительность 132
Управление памятью 132
Резюме 133

Глава 5. Ссылочные типы 135

Тип Object 136
Тип Array 138
Идентификация массивов 141
Методы преобразования массивов 142
Методы для работы с массивом как со стеком 144
Методы для работы с массивом как с очередью 145
Методы изменения порядка следования элементов 146
Методы манипулирования элементами 148
Методы поиска элементов 150
Методы перебора элементов 151
Методы редукции массивов 153
Тип Date 154
Унаследованные методы 156
Методы форматирования дат 157
Методы для работы с компонентами даты/времени 158
Тип RegExp 160
Свойства экземпляра RegExp 162
Методы экземпляра RegExp 163
Свойства конструктора RegExp 165
Ограничения шаблонов 167
Тип Function 168
Никакой перегрузки (новый взгляд) 169
Объявления функций и функции-выражения 170
Функции как значения 171
Внутри функций 172
Свойства и методы функций 175

Оболочки примитивных типов ... 178
Тип Boolean ... 180
Тип Number ... 181
Тип String ... 183
Встроенные одиночные объекты ... 193
Объект Global ... 193
Объект Math ... 198
Резюме ... 201

Глава 6. Объектно-ориентированное программирование ... 203

Общие сведения об объектах ... 203
Типы свойств ... 204
Определение нескольких свойств ... 208
Чтение атрибутов свойств ... 209
Создание объектов ... 210
Паттерн Фабрика ... 210
Паттерн Конструктор ... 211
Паттерн Прототип ... 214
Объединение паттернов Конструктор и Прототип ... 227
Паттерн Динамический прототип ... 228
Паттерн Паразитный конструктор ... 229
Паттерн Защищенный конструктор ... 231
Наследование ... 232
Цепочки прототипов ... 232
Кража конструктора ... 237
Комбинированное наследование ... 239
Прототипное наследование ... 240
Паразитное наследование ... 242
Паразитное комбинированное наследование ... 243
Резюме ... 245

Глава 7. Функции-выражения ... 247

Рекурсия ... 249
Замыкания ... 251
Замыкания и переменные ... 254
Объект this ... 255
Утечки памяти ... 257
Закрытые переменные ... 261
Статические закрытые переменные ... 262
Паттерн Модуль ... 264
Расширенный паттерн Модуль ... 266
Резюме ... 267

Глава 8. Объектная модель браузера 269

Объект window 269
Глобальная область видимости 270
Отношения окон и фреймов 271
Расположение окна 274
Размеры окна 275
Открытие окон и навигация 277
Интервалы и тайм-ауты 281
Системные диалоговые окна 284
Объект location 286
Аргументы строки запроса 287
Работа с объектом location 288
Объект navigator 290
Обнаружение подключаемых модулей 292
Регистрация обработчиков 295
Объект screen 296
Объект history 297
Резюме 299

Глава 9. Распознавание клиента 300

Распознавание возможностей 301
Надежное распознавание возможностей 302
Распознавание возможностей — не распознавание браузера 304
Распознавание особенностей 305
Распознавание пользовательского агента 306
История 306
Идентификация пользовательского агента 316
Полный сценарий 333
Использование сценария 336
Резюме 336

Глава 10. Объектная модель документа 338

Иерархия узлов 339
Тип Node 340
Тип Document 346
Тип Element 356
Тип Text 368
Тип Comment 371
Тип CDATASection 372
Тип DocumentType 373
Тип DocumentFragment 374
Тип Attr 375

Работа с DOM ... 376
Динамические сценарии ... 376
Динамические стили ... 378
Работа с таблицами ... 381
Использование объектов NodeList ... 383
Резюме ... 384

Глава 11. Расширения DOM ... 386
Selectors ... 386
Метод querySelector() ... 387
Метод querySelectorAll() ... 387
Метод matchesSelector() ... 388
Element Traversal ... 389
HTML5 ... 390
Новые средства работы с классами ... 390
Управление фокусом ... 393
Изменения типа HTMLDocument ... 394
Свойства кодировки ... 395
Пользовательские атрибуты данных ... 395
Вставка разметки ... 396
Метод scrollIntoView() ... 401
Фирменные расширения ... 402
Режим документа ... 402
Свойство children ... 403
Метод contains() ... 404
Вставка разметки ... 406
Прокрутка ... 408
Резюме ... 409

Глава 12. DOM Level 2 и 3 ... 410
Изменения DOM ... 411
XML-пространства имен ... 411
Другие изменения ... 415
Стили ... 420
Доступ к стилям элементов ... 420
Работа с таблицами стилей ... 425
Размеры элементов ... 430
Обход ... 437
Тип NodeIterator ... 439
Тип TreeWalker ... 442
Диапазоны ... 444
Диапазоны в DOM ... 444
Диапазоны в Internet Explorer 8 и более ранних версий ... 454
Резюме ... 459

Глава 13. События 461

Распространение событий 462
Всплытие событий 462
Перехват событий 463
Распространение DOM-событий 464
Обработчики событий 465
HTML-обработчики событий 465
Обработчики событий DOM Level 0 467
Обработчики событий DOM Level 2 468
Обработчики событий в Internet Explorer 470
Кроссбраузерные обработчики событий 471
Объект event 473
Объект event в DOM 473
Объект event в Internet Explorer 477
Кроссбраузерный объект event 479
Типы событий 482
События пользовательского интерфейса 482
События изменения фокуса 489
События мыши и колесика мыши 490
События клавиатуры и редактирования текста 503
События композиции 509
События изменения DOM-структуры 510
События HTML5 514
События устройств 523
События касаний и жестов 528
Память и быстродействие 532
Делегирование событий 532
Удаление обработчиков событий 534
Имитация событий 536
Имитация DOM-событий 536
Имитация событий в Internet Explorer 542
Резюме 544

Глава 14. Работа с формами 545

Общие сведения о формах 545
Отправка данных формы 546
Сброс формы 547
Поля форм 548
Работа с текстовыми полями 554
Выделение текста 555
Фильтрация ввода 559
Автоматический переход по нажатию клавиши табуляции 563

API проверки ограничений в HTML5 ... 564
Работа со списками ... 569
Выбор элементов списка ... 571
Добавление элементов в список ... 572
Удаление элементов списка ... 573
Перемещение и переупорядочение элементов списка ... 574
Сериализация форм ... 575
Редактирование форматированного текста ... 577
Атрибут contenteditable ... 578
Работа с форматированным текстом ... 579
Выделение форматированного текста ... 582
Форматированный текст в формах ... 584
Резюме ... 585

Глава 15. Рисование на холсте ... 587

Основы работы с элементом <canvas> ... 587
Двухмерный контекст ... 589
Заливка и рисование контура ... 589
Рисование прямоугольников ... 590
Рисование путей ... 592
Рисование текста ... 594
Преобразования ... 597
Рисование изображений ... 600
Тени ... 602
Градиенты ... 603
Узоры ... 605
Работа с данными изображений ... 606
Композиция изображений ... 608
WebGL ... 610
Типизированные массивы ... 610
Контекст WebGL ... 615
Поддержка ... 629
Резюме ... 630

Глава 16. HTML5 ... 631

Передача сообщений между документами ... 631
Встроенная поддержка перетаскивания ... 633
События перетаскивания ... 633
Пользовательские целевые элементы для данных ... 634
Объект dataTransfer ... 635
Свойства dropEffect и effectAllowed ... 637
Возможность перетаскивания ... 638
Дополнительные члены ... 638

Элементы для медиафайлов ... 639
Свойства ... 640
События ... 642
Пользовательские плееры ... 643
Распознавание кодеков ... 644
Тип Audio ... 645
Управление состоянием журнала ... 645
Резюме ... 647

Глава 17. Обработка ошибок и отладка ... 648

Уведомления об ошибках ... 648
Internet Explorer ... 649
Firefox ... 650
Safari ... 652
Opera ... 652
Chrome ... 655
Обработка ошибок ... 656
Инструкция try-catch ... 657
Генерирование ошибок ... 661
Событие error ... 664
Стратегии обработки ошибок ... 665
Идентификация потенциальных источников ошибок ... 666
Различение критичных и некритичных ошибок ... 671
Протоколирование ошибок на сервере ... 672
Приемы отладки ... 673
Вывод сообщений на консоль ... 673
Вывод сообщений на страницу ... 676
Генерирование ошибок ... 676
Частые ошибки Internet Explorer ... 678
Операция прервана ... 678
Недопустимый символ ... 680
Член группы не найден ... 680
Неизвестная ошибка выполнения ... 681
Синтаксическая ошибка ... 681
Не удается найти указанный ресурс ... 681
Резюме ... 682

Глава 18. XML в JavaScript ... 684

Поддержка XML DOM в браузерах ... 684
DOM Level 2 Core ... 684
Тип DOMParser ... 685
Тип XMLSerializer ... 687
XML в Internet Explorer 8 и более ранних версий ... 687
Кроссбраузерная обработка XML ... 692

Поддержка XPath в браузерах ... 694
DOM Level 3 XPath ... 694
XPath в Internet Explorer ... 700
Кроссбраузерная обработка XPath ... 701
Поддержка XSLT в браузерах ... 704
XSLT в Internet Explorer ... 704
Тип XSLTProcessor ... 709
Кроссбраузерные XSLT-преобразования ... 711
Резюме ... 712

Глава 19. ECMAScript для XML ... 714

Типы E4X ... 714
Тип XML ... 715
Тип XMLList ... 716
Тип Namespace ... 717
Тип QName ... 718
Общие принципы использования ... 719
Доступ к атрибутам ... 721
Другие типы узлов ... 723
Запросы ... 724
Конструирование и выполнение XML-кода ... 726
Параметры синтаксического анализа и сериализации ... 729
Пространства имен ... 730
Другие изменения ... 732
Полная поддержка E4X ... 733
Резюме ... 733

Глава 20. JSON ... 735

Синтаксис ... 736
Простые значения ... 736
Объекты ... 736
Массивы ... 737
Синтаксический анализ и сериализация ... 739
Объект JSON ... 739
Параметры сериализации ... 740
Параметры синтаксического анализа ... 744
Резюме ... 745

Глава 21. Ajax и Comet ... 746

Объект XMLHttpRequest ... 747
Использование объекта XHR ... 748
Заголовки HTTP ... 751
Запросы GET ... 753
Запросы POST ... 754

XMLHttpRequest LEVEL 2 ... 755
Тип FormData ... 755
Тайм-ауты ... 756
Метод overrideMimeType() ... 757
События хода обмена данными ... 758
Событие load ... 758
Событие progress ... 759
Обмен ресурсами с запросом происхождения ... 760
CORS в Internet Explorer ... 761
CORS в других браузерах ... 763
Предварительные запросы ... 763
Запросы с учетными данными ... 764
Кроссбраузерный CORS ... 765
Альтернативные методики кроссдоменного взаимодействия ... 766
Проверка связи с помощью изображения ... 766
JSONP ... 767
Comet ... 768
События, посылаемые сервером ... 771
Веб-сокеты ... 773
SSE или веб-сокеты? ... 776
Безопасность ... 776
Резюме ... 777

Глава 22. Более сложные приемы ... 779

Расширенное применение функций ... 779
Безопасное распознавание типов ... 779
Безопасные для областей видимости конструкторы ... 781
Отложенная загрузка функций ... 784
Связывание функций ... 787
Каррирование функций ... 789
Защищенные от изменений объекты ... 792
Нерасширяемые объекты ... 793
Запечатанные объекты ... 794
Замороженные объекты ... 794
Расширенные возможности работы с таймерами ... 795
Повторяющиеся таймеры ... 798
Управление процессами ... 800
Регулирование функций ... 802
Пользовательские события ... 805
Перетаскивание ... 808
Исправленное перетаскивание ... 811
Добавление пользовательских событий ... 813
Резюме ... 815

Глава 23. Автономный режим и клиентское хранилище 817

Распознавание автономного режима 818
Кэш приложений 819
Хранилище данных 821
Cookie-файлы 821
Пользовательские данные в Internet Explorer 832
Веб-хранилище 833
IndexedDB 840
Резюме 854

Глава 24. Наилучшие методики 856

Удобство сопровождения кода 856
Какой код удобно сопровождать? 857
Конвенции кодирования 857
Слабая связанность 861
Принципы программирования 865
Быстродействие 871
Область видимости 871
Выбор оптимального подхода 873
Сокращение количества инструкций 879
Оптимизация взаимодействия с DOM 882
Развертывание 885
Процесс сборки 885
Проверка кода 887
Сжатие 889
Резюме 892

Глава 25. Перспективные API 894

requestAnimationFrame() 894
Ранние способы создания анимаций 895
Проблемы с интервалами 895
mozRequestAnimationFrame 896
webkitRequestAnimationFrame и msRequestAnimationFrame 897
Page Visibility API 899
Geolocation API 901
File API 904
Тип FileReader 904
Частичное чтение 907
URL-адреса объектов 908
Чтение файлов и перетаскивание 909
Отправка файлов с помощью объекта XHR 910
Web Timing API 911

Рабочие веб-потоки ... 913
Использование рабочего веб-потока ... 913
Глобальная область видимости рабочего веб-потока ... 915
Подключение других сценариев ... 916
Будущее рабочих веб-потоков ... 917
Резюме ... 917

Приложение А. ECMAScript Harmony ... 919

Общие изменения ... 919
Константы ... 920
Блочная и другие области видимости ... 920
Функции ... 921
Прочие и распределенные аргументы ... 922
Предлагаемые по умолчанию значения аргументов ... 922
Генераторы ... 923
Массивы и другие структуры ... 924
Итераторы ... 924
Абстракции массивов ... 925
Присваивание с деструктуризацией ... 926
Новые типы объектов ... 928
Прокси-объекты ... 928
Функции-прокси ... 931
Типы Map и Set ... 931
Тип WeakMap ... 932
Тип StructType ... 932
Тип ArrayType ... 934
Классы ... 934
Закрытые члены ... 935
Методы чтения и записи свойств ... 936
Наследование ... 936
Модули ... 937
Внешние модули ... 938

Приложение Б. Строгий режим ... 940

Включение строгого режима ... 940
Переменные ... 941
Объекты ... 941
Функции ... 942
Функция eval() ... 944
Идентификаторы eval и arguments ... 944
Преобразование значения this ... 945
Другие изменения ... 945

Приложение В. JavaScript-библиотеки 947

Библиотеки общего назначения 947
YUI 947
Prototype 948
Dojo Toolkit 948
MooTools 948
jQuery 949
MochiKit 949
Underscore.js 949
Библиотеки для интернет-приложений 949
Backbone.js 950
Rico 950
qooxdoo 950
Библиотеки для анимации и эффектов 950
script.aculo.us 951
moo.fx 951
Lightbox 951
Библиотеки для криптографии 951
JavaScript MD5 952
JavaScrypt 952

Приложение Г. JavaScript-инструменты 953

Средства проверки кода 953
JSLint 953
JSHint 954
JavaScript Lint 954
Средства сокращения объема кода 954
JSMin 954
Dojo ShrinkSafe 955
YUI Compressor 955
Средства модульного тестирования 955
JsUnit 955
YUI Test 956
Dojo Object Harness (DOH) 956
qUnit 956
Генераторы документации 956
JsDoc Toolkit 957
YUI Doc 957
AjaxDoc 957
Среды безопасного выполнения кода 958
ADsafe 958
Caja 958

Посвящаю эту книгу моим родителям,
которые всегда поддерживали
и вдохновляли меня.

Об авторе

НИКОЛАС ЗАКАС занимается разработкой веб-приложений более десяти лет. Он принимал участие в создании корпоративных интранет-приложений для ряда крупнейших компаний в мире и разрабатывал такие известные потребительские веб-сайты, как My Yahoo! и домашняя страница Yahoo!. В качестве архитектора уровня представлений в Yahoo! Николас руководил разработкой клиентских компонентов популярнейших сайтов в мире. Он регулярно выступает на корпоративных собраниях, конференциях и неформальных мероприятиях, посвященных новым технологиям и лучшим методикам разработки клиентских компонентов веб-приложений. Является автором нескольких книг, включая *Professional Ajax and High Performance JavaScript*, и ведет блог на сайте http://www.nczonline.net/. В Twitter автора можно найти под ником @slicknet.

О научном редакторе

Джон Пелоквин (John Peloquin) занимается разработкой клиентских компонентов веб-приложений и за десять лет работы с JavaScript поучаствовал в создании решений всех размеров. Он получил диплом бакалавра по математике в Калифорнийском университете (Беркли) и в настоящее время занимает должность ведущего разработчика в молодой компании, где использует новейшие технологии программирования клиентских систем. Джон был научным редактором книги *JavaScript 24-Hour Trainer* (Wiley, 2010), которую написал Джереми Макпик (Jeremy McPeak). Если Джон не программирует и не ищет ошибки в книгах, его можно застать за занятиями математикой, философией или жонглированием.

Благодарности

Хотя на обложке этой книги красуется только имя ее автора, любая книга — результат коллективных усилий, и я хотел бы поблагодарить людей, которые помогали мне в работе над ней.

Прежде всего, я хочу сказать спасибо издательству John Wiley & Sons за то, что у меня вообще есть возможность писать. В свое время они рискнули выпустить в свет первое издание этой книги от неизвестного тогда автора, и за это я буду признателен им до конца своих дней.

Я благодарю сотрудников John Wiley & Sons, особенно Кевина Кента (Kevin Kent) и Джона Пелоквина (John Peloquin), которые помогли мне выразить свои идеи и хладнокровно принимали многие изменения, которые я вносил в книгу по мере ее написания.

Хочу также поблагодарить всех, кто предоставил отзывы о моих черновиках. Это Роб Фризел (Rob Friesel), Сергей Ильинский (Sergey Ilinsky), Дэн Килп (Dan Kielp), Питер-Пол Кох (Peter-Paul Koch), Джереми Макпик (Jeremy McPeak), Алекс Петреску (Alex Petrescu), Дмитрий Сошников (Dmitry Soshnikov) и Юрий «Kangax» Зайцев (Juriy «Kangax» Zaytsev). Ваши замечания помогли сделать эту книгу тем, чем я могу по-настоящему гордиться.

Выражаю особую благодарность Брендану Айку (Brendan Eich) за уточнения истории JavaScript, описанной в главе 1.

Наконец, я хочу сказать спасибо Рею Банго (Rey Bango) за предисловие. Я был очень рад впервые встретиться с Реем в 2010 году после нескольких лет общения через Интернет. Он один из самых приятных людей в IT-сообществе, и я горжусь тем, что он согласился внести свой вклад в эту книгу.

Предисловие

Оглядываясь назад, я могу назвать много технологий и людей, которые оказали важное влияние на мою профессиональную жизнь и мои решения. Тем не менее если бы мне нужно было выбрать одну технологию, повлиявшую на меня сильнее остальных, я бы выбрал JavaScript. Не буду врать — я не всегда верил в JavaScript. Как и многие разработчики, поначалу я думал, что это игрушечный язык, пригодный разве что для анимации банеров и добавления некоторых интересных эффектов на страницы. Я разрабатывал серверные компоненты приложений, и было жаль тратить время на такую ерунду. Но затем появился Ajax.

Никогда не забуду шумиху, которая поднялась вокруг этой технологии. Все только и говорили о том, какой это прорыв. Решив разобраться что к чему, я был ошеломлен, когда по мере чтения понял, что игрушечный язык, который я пренебрежительно игнорировал, превратился в технологию, приковавшую к себе внимание всех профессиональных веб-разработчиков. Мое отношение к веб-программированию тут же изменилось, и по мере изучения Ajax я понял, что JavaScript — невероятно мощный и эффективный язык, который непременно нужно попробовать в деле. Я с головой погрузился в него, чтобы понять все его нюансы, присоединился к разработчикам jQuery и сосредоточился на разработке клиентских компонентов веб-приложений. Жизнь заиграла для меня новыми красками.

Чем глубже я погружался в JavaScript, тем больше встречал разработчиков, которых по сей день считаю «звездами» и учителями. Николас Закас — один из них. Помню, как много знаний я почерпнул из второго издания, несмотря на весь свой многолетний опыт. Книга воспринималась как искренний глубокомысленный рассказ настоящего эксперта, способного объяснить материал людям с самыми разными уровнями подготовки. Это по-настоящему выделяло ее на фоне других технических книг. Многие авторы вываливают на читателя вагон технических деталей, чтобы впечатлить его своими знаниями, но эта книга была совсем не такой. Она захватила мои мысли на многие недели, и я стал рекомендовать ее всем, кто желал по-настоящему понять JavaScript. Я хотел, чтобы все почувствовали то, что чувствовал я, и поняли, насколько ценен такой источник знаний.

А затем на конференции по jQuery мне посчастливилось встретиться с автором «вживую». Один из лучших JavaScript-программистов, работающий с важнейшими веб-активами в мире (Yahoo!), оказался еще и одним из самых приятных людей, которых я когда-либо встречал. Признаю, что я даже испытывал легкое благоговение

перед ним, но оказалось, что он скромный и практичный человек, просто желающий помочь разработчикам стать лучше. В результате эта книга изменила мое мышление о JavaScript, а знакомство с ее автором вызвало у меня желание поработать вместе и узнать его лучше.

Когда он попросил меня написать это предисловие, я был весьма польщен. Для меня это как предварять выступление гуру на конференции, и я на самом деле не могу найти слов, чтобы выразить свое восхищение Николасом. Однако еще важнее для меня убедить читателей в важности этой книги. Я прочитал много книг о JavaScript и могу назвать несколько очень достойных работ, но эта стоит особняком в том смысле, что содержит все, что необходимо, чтобы стать профессиональным компетентным JavaScript-разработчиком.

Постепенный и продуманный переход от основ, таких как выражения и объявления переменных, к более сложным темам вроде замыканий и объектно-ориентированных приемов разработки — вот что отличает это руководство от других книг, которые либо грешат чрезмерной простотой, либо, наоборот, написаны так, как будто читатели с помощью JavaScript собираются программировать системы наведения ядерных боеголовок. Эта книга поможет вам писать код, которым вы сможете гордиться, и создавать сайты, делающие жизнь пользователей комфортнее и радостнее.

Рей Банго
Старший популяризатор технологий корпорации Microsoft
Рабочая группа jQuery

Введение

По мнению некоторых аналитиков, JavaScript является в настоящее время самым популярным языком программирования, что очень похоже на правду, если учесть, что на нем написано множество сложных веб-приложений, используемых по всему миру для бизнес-аналитики, закупок, управления процессами и т. д.

JavaScript имеет поверхностное сходство с Java — объектно-ориентированным языком программирования, который приобрел некоторую популярность среди веб-разработчиков благодаря встроенным апплетам. И все же несмотря на то, что синтаксис и методология программирования на JavaScript и Java похожи, JavaScript — это не упрощенная версия Java, а отдельный динамический язык, который надежно обосновался в веб-браузерах, помогая пользователям взаимодействовать с веб-сайтами и веб-приложениями.

Эта книга полностью описывает эволюцию JavaScript, начиная с его реализаций в ранних браузерах Netscape и заканчивая современными возможностями, включающими поддержку технологий DOM и Ajax. Вы узнаете, как расширить язык в соответствии с конкретными требованиями и как реализовать эффективное взаимодействие клиента с сервером без промежуточных компонентов, таких как Java-апплеты или скрытые фреймы. Используя полученные значения, вы сможете решать бизнес-проблемы, с которыми каждый день сталкиваются многие веб-разработчики по всему миру.

Целевая аудитория

Эта книга ориентирована на три группы читателей:

- профессиональные разработчики, имеющие опыт объектно-ориентированного программирования и желающие изучить JavaScript в контексте традиционных объектно-ориентированных языков, таких как Java и C++;
- разработчики веб-приложений, которым нужно сделать свои веб-сайты и веб-приложения более удобными в использовании;
- начинающие разработчики на JavaScript, желающие лучше понять этот язык.

Кроме того, книга может заинтересовать вас, если вы используете какие-либо из следующих технологий:

- Java;
- PHP;
- ASP.NET;
- HTML;
- CSS;
- XML.

Эта книга не подойдет вам, если вы не владеете базовыми навыками программирования или всего лишь хотите добавить на веб-сайт простые средства взаимодействия с пользователями. Если вы узнали себя в этом описании, вам лучше обратиться к четвертому изданию книги *Beginning JavaScript* от Wrox (Wiley, 2009).

Темы, рассматриваемые в книге

Книга, которую вы держите в руках, объединяет введение в JavaScript для разработчиков и описание более сложных и полезных возможностей этого языка.

Сначала мы рассмотрим историю и эволюцию JavaScript, после чего подробно обсудим компоненты языка, уделив особое внимание стандартам, таким как ECMAScript и Document Object Model (DOM), а также различиям JavaScript-реализаций в популярных веб-браузерах.

Взяв эту информацию за основу, далее мы рассмотрим базовые концепции JavaScript, в том числе особенности объектно-ориентированного программирования с его помощью, наследование и использование его в HTML. Вслед за подробным обсуждением обработки событий вы ознакомитесь с приемами распознавания браузеров и новыми API, такими как HTML5, Selectors и File.

Последняя часть книги посвящена более сложным темам, таким как оптимизация быстродействия и использования памяти, наилучшие методики работы и перспективные направления развития JavaScript.

Структура книги

В книге 25 глав.

1. **Что такое JavaScript?** Эта глава содержит общие сведения о JavaScript: вы узнаете, как появился этот язык, как он развивался и что он представляет собой сегодня. Мы обсудим, как JavaScript соотносится с ECMAScript, объектной моделью документа (DOM) и объектной моделью браузера (BOM). Кроме того, вы ознакомитесь

с соответствующими стандартами от Европейской ассоциации производителей вычислительной техники (ECMA) и консорциума World Wide Web (W3C).

2. **JavaScript в HTML.** В этой главе описано применение JavaScript в сочетании с HTML для создания динамических веб-страниц. Также рассмотрены различные способы внедрения JavaScript-кода в страницу, типы JavaScript-контента и их использование с элементом `<script>`.

3. **Основы языка.** В этой главе рассмотрены базовые концепции языка, в том числе его синтаксис и управляющие инструкции. В ней указаны сходства и различия JavaScript и других C-подобных языков, а также описано приведение типов в связи со встроенными операторами.

4. **Переменные, область видимости и память.** В этой главе рассказано о переменных, которые в JavaScript являются слабо типизированными. Глава содержит сведения о различиях между примитивными и ссылочными значениями и контексте выполнения в связи с переменными. Вы также узнаете о том, как работает сборщик мусора и как память возвращается среде, когда переменные покидают область видимости.

5. **Ссылочные типы.** Эта глава посвящена встроенным в JavaScript ссылочным типам, таким как `Object` и `Array`. Для каждого ссылочного типа, определенного в ECMA-262, приведены как теоретические сведения, так и подробности его реализации в браузерах.

6. **Объектно-ориентированное программирование.** В этой главе рассмотрены приемы объектно-ориентированного программирования на JavaScript. Поскольку в JavaScript нет классов, мы обсудим несколько популярных методик создания объектов и наследования. Глава также содержит сведения о прототипах функций и их использовании в рамках объектно-ориентированного подхода.

7. **Функции-выражения.** Функции-выражения относятся к наиболее мощным аспектам применения JavaScript. В этой главе описываются замыкания, детали функционирования объекта `this`, паттерн Модуль и создание закрытых членов объектов.

8. **Объектная модель браузера.** В этой главе описана объектная модель браузера (BOM), которая предоставляет объекты для взаимодействия с браузером. Вы ознакомитесь со всеми BOM-объектами, включая `window`, `document`, `location`, `navigator` и `screen`.

9. **Распознавание клиента.** В этой главе рассмотрены способы распознавания клиентского браузера и поддерживаемых им функциональных возможностей. Вы узнаете о распознавании возможностей, анализе строки пользовательского агента, о достоинствах и недостатках каждого подхода и о том, какой подход оптимален в той или иной ситуации.

10. **Объектная модель документа.** В этой главе описаны объекты, определенные в спецификации DOM Level 1. После ознакомления с XML в контексте DOM

вы сможете подробно изучить модель DOM и предоставляемые ею возможности по манипулированию содержимым страницы.

11. **Расширения DOM.** Эта глава содержит сведения о том, как API и сами браузеры расширяют функционал DOM. В число рассматриваемых тем входят Selectors, Element Traversal API и расширения HTML5.

12. **DOM Level 2 и 3.** В этой главе, основанной на двух предыдущих главах, рассказано о том, как спецификации DOM Level 2 и 3 расширяют DOM дополнительными свойствами, методами и объектами. Также рассмотрены проблемы совместимости Internet Explorer и других браузеров.

13. **События.** Из этой главы вы узнаете о природе JavaScript-событий, их генерировании, поддержке унаследованных возможностей и о том, как события переопределены в DOM. Помимо прочего, в главе рассмотрены события устройств Wii и iPhone.

14. **Работа с формами.** В этой главе рассказывается, как с помощью JavaScript улучшить взаимодействие с формами и обойти ограничения браузера. Особое внимание уделено работе с элементами форм, такими как текстовые поля и списки, а также проверке и обработке данных.

15. **Рисование на холсте.** Эта глава посвящена тегу `<canvas>` и его использованию для динамичного создания графики. Рассмотрены двухмерный контекст и контекст WebGL (трехмерный), что поможет вам приступить к созданию анимаций и игр.

16. **HTML5.** В этой главе представлены изменения JavaScript API в HTML5. Глава включает сведения о передаче сообщений между документами, элементах `<audio>` и `<video>` из Drag-and-Drop API, а также управлении состоянием журнала.

17. **Обработка ошибок и отладка.** В этой главе рассмотрены способы обработки ошибок в JavaScript-коде. Также описаны инструменты и приемы отладки для каждого браузера и приведены рекомендации по упрощению процесса отладки.

18. **XML в JavaScript.** В этой главе рассмотрены возможности JavaScript, используемые для чтения XML-данных и манипулирования ими. Описаны различия возможностей и объектов в разных веб-браузерах и приведены советы по написанию кроссбраузерного кода. Также в главе приведены сведения об использовании XSLT-преобразований для трансформации XML-данных на клиентских системах.

19. **ECMAScript для XML.** Эта глава посвящена расширению ECMAScript для XML (E4X), которое упрощает работу с XML. Также в ней описаны преимущества E4X над манипулированием XML с помощью DOM.

20. **JSON.** В этой главе представлен формат JSON — альтернатива XML. Описаны возможности синтаксического анализа и сериализации JSON и приведены сведения о том, как обеспечить безопасность при использовании JSON.

21. **Ajax и Comet.** В этой главе описаны популярные приемы работы с Ajax, в том числе использование объекта `XMLHttpRequest` и обмен ресурсами с запросом происхождения (CORS). Также рассмотрены различия реализаций и поддержки Ajax в браузерах и приведены рекомендации по использованию этой технологии.
22. **Более сложные приемы.** В этой главе описаны более сложные шаблоны JavaScript, такие как каррирование функций, частичное применение функций и динамические функции. Кроме того, в главе рассмотрены пользовательские события и создание объектов, защищенных от изменений, с помощью ECMAScript 5.
23. **Автономный режим и клиентское хранилище.** В этой главе рассказано о том, как определить, что приложение работает в автономном режиме, и описаны различные методики сохранения данных на клиентском компьютере. В главе рассмотрены как традиционные файлы cookie, так и более новые возможности, такие как веб-хранилище и база данных IndexedDB.
24. **Наилучшие методики.** Эта глава посвящена использованию JavaScript в корпоративной среде. В ней описаны приемы обслуживания кода, в том числе методики кодирования, форматирования кода и общие приемы программирования. Также приведены советы по оптимизации и повышению быстродействия кода. Наконец, рассмотрены вопросы развертывания приложений, включая реализацию процесса сборки.
25. **Перспективные API.** В этой главе представлены API, разрабатываемые для расширения возможностей JavaScript в браузере. Несмотря на то, что эти API пока реализованы не полностью, разработчики браузеров уже начинают их внедрение. Глава содержит сведения о Web Timing, Geolocation и File API.

Что нужно для эффективной работы с книгой

Для выполнения примеров из книги вам потребуется следующее:

- компьютер с операционной системой Windows XP, Windows 7 или Mac OS X;
- браузер Internet Explorer 6 или более поздней версии, Firefox 2 или более поздней версии, Opera 9 или более поздней версии, Chrome либо Safari 2 или более поздней версии.

Полный исходный код примеров можно загрузить с сайта www.wrox.com.

Принятые соглашения

Чтобы вы могли извлечь максимальную пользу от прочтения книги, мы использовали ряд специальных обозначений.

Врезки со значком предупреждения содержат важную информацию, непосредственно связанную с окружающим текстом.

Врезки со значком примечания содержат заметки, советы, рекомендации и отступления от темы.

В тексте также используются разные варианты начертания.

- Новые термины и важные слова при их употреблении в первый раз выделяются *курсивом*.
- Сочетания клавиш отформатированы следующим образом: Ctrl+A.
- Имена файлов и URL-адреса в тексте выделены следующим образом: persistence.properties.
- Фрагменты кода внутри текстовых абзацев (команды, ключевые слова и т. д.) выделены так: `yield`.
- Код в книге отформатирован двумя разными способами:

```
Все фрагменты кода набраны моноширинным шрифтом.
Код, особенно важный в текущем контексте, выделен полужирным шрифтом.
```

Исходный код

Изучая примеры кода, вы можете набирать код вручную или использовать прилагаемые к книге файлы. Исходный код всех приведенных в книге примеров можно загрузить с сайта www.wrox.com. Посетив сайт, просто укажите название книги в поле поиска или одном из списков книг и щелкните на ссылке Download Code (загрузить код) на странице с подробными сведениями о книге, чтобы загрузить весь исходный код к ней. Если код доступен на веб-сайте, вы увидите соответствующее сообщение.

Имена файлов с исходным кодом указаны перед листингами.

Загрузите код с сайта и извлеките его из архива. Вы также можете посетить главную страницу загрузки кода для книг издательства Wrox на сайте www.wrox.com/dynamic/books/download.aspx, где доступен код для этой и других книг издательства Wrox.

Ошибки

Мы сделали все возможное, чтобы ошибок в книге было как можно меньше. Однако никто не совершенен, и ошибки случаются. Если вы найдете ошибку в тексте или коде одной из наших книг, мы будем очень благодарны вам за сообщение нам о ней. Сделав это, вы сэкономите время других читателей и поможете нам повысить качество наших книг.

Чтобы ознакомиться со списком ошибок, найденных в этой книге, загрузите страницу www.wrox.com и укажите название книги в поле поиска или одном из списков книг. Затем на странице с подробными сведениями о книге щелкните на ссылке Errata. На этой странице вы можете просмотреть все ошибки, которые были проверены и опубликованы редакторами издательства Wrox. Полный список книг со ссылками на ошибки в каждой из них также доступен по ссылке www.wrox.com/misc-pages/booklist.shtml.

Если вы не можете найти «свою» ошибку на странице Errata, перейдите по ссылке www.wrox.com/contact/techsupport.shtml и заполните форму, чтобы отправить нам сведения об ошибке. Мы проверим информацию, и если она окажется верной, опубликуем сообщение на странице ошибок и исправим ошибку в следующем издании книги.

Страница p2p.wrox.com

Чтобы пообщаться с автором и единомышленниками, вы можете присоединиться к участникам веб-форумов P2P на сайте p2p.wrox.com. На этих форумах вы можете публиковать сообщения о книгах издательства Wrox и связанных технологиях и общаться с другими читателями. Кроме того, вы можете подписаться на почтовую рассылку по интересующим вас темам, чтобы получать уведомления о новых сообщениях. Наши форумы посещают авторы и редакторы издательства Wrox, другие отраслевые эксперты и, конечно же, обычные читатели.

Форумы на странице p2p.wrox.com будут вам полезны не только при чтении книги, но и при разработке ваших собственных приложений. Чтобы зарегистрироваться на форумах, выполните следующие действия.

1. Загрузите страницу p2p.wrox.com и щелкните на ссылке Register.
2. Ознакомьтесь с условиями использования и примите их.
3. Укажите необходимую информацию и при желании любые дополнительные сведения, а затем отправьте форму.
4. Вы получите электронное письмо со сведениями о том, как подтвердить учетную запись и завершить процесс регистрации.

Вы можете читать сообщения на форумах без регистрации, но публиковать новые сообщения могут только зарегистрированные пользователи.

После регистрации вы сможете в любое время публиковать новые сообщения и отвечать на сообщения других пользователей. Если вы захотите получать новые сообщения с конкретного форума по электронной почте, подпишитесь на форум, щелкнув на соответствующем значке в списке форумов.

Чтобы лучше узнать, как работают форумы издательства Wrox, ознакомьтесь с ответами на часто задаваемые вопросы о P2P, которые также включают многие вопросы о книгах издательства Wrox. Ссылка на список часто задаваемых вопросов доступна на каждой странице P2P.

1 Что такое JavaScript?

- Обзор истории JavaScript
- Общие сведения о JavaScript
- JavaScript как реализация ECMAScript
- Разные версии JavaScript

Когда в 1995 году появился JavaScript, его основным назначением была проверка вводимых пользователем данных, что прежде выполняли такие серверные языки, как Perl. Раньше, чтобы определить, не пропущено ли обязательное поле и допустимы ли введенные в форму значения, требовалось обращение к серверу. В Netscape Navigator с помощью JavaScript была предпринята попытка изменить ситуацию. Во времена коммутируемого доступа к Интернету возможность выполнять простую проверку на стороне клиента была воспринята с неподдельным энтузиазмом. Из-за низкой скорости подключения каждое обращение к серверу становилось настоящим испытанием терпения пользователей.

За прошедшее время JavaScript стал важным компонентом каждого популярного веб-браузера. Задачи JavaScript больше не ограничиваются простой проверкой данных: теперь он отвечает за взаимодействие почти всех составляющих окна браузера и его контента. JavaScript стал полноценным языком программирования, поддерживающим сложные вычисления и конструкции, включая замыкания, анонимные (лямбда) функции и даже метапрограммирование. JavaScript превратился в такую важную часть Сети, что его поддерживают даже альтернативные браузеры, в том числе браузеры для мобильных устройств и пользователей с ограниченными возможностями. Даже Microsoft использует собственную реализацию JavaScript в браузере Internet Explorer (с самых ранних версий), несмотря на наличие собственного клиентского языка сценариев VBScript.

Предугадать превращение JavaScript из простого инструмента для проверки вводимых данных в мощный язык программирования было невозможно. Он одновременно и прост, и сложен. Изучить его синтаксис можно за считанные минуты, но чтобы научиться применять его мастерски, требуются многие годы. Чтобы раскрыть полный потенциал JavaScript, важно понимать его природу, историю и ограничения.

Краткая история JavaScript

По мере роста популярности Интернета обозначилась потребность в языках сценариев для клиентской стороны. Хотя большинство пользователей подключались к Интернету с помощью модемов на скорости 28,8 Кбит/с, размер и сложность веб-страниц постоянно росли. Хуже того: даже для простой проверки форм требовалось несколько раз обращаться к серверу. Только представьте, каково было заполнить форму, щелкнуть на кнопке отправки, подождать 30 секунд, пока информация будет обработана, и получить сообщение о том, что при вводе данных было пропущено обязательное поле. В компании Netscape, бывшей тогда на рубеже инноваций, начали всерьез задумываться о разработке языка сценариев для простой обработки данных на клиентской стороне.

Брендан Айк (Brendan Eich), работавший тогда в Netscape, в 1995 году начал создавать язык сценариев Mocha (позднее переименованный в LiveScript) для браузера Netscape Navigator 2. Предполагалось, что этот язык будет использоваться и в браузере, и на сервере (под названием LiveWire). Чтобы успеть завершить реализацию LiveScript до выпуска браузера, компания Netscape объединила усилия с Sun Microsystems. Незадолго до выхода Netscape Navigator 2 в Netscape решили переименовать LiveScript в JavaScript, чтобы попытаться извлечь выгоду из популярности Java.

JavaScript 1.0 оказался очень успешным, и Netscape выпустила его версию 1.1 в составе Netscape Navigator 3. Популярность Интернета стремительно росла, и Netscape заслуженно занимала ведущее место на этом рынке. Тем временем в Microsoft решили выделить больше ресурсов на разработку конкурирующего браузера Internet Explorer. Вскоре после выхода Netscape Navigator 3 корпорация Microsoft представила Internet Explorer 3 со своей реализацией JavaScript под названием JScript (чтобы избежать проблем с Netscape, связанных лицензированием). Вторжение Microsoft в мир веб-браузеров в августе 1996 года оказалось началом конца Netscape, но в то же время ускорило развитие JavaScript.

То, что JavaScript был реализован в Microsoft, означало, что появилось две версии языка: JavaScript (Netscape Navigator) и JScript (Internet Explorer). В отличие от C и многих других языков программирования, на тот момент не было никаких стандартов JavaScript, определяющих его синтаксис и функциональность, и существование разных версий языка только подчеркивало эту проблему. Чтобы развеять опасения представителей отрасли, было решено стандартизировать язык.

В 1997 году спецификация JavaScript 1.1 была принята Европейской ассоциацией производителей вычислительной техники (European Computer Manufacturers

Association, Ecma). Был организован Технический комитет № 39 (Technical Committee, TC39), перед которым стояла задача «стандартизировать синтаксис и семантику кросс-платформенного независимого языка сценариев общего назначения» (www.ecma-international.org/memento/TC39.htm). Комитет TC39 объединил программистов Netscape, Sun, Microsoft, Borland, NOMBAS и других компаний, проявляющих интерес к будущему языков сценариев, и за несколько месяцев разработал стандарт ECMA-262, определивший новый язык сценариев с названием ECMAScript.

В следующем году Международная организация по стандартизации (International Organization for Standardization, ISO) и Международная электротехническая комиссия (International Electrotechnical Commission, IEC) также приняли ECMAScript в качестве стандарта (ISO/IEC-16262). С тех пор разработчики браузеров с переменным успехом используют ECMAScript как основу для реализации своих версий JavaScript.

Реализации JavaScript

Хотя названия JavaScript и ECMAScript часто используются как синонимы, JavaScript — это гораздо больше, чем стандарт ECMA-262. Полная реализация JavaScript состоит из трех частей (рис. 1.1):

- ядро (ECMAScript);
- объектная модель документа (Document Object Model, DOM);
- объектная модель браузера (Browser Object Model, BOM).

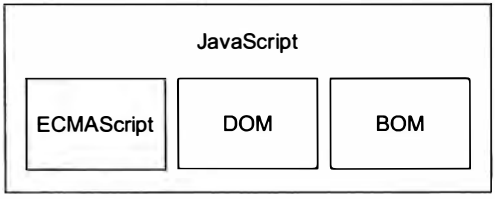

Рис. 1.1

ECMAScript

Сфера применения *ECMAScript* — языка, определенного в ECMA-262, — никак не связана с веб-браузерами. На самом деле в нем даже нет методов ввода и вывода данных. Стандарт ECMA-262 определяет этот язык как основу для создания полноценных языков сценариев. Веб-браузеры — это всего лишь одна из *сред выполнения* (host environment), в которых может работать ECMAScript-реализация. Среда выполнения содержит базовую ECMAScript-реализацию и ее расширения,

разработанные для взаимодействия с самой средой. Среди других сред выполнения можно отметить NodeJS (серверная JavaScript-платформа) и Adobe Flash.

Что же определяет стандарт ECMA-262, если в нем не фигурируют веб-браузеры? На базовом уровне он определяет следующие части языка:

- синтаксис;
- типы;
- инструкции;
- ключевые слова;
- зарезервированные слова;
- операторы;
- объекты.

ECMAScript — это просто описание языка, в котором реализованы все аспекты спецификации, а JavaScript — это реализация ECMAScript, но и Adobe ActionScript — тоже реализация ECMAScript.

Редакции ECMAScript

Версии ECMAScript называют *редакциями* (в соответствии с «номером» ECMA-262, в которой написана конкретная реализация). Последняя (пятая) редакция ECMA-262 вышла в 2009 году. Первая редакция ECMA-262 была почти такой же, как и JavaScript 1.1 от Netscape, но из нее были удалены все ссылки на код, специфичный для браузеров. Кроме того, в нее были внесены небольшие изменения: ECMA-262 должна была поддерживать стандарт Юникод (для использования других языков) и независимость объектов от платформы (в Netscape JavaScript 1.1 встречались реализации объектов, например `Date`, зависящие от платформы). Таким образом, реализации JavaScript 1.1 и 1.2 не соответствовали первой редакции ECMA-262.

Вторая редакция ECMA-262 появилась из бюрократических соображений. Стандарт был обновлен для согласования с ISO/IEC-16262 и не содержал никаких изменений. В реализациях ECMAScript вторая редакция обычно не используется для оценки соответствия стандарту.

Третья редакция ECMA-262 стала первым реальным усовершенствованием стандарта. В ней были обновлены спецификации обработки строк, определения ошибок и вывода чисел, добавлены регулярные выражения, некоторые управляющие инструкции и обработка исключений с помощью блоков `try-catch`. Также в нее были внесены небольшие изменения, позволяющие подготовить стандарт к интернационализации. Именно с третьей редакции многие начали воспринимать ECMAScript как настоящий язык программирования.

В четвертой редакции ECMA-262 язык был полностью переработан. Привлеченные популярностью JavaScript в Интернете, разработчики начали адаптировать

ECMAScript к растущим требованиям пользователей со всех уголков мира. В связи с этим снова был созван комитет TC39, чтобы определиться с будущим языка. Итоговая спецификация описывала практически новый язык, созданный на базе третьей редакции. Четвертая редакция включала строго типизированные переменные, новые инструкции и структуры данных, полноценные классы, классическое наследование и новые способы взаимодействия с данными.

В качестве альтернативного предложения подкомитет TC39 разработал спецификацию ECMAScript 3.1, которая не так сильно отличалась от третьей редакции. Она определяла дополнения к ECMAScript, которые могли быть реализованы поверх существующих модулей JavaScript. В итоге подкомитет ES3.1 склонил на свою сторону участников TC39, и работа над четвертой редакцией ECMA-262 была приостановлена.

Спецификация ECMAScript 3.1 стала пятой редакцией ECMA-262 и была официально опубликована 3 декабря 2009 года. Она проясняет неоднозначные места третьей редакции и вводит новую функциональность, в том числе встроенный объект `JSON` для синтаксического анализа и сериализации данных в формате JSON, методы наследования и расширенного определения свойств и новый строгий режим, немного расширяющий возможности интерпретации и выполнения кода модулями ECMAScript.

Что означает «соответствие спецификации ECMAScript»?

ECMA-262 проверяет соответствие спецификации ECMAScript. Чтобы считаться ECMAScript-реализацией, язык должен:

- поддерживать все «типы, значения, объекты, свойства, функции, а также синтаксис и семантику программ» (ECMA-262, p. 1) согласно их описанию в ECMA-262;
- поддерживать стандарт символов Юникода.

Кроме того, реализация, соответствующая требованиям, может:

- содержать «дополнительные типы, значения, объекты, свойства и функции», не указанные в ECMA-262 (дополнительные элементы описаны преимущественно как новые объекты или новые свойства объектов, которых нет в спецификации);
- поддерживать «синтаксис программ и регулярных выражений», не определенный в ECMA-262 (то есть встроенные средства поддержки регулярных выражений можно изменять и расширять).

Данные критерии позволяют разработчикам создавать новые языки на основе спецификации ECMAScript, чем частично и объясняется ее популярность.

Поддержка ECMAScript в веб-браузерах

Браузер Netscape Navigator 3 с поддержкой JavaScript 1.1 был выпущен в 1996 году. Эта же спецификация JavaScript 1.1 затем была отправлена в Ecma как предложение

нового стандарта ECMA-262. Наблюдая за взрывным ростом популярности JavaScript, в Netscape с радостью приступили к разработке версии 1.2. Однако существовала проблема: в Ecma еще не приняли предложение Netscape.

Вскоре после выпуска Netscape Navigator 3 корпорация Microsoft представила Internet Explorer 3. Эта версия IE включала язык Jscript 1.0, который должен был стать аналогом JavaScript 1.1, однако из-за недокументированных и неточно воспроизведенных функций оказался гораздо менее популярным.

Браузер Netscape Navigator 4 с JavaScript 1.2 вышел в 1997 году, опередив принятую в том же году первую редакцию ECMA-262. В результате оказалось, что JavaScript 1.2 не соответствует первой редакции ECMAScript, которая должна была базироваться на JavaScript 1.1.

Следующим обновлением JScript стал Jscript 3.0, входящий в состав Internet Explorer 4 (версия 2.0 была в составе Microsoft Internet Information Server 3.0, но никогда не включалась в браузеры). Microsoft выпустила пресс-релиз, расхваливающий Jscript 3.0 как первый по-настоящему совместимый с Ecma язык сценариев. Окончательная версия ECMA-262 все еще не была принята, так что Jscript 3.0 постигла та же судьба, что и JavaScript 1.2, — оказалось, что он не соответствует ECMAScript.

Netscape Navigator 4.06 вышел с обновленной версией JavaScript 1.3, которая была полностью совместимой с первой редакцией ECMA-262. Netscape добавила поддержку стандарта Юникод и сделала все объекты платформенно-независимыми, сохранив при этом возможности, представленные в JavaScript 1.2.

Когда Netscape открыла исходный код браузера в рамках проекта Mozilla, предполагалось, что JavaScript 1.4 войдет в Netscape Navigator 5, однако радикальное решение полностью переработать код браузера поставило крест на этих ожиданиях. JavaScript 1.4 был выпущен только как серверный язык для Netscape Enterprise Server и никогда не использовался в браузерах.

К 2008 году все пять основных веб-браузеров (Internet Explorer, Firefox, Safari, Chrome и Opera) соответствовали третьей редакции ECMA-262. Internet Explorer 8 стал первым браузером, в котором была начата реализация пятой редакции ECMA-262, а в Internet Explorer 9 она была реализована полностью. Вскоре к IE присоединился Firefox 4. Сведения о поддержке ECMAScript в наиболее популярных веб-браузерах представлены в следующей таблице.

Браузер	Соответствие спецификации ECMAScript
Netscape Navigator 2	—
Netscape Navigator 3	—
Netscape Navigator 4–4.05	—
Netscape Navigator 4.06–4.79	Редакция 1

Браузер	Соответствие спецификации ECMAScript
Netscape 6+ (Mozilla 0.6.0+)	Редакция 3
Internet Explorer 3	—
Internet Explorer 4	—
Internet Explorer 5	Редакция 1
Internet Explorer 5.5–7	Редакция 3
Internet Explorer 8	Редакция 5*
Internet Explorer 9+	Редакция 5
Opera 6–7.1	Редакция 2
Opera 7.2+	Редакция 3
Safari 1–2.0.x	Редакция 3*
Safari 3.x	Редакция 3
Safari 4.x–5.x	Редакция 5*
Chrome 1+	Редакция 3
Firefox 1–2	Редакция 3
Firefox 3.0.x	Редакция 3
Firefox 3.5–3.6	Редакция 5*
Firefox 4+	Редакция 5

* Неполная реализация.

Объектная модель документа

Объектная модель документа (DOM) — это прикладной программный интерфейс (Application Programming Interface, API) для XML, применение которого было расширено на HTML. В DOM вся страница представляется как иерархия узлов. Каждый элемент HTML- или XML-страницы является узлом определенного типа, содержащим те или иные данные. Рассмотрим следующую HTML-страницу:

```
<html>
    <head>
        <title>Sample Page</title>
    </head>
    <body>
        <p>Hello World!</p>
    </body>
</html>
```

Этот код можно изобразить с помощью DOM как иерархию узлов (рис. 1.2).

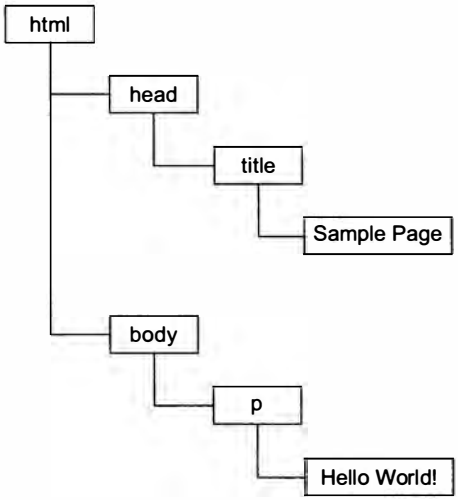

Рис. 1.2

Представляя документ в виде дерева, DOM предоставляет разработчикам беспрецедентный контроль над его контентом и структурой. Используя DOM API, можно с легкостью удалять узлы, добавлять, заменять и изменять их.

Почему необходима модель DOM

Благодаря реализации динамического HTML (Dynamic HTML, DHTML), в Internet Explorer 4 и Netscape Navigator 4 разработчики впервые смогли изменять вид и контент веб-страниц без их перезагрузки. Это стало важным этапом развития веб-технологий, но возникла серьезная проблема. Netscape и Microsoft выбрали разные пути развития DHTML, что завершило период, когда можно было писать HTML-страницы, не задумываясь о веб-браузере.

Стало ясно, что для сохранения кросс-платформенности нужно что-то делать, ведь из-за потенциальных разногласий Netscape и Microsoft Сеть могла разделиться на две части, каждая из которых была бы доступна только пользователям конкретного браузера. Комитет по стандартизации веб-коммуникаций консорциума World Wide Web (W3C) приступил к работе над DOM.

Уровни DOM

В октябре 1998 года спецификация DOM Level 1 получила статус рекомендации W3C. Она состояла из двух модулей: DOM Core (ядро DOM), определяющий способ представления структуры XML-документа и обеспечивающий удобный доступ к любым частям документа и выполнения операций над ними, и DOM HTML — расширение ядра DOM, определяющее объекты и методы, специфичные для HTML.

Имейте в виду, что модель DOM не является специфичной для JavaScript и реализована во многих других языках. Однако для веб-браузеров DOM реализована с использованием ECMAScript и теперь является значимой частью JavaScript.

Спецификация DOM Level 1 была предназначена для представления структуры документа, а DOM Level 2 охватывала гораздо больше областей. В оригинальную модель DOM была добавлена поддержка мыши и событий пользовательского интерфейса (давно поддерживавшихся в DHTML), диапазонов, способов обхода DOM-элементов, а также поддержка каскадных таблиц стилей (Cascading Style Sheets, CSS) с помощью объектных интерфейсов. Ядро DOM, представленное в DOM Level 1, также было расширено поддержкой пространств имен XML. Для работы с новыми интерфейсами в DOM Level 2 были представлены новые модули:

- **DOM Views** — интерфейсы для отслеживания различных представлений документа (например, документ до и после применения стилей CSS);
- **DOM Events** — интерфейсы для событий и обработки событий;
- **DOM Style** — интерфейсы для работы со стилизацией элементов с помощью CSS;
- **DOM Traversal and Range** — интерфейсы для обхода элементов дерева документа и выполнения операций над ними.

DOM Level 3 дополнила DOM унифицированными методами загрузки и сохранения документов (содержатся в новом модуле DOM Load and Save) и методами проверки документа (DOM Validation). В ядре DOM была реализована поддержка всей спецификации XML 1.0, включая XML Infoset, XPath и XML Base.

Изучая DOM, вы можете встретить упоминания DOM Level 0. Имейте в виду, что стандарта DOM Level 0 не существует, это просто стартовая точка истории DOM. За DOM Level 0 принимают оригинальный DHTML, поддержка которого была реализована в Internet Explorer 4.0 и Netscape Navigator 4.0.

Другие DOM

Кроме интерфейсов DOM Core и DOM HTML отдельные стандарты DOM опубликованы для нескольких других языков. Так, следующие языки основаны на XML, и для каждого из них DOM добаляет уникальные методы и интерфейсы:

- Scalable Vector Graphics (SVG) 1.0;
- Mathematical Markup Language (MathML) 1.0;
- Synchronized Multimedia Integration Language (SMIL).

Некоторые языки, такие как XML User Interface Language (XUL) от Mozilla, содержат собственные DOM-реализации, но только языки из приведенного списка являются стандартными рекомендациями W3C.

Поддержка DOM в веб-браузерах

Спустя некоторое время стандарт DOM начали реализовывать в браузерах. В Internet Explorer начало было положено в версии 5, но серьезной поддержки DOM в IE не было до версии 5.5, в которой была реализована основная часть DOM Level 1. В Internet Explorer 6 и 7 новые функции DOM не добавлялись, а в версии 8 были исправлены некоторые ошибки.

Netscape Navigator не поддерживал DOM до версии Netscape 6 (Mozilla 0.6.0), а после Netscape 7 разработчики из Mozilla сосредоточились на браузере Firefox. Firefox 3+ полностью поддерживает DOM Level 1, почти полностью Level 2 и частично Level 3 (в Mozilla поставлена цель создать браузер, полностью совместимый со стандартами, и усилия себя оправдывают).

Поддержка DOM стала одним из приоритетов для производителей браузеров, которые с каждым выпуском делают ее более полной. В следующей таблице представлены сведения о поддержке DOM в популярных браузерах.

Браузер	Соответствие спецификации DOM
Netscape Navigator 1.–4.x	—
Netscape 6+ (Mozilla 0.6.0+)	Level 1, 2 (почти полностью), 3 (частично)
Internet Explorer 2–4.x	—
Internet Explorer 5	Level 1 (минимально)
Internet Explorer 5.5–8	Level 1 (почти полностью)
Internet Explorer 9+	Level 1, 2, 3
Opera 1–6	—
Opera 7–8.x	Level 1 (почти полностью), 2 (частично)
Opera 9–9.9	Level 1, 2 (почти полностью), 3 (частично)
Opera 10+	Level 1, 2, 3 (частично)
Safari 1.0.x	Level 1
Safari 2+	Level 1, 2 (частично)
Chrome 1+	Level 1, 2 (частично)
Firefox 1+	Level 1, 2 (почти полностью), 3 (частично)

Объектная модель браузера

В Internet Explorer 3 и Netscape Navigator 3 была представлена *объектная модель браузера* (BOM), которая обеспечивает доступ к окну браузера и позволяет манипулировать его элементами. Используя BOM, можно взаимодействовать с браузером вне контекста отображаемой страницы. До недавних пор BOM была единственной частью реализации JavaScript, не имеющей стандарта, из-за чего при работе с ней

часто возникали проблемы. Формализация многих элементов BOM в HTML5 изменила ситуацию к лучшему, прояснив многие неясные аспекты модели.

BOM регламентирует работу с окном и фреймами браузера, но любое специфичное для браузера JavaScript-расширение тоже обычно считается частью BOM. Вот некоторые такие расширения:

- функция отображения всплывающих окон в браузере;
- возможность перемещать, закрывать и изменять размеры окна браузера;
- объект `navigator`, предоставляющий подробные сведения о браузере;
- объект `location`, предоставляющий подробные сведения о странице, загруженной в браузере;
- объект `screen`, предоставляющий подробные сведения о разрешении экрана;
- поддержка cookie-файлов;
- настраиваемые объекты, включая `XMLHttpRequest`, а также `ActiveXObject` в Internet Explorer.

Стандартов BOM долго не было, поэтому в каждом браузере она реализована по-своему. Де-факто некоторые стандарты существуют, например поддержка объектов `window` и `navigator`, но каждый браузер определяет для этих и других объектов свои методы и свойства. Ожидается, что благодаря HTML5 реализации BOM будут развиваться более согласованно. Подробнее тема BOM обсуждается в главе 8.

Версии JavaScript

Как преемник Netscape, фонд Mozilla является единственным производителем браузеров, который продолжает оригинальную нумерацию версий Javascript. Когда был создан проект Mozilla с открытым исходным кодом, новейшей версией JavaScript в браузерах была версия 1.3 (как уже отмечалось, версия 1.4 использовалась только на сервере). По мере добавления в язык новых возможностей, ключевых слов и элементов синтаксиса номера версий увеличивались. В приведенной далее таблице показано, как изменялись номера версий JavaScript в браузерах Netscape/Mozilla.

Браузер	Версия JavaScript
Netscape Navigator 2	1.0
Netscape Navigator 3	1.1
Netscape Navigator 4	1.2
Netscape Navigator 4.06	1.3
Netscape 6+ (Mozilla 0.6.0+)	1.5
Firefox 1	1.5

Браузер	Версия JavaScript
Firefox 1.5	1.6
Firefox 2	1.7
Firefox 3	1.8
Firefox 3.5	1.8.1
Firefox 3.6	1.8.2

Предполагалось, что в Firefox 4 будет использоваться JavaScript 2.0, и каждое увеличение номера версии указывает, насколько близка реализация JavaScript к спецификации 2.0. Однако эволюция JavaScript пошла по иному пути, нарушив первоначальный план. В настоящее время у Mozilla нет планов выпуска конкретного продукта с JavaScript 2.0.

Важно отметить, что этой схеме нумерации соответствуют только браузеры Netscape/Mozilla. Например, в Internet Explorer для нумерации версий JScript применяется совершенно другая схема. Более того, в большинстве браузеров поддержку JavaScript оценивают по соответствию спецификации ECMAScript и поддержке DOM.

Резюме

JavaScript — это язык сценариев, разработанный для взаимодействия с веб-страницами и состоящий из трех основных компонентов:

- язык ECMAScript, определенный в стандарте ECMA-262, обеспечивает базовую JavaScript-функциональность;
- объектная модель документа предоставляет методы и интерфейсы для работы с контентом веб-страницы;
- объектная модель браузера предоставляет методы и интерфейсы для взаимодействия с браузером.

Пять основных веб-браузеров (Internet Explorer, Firefox, Chrome, Safari и Opera) различаются по поддержке компонентов JavaScript. Все производители браузеров в целом адекватно реализовали спецификацию ECMAScript 3 и постепенно расширяют поддержку ECMAScript 5, тогда как степень реализации DOM варьируется в широких пределах. Поддержка модели BOM, недавно формализованной в HTML5, также зависит от браузера, хотя некоторые ее элементы универсальны.

2 JavaScript в HTML

- Использование элемента `<script>`
- Сравнение встроенных и внешних сценариев
- Режимы документа в контексте JavaScript
- Показ веб-страниц без JavaScript

С появлением JavaScript немедленно возник вопрос его интеграции с HTML, главным языком веб-страниц. Еще при разработке JavaScript специалисты Netscape думали над тем, как использовать JavaScript в HTML-страницах так, чтобы не нарушить их визуализацию в других браузерах. После многочисленных проб, ошибок и споров удалось прийти к некоторым решениям по универсальной поддержке сценариев в Сети. Многое из того, что было изобретено на заре развития веб-технологий, было формализовано в спецификации HTML и используется по сей день.

Элемент <script>

Для вставки JavaScript-кода в HTML-страницу обычно используют элемент `<script>`, который впервые появился в браузере Netscape Navigator 2, а позднее вошел в формальную спецификацию HTML. Он имеет шесть атрибутов:

- `async` (необязательный). Указывает, что нужно немедленно начать загрузку сценария с сервера и сразу же перейти к выполнению других действий на странице, таких как загрузка ресурсов или других сценариев. Действителен только для внешних файлов сценариев.
- `charset` (необязательный). Определяет кодировку сценария, указанного с помощью атрибута `src`. Этот атрибут используется редко, и большинство браузеров его игнорируют.

- `defer` (необязательный). Указывает, что выполнение сценария можно безопасно отложить, пока не будут полностью закончены синтаксический анализ и визуализация контента документа. Действителен только для внешних сценариев. В Internet Explorer 7 и более ранних версиях этот атрибут можно использовать во встроенных сценариях.
- `language` (устарел). Определял язык сценариев, используемый в блоке кода (например, `"JavaScript"`, `"JavaScript1.2"` или `"VBScript"`). Большинство браузеров игнорируют этот атрибут, поэтому использовать его не следует.
- `src` (необязательный). Указывает внешний файл с кодом, который нужно выполнить.
- `type` (необязательный). Заменяет атрибут `language`. Указывает тип контента (MIME-тип) языка сценариев, который используется в блоке кода. Традиционно этот атрибут имел значение `"text/javascript"`, но оба значения, и `"text/javascript"`, и `"text/ecmascript"`, устарели. JavaScript-файлы обычно возвращаются сервером с MIME-типом `"application/x-javascript"`, хотя присвоение этого значения атрибуту `type` может привести к игнорированию сценария. В браузерах, отличных от Internet Explorer, также поддерживаются `"application/javascript"` и `"application/ecmascript"`. По традиции и ради обеспечения совместимости этому атрибуту обычно присваивают значение `"text/javascript"`. Атрибут `type` можно безопасно опускать, так как при его отсутствии используется значение `"text/javascript"`.

Есть два способа применения элемента `<script>`: можно внедрить JavaScript-код непосредственно в страницу или включить в нее сценарий из внешнего файла.

Чтобы встроить JavaScript-код непосредственно в страницу, поместите его прямо в элемент `<script>`:

```
<script type="text/javascript">
    function sayHi(){
        alert("Hi!");
    }
</script>
```

JavaScript-код в элементе `<script>` обрабатывается сверху вниз. В приведенном примере определение функции интерпретируется и сохраняется в среде интерпретатора. Остальной контент страницы не загружается и не отображается, пока не будет выполнен весь код в элементе `<script>`.

При написании встроенного JavaScript-кода помните, что использовать строку `"</script>"` в своем коде нельзя. Например, при загрузке следующего кода в браузере возникнет ошибка:

```
<script type="text/javascript">
    function sayScript(){
        alert("</script>");
    }
</script>
```

При синтаксическом анализе встроенного в страницу сценария браузер интерпретирует строку "</script>" как закрывающий тег </script>. Чтобы решить эту проблему, экранируйте знак /, как показано в этом фрагменте:

```
<script type="text/javascript">
    function sayScript(){
        alert("<\/script>");
    }
</script>
```

Это изменение устраняет ошибку, делая код понятным для браузеров.

Чтобы включить в страницу JavaScript-код из внешнего файла, нужно использовать атрибут src. Его значением должен быть URL-адрес файла со сценарием, например:

```
<script type="text/javascript" src="example.js"></script>
```

В этом примере в страницу загружается внешний файл с именем example.js. Сам файл должен содержать только JavaScript-код, который иначе располагался бы между тегами <script> и </script>. Как и в случае встроенного JavaScript-кода, на время интерпретации внешнего файла обработка страницы приостанавливается (также требуется некоторое время на загрузку файла). В XHTML-документах закрывающий тег можно опускать, например:

```
<script type="text/javascript" src="example.js" />
```

Не используйте такой синтаксис в HTML-документах, потому что он нарушает правила HTML и неправильно обрабатывается некоторыми браузерами, в частности Internet Explorer.

Внешним JavaScript-файлам обычно назначают расширение .js, но это не обязательно, поскольку браузеры не проверяют расширения включаемых файлов. Это позволяет динамически генерировать JavaScript-код с помощью JSP, PHP или другого языка сценариев на стороне сервера. Помните, однако, что серверы часто используют расширение файла для определения правильного MIME-типа, назначаемого ответу. Если вы не применяете расширение .js, убедитесь, что ваш сервер возвращает правильный MIME-тип.

Элемент <script> с атрибутом src не может содержать дополнительный JavaScript-код между тегами <script> и </script>, в противном случае, хотя сценарий загружается и выполняется, встроенный код игнорируется.

Одной из наиболее мощных и противоречивых особенностей элемента <script> является возможность включать JavaScript-файлы из внешних доменов. По аналогии с элементом <img> атрибуту src элемента <script> можно назначить полный URL-адрес, не относящийся к домену текущей HTML-страницы, например:

```
<script type="text/javascript" src="http://www.somewhere.com/afile.js"></script>
```

Код из внешнего домена загружается и интерпретируется как часть страницы, в которую он загружается. Это позволяет при необходимости получать JavaScript-сценарии из других доменов, но будьте осторожны, загружая сценарий с сервера, который вы не контролируете: злоумышленник может в любой момент заменить загружаемый файл. Включайте JavaScript-файлы из других доменов, только если эти домены принадлежат вам или если вы доверяете их владельцам.

Независимо от того, как код включается в HTML-страницу, элементы `<script>` интерпретируются в том порядке, в котором они расположены, при условии, что у них нет атрибутов `defer` и `async`. Код первого элемента `<script>` должен быть полностью интерпретирован, чтобы можно было приступить ко второму элементу `<script>`, второй элемент должен быть полностью обработан перед третьим, и т. д.

Расположение тегов

Все элементы `<script>` в коде страницы традиционно размещались внутри элемента `<head>`, как в следующем примере HTML-страницы:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Example HTML Page</title>
    <script type="text/javascript" src="example1.js"></script>
    <script type="text/javascript" src="example2.js"></script>
  </head>
  <body>
    <!-- контент -->
  </body>
</html>
```

Такой формат использовался во многом для того, чтобы ссылки на внешние CSS- и JavaScript-файлы находились в одном месте. Однако подобный формат имеет существенный недостаток: загрузка, синтаксический анализ и интерпретация всех сценариев должны быть завершены до начала визуализации страницы (визуализация начинается, когда браузер получает открывающий тег `<body>`). Если на странице используется много JavaScript-кода, это может приводить к длительным задержкам, в течение которых пользователь видит пустое окно браузера. Поэтому в современных веб-приложениях все ссылки на JavaScript-сценарии обычно указываются в элементе `<body>` после контента страницы:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Example HTML Page</title>
  </head>
  <body>
    <!-- контент -->
    <script type="text/javascript" src="example1.js"></script>
    <script type="text/javascript" src="example2.js"></script>
  </body>
</html>
```

При таком подходе страница полностью визуализируется в браузере до обработки JavaScript-кода. Для пользователей это предпочтительнее, потому что контент отображается раньше.

Отложенные сценарии

В HTML 4.01 для элемента `<script>` определен атрибут `defer`, который указывает, что сценарий не будет изменять структуру страницы, а потому его можно безопасно выполнить после синтаксического анализа всей страницы. Атрибут `defer` сигнализирует браузеру, что загрузку сценария можно начать немедленно, но его выполнение следует отложить:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Example HTML Page</title>
    <script type="text/javascript" defer src="example1.js"></script>
    <script type="text/javascript" defer src="example2.js"></script>
  </head>
  <body>
    <!-- контент -->
  </body>
</html>
```

Несмотря на то что в этом примере элементы `<script>` включены в элемент `<head>` документа, выполнение сценариев не начнется, пока браузер не получит закрывающий тег `</html>`.

В спецификации HTML5 указано, что сценарии интерпретируются по порядку, так что первый отложенный сценарий выполняется перед вторым, а оба они будут выполнены до события `DOMContentLoaded` (см. главу 13). Однако в реальности требования спецификации не всегда соблюдаются, поэтому по возможности лучше включать в страницу только один отложенный сценарий.

Как уже отмечалось, атрибут `defer` поддерживается только для внешних файлов сценариев. Это уточнение было добавлено в HTML5, так что браузеры, поддерживающие HTML5 (включая Internet Explorer 8 и более поздних версий), игнорируют атрибут `defer`, если он задан для встроенного сценария. Браузеры Internet Explorer 4–7 работают по-старому.

Поддержка атрибута `defer` появилась в браузерах Internet Explorer 4, Firefox 3.5, Safari 5 и Chrome 7. Более старые браузеры просто игнорируют этот атрибут и обрабатывают сценарии с ним обычным образом, поэтому отложенные сценарии лучше располагать в конце страницы.

В XHTML-документах указывайте атрибут defer как defer="defer".

Асинхронные сценарии

В HTML5 для элемента `<script>` представлен атрибут `async`, который похож на атрибут `defer` в том смысле, что он тоже изменяет способ обработки сценария. Он также применяется только к внешним сценариям и указывает браузеру немедленно начать загрузку файла, но для сценариев с атрибутом `async` не гарантируется выполнение в порядке их добавления, например:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Example HTML Page</title>
    <script type="text/javascript" async src="example1.js"></script>
    <script type="text/javascript" async src="example2.js"></script>
  </head>
  <body>
    <!-- контент -->
  </body>
</html>
```

Здесь второй сценарий может быть выполнен перед первым, поэтому важно, чтобы между ними не было зависимостей. Атрибут `async` используется, если нужно разрешить браузеру продолжить загрузку страницы, не дожидаясь завершения загрузки и выполнения сценария. По этой причине рекомендуется не изменять в асинхронных сценариях DOM-элементы.

Асинхронные сценарии гарантированно выполняются до события `load` страницы, но могут выполняться до или после события `DOMContentLoaded` (см. главу 13). Они поддерживаются в браузерах Firefox 3.6, Safari 5 и Chrome 7.

В XHTML-документах указывайте атрибут async как async="async".

Изменения в XHTML

Расширяемый язык гипертекстовой разметки (Extensible HyperText Markup Language, XHTML) переопределяет HTML как разновидность XML. Правила написания XHTML-кода строже в сравнении с HTML, это касается и элементов `<script>` с внедренным JavaScript-кодом. Например, следующий код допустим в HTML, но недопустим в XHTML:

```
<script type="text/javascript">
    function compare(a, b) {
        if (a < b) {
            alert("A is less than B");
        } else if (a > b) {
            alert("A is greater than B");
        } else {
```

```
            alert("A is equal to B");
        }
    }
</script>
```

В HTML есть специальные правила синтаксического анализа содержимого элемента `<script>`, которые в XHTML не применяются. Из-за этого символ «меньше» (`<`) в инструкции `a < b` интерпретируется как начало тега, что приводит к синтаксической ошибке (за символом «меньше» не может следовать пробел).

Есть два способа исправить эту синтаксическую ошибку в XHTML-коде. Первый — заменить все экземпляры символа «меньше» (`<`) его HTML-мнемоникой (`<`):

```
<script type="text/javascript">
    function compare(a, b) {
        if (a &lt; b) {
            alert("A is less than B");
        } else if (a > b) {
            alert("A is greater than B");
        } else {
            alert("A is equal to B");
        }
    }
</script>
```

Хотя этот код будет работать на XHTML-странице, читать его сложнее. К счастью, есть другой способ: поместить JavaScript-код в раздел `CDATA`. В XHTML (и XML) разделы `CDATA` применяются для указания областей документа с произвольным текстом, который исключается из синтаксического анализа. Это позволяет использовать любые знаки, включая символ «меньше», и ошибок не возникает:

```
<script type="text/javascript"><![CDATA[
    function compare(a, b) {
        if (a < b) {
            alert("A is less than B");
        } else if (a > b) {
            alert("A is greater than B");
        } else {
            alert("A is equal to B");
        }
    }
]]></script>
```

В веб-браузерах, совместимых с XHTML, это устраняет проблему, однако многие браузеры все еще не поддерживают XHTML и раздел `CDATA`. Для обходного решения этой проблемы нужно добавить перед разметкой `CDATA` JavaScript-комментарий:

```
<script type="text/javascript">
//<![CDATA[
    function compare(a, b) {
        if (a < b) {
            alert("A is less than B");
        } else if (a > b) {
```

```
            alert("A is greater than B");
        } else {
            alert("A is equal to B");
        }
    }
//]]>
</script>
```

Данный способ работает во всех современных браузерах. Конечно, это нестандартный прием, но он соответствует требованиям XHTML и не вызывает проблем в браузерах, не поддерживающих XHTML.

Режим XHTML включается, если для страницы задан MIME-тип «application/xhtml+xml». Не все браузеры официально поддерживают такой XHTML-код.

Устаревший синтаксис

С появлением элемента `<script>` потребовалось обновить традиционный способ синтаксического анализа HTML-кода. К содержимому элемента нужно было применять специальные правила, из-за чего возникали проблемы в браузерах, не поддерживающих JavaScript, таких как Mosaic. Эти браузеры просто выводили содержимое элементов `<script>` на страницах, что портило все впечатление.

Компании Netscape и Mosaic совместно разработали решение, скрывающее встроенный JavaScript-код от браузеров, которые его не поддерживали. Было предложено заключать код сценариев в HTML-комментарии:

```
<script><!--
    function sayHi(){
        alert("Hi!");
    }
//--></script>
```

В результате браузеры вроде Mosaic безопасно игнорировали содержимое тегов `<script>`, а браузеры, поддерживающие JavaScript, распознавали этот шаблон, извлекая код из комментариев.

Хотя этот формат все еще распознается и правильно интерпретируется всеми веб-браузерами, он больше не нужен, поэтому использовать его не следует. Кроме того, в режиме XHTML такой код оказывается скрытым в правильных XML-комментариях и поэтому игнорируется.

Встроенный код или внешние файлы?

Хотя JavaScript-сценарии можно внедрять непосредственно в HTML-страницы, считается, что лучше включать JavaScript-код из внешних файлов. Хотя это не является незыблемым принципом, здесь приведены некоторые аргументы в пользу такого подхода.

- **Удобство сопровождения.** JavaScript-код, разбросанный по многим HTML-страницам, трудно сопровождать. Гораздо проще создать один каталог для всех JavaScript-файлов, чтобы разработчики могли редактировать код независимо от разметки, в которой он используется.
- **Кэширование.** Браузеры кэшируют все связанные внешние JavaScript-файлы, и если в двух страницах используется один файл, он загружается только один раз, что ускоряет загрузку страниц.
- **Готовность к будущему.** При включении сценариев из внешних файлов не нужно использовать упомянутые ранее приемы с комментариями и XHTML-кодом. Синтаксис включения в HTML и XHTML внешних файлов одинаков.

Режимы документа

В Internet Explorer 5.5 были представлены режимы документа, выбираемые путем переключения типов документа. Первыми двумя режимами были *режим совместимости* (quirks mode), в котором Internet Explorer работал как версия 5 (с несколькими нестандартными функциональными возможностями), и *стандартный режим* (standards mode), в котором Internet Explorer работал согласно стандартам. Хотя основные различия этих режимов связаны с использованием CSS, косвенно они затронули и JavaScript. Эти побочные эффекты обсуждаются на многих страницах книги.

Вслед за Internet Explorer режимы документа были реализованы и в других браузерах, при этом появился третий режим, названный *почти стандартным* (almost standards mode). Он во многом напоминает стандартный режим, но менее строг. Основное отличие проявляется в том, как обрабатываются интервалы вокруг изображений, что наиболее заметно, если изображения находятся в таблицах.

Режим совместимости включается во всех браузерах, если в начале документа не указан его тип. Это считается плохой практикой, потому что реализация режима совместимости сильно различается в браузерах и добиться их согласованной работы в этом режиме очень непросто.

Стандартный режим включается для следующих типов документов:

```
<!-- HTML 4.01 Strict -->
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">

<!-- XHTML 1.0 Strict -->
<!DOCTYPE html PUBLIC
"-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- HTML5 -->
<!DOCTYPE html>
```

Почти стандартный режим включается при указании в качестве типов документов `transitional` и `frameset`:

```
<!-- HTML 4.01 Transitional -->
<!DOCTYPE HTML PUBLIC
"-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">

<!-- HTML 4.01 Frameset -->
<!DOCTYPE HTML PUBLIC
"-//W3C//DTD HTML 4.01 Frameset//EN"
"http://www.w3.org/TR/html4/frameset.dtd">

<!-- XHTML 1.0 Transitional -->
<!DOCTYPE html PUBLIC
"-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<!-- XHTML 1.0 Frameset -->
<!DOCTYPE html PUBLIC
"-//W3C//DTD XHTML 1.0 Frameset//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-frameset.dtd">
```

Из-за того что стандартный и почти стандартный режимы так похожи, обычно их не различают. Когда говорят о «стандартном режиме», речь может идти о любом из них, и определение типа документа (которое мы обсудим позже) также не проводит различие между ними. В этой книге под *стандартным режимом* (standards mode) понимается любой режим, кроме режима совместимости.

Элемент <noscript>

В ранних браузерах, не поддерживающих JavaScript, очень важно было элегантно переключаться на упрощенные версии страниц. Для этого был создан элемент `<noscript>`, позволяющий предоставлять альтернативный контент таким браузерам. Он может содержать любые HTML-элементы (кроме `<script>`), которые могут быть добавлены в элемент `<body>` документа. Любое содержимое элемента `<noscript>` выводится на экран только в двух следующих ситуациях:

- браузер не поддерживает сценарии;
- поддержка сценариев в браузере отключена.

Если одно из этих условий выполнено, содержимое элемента `<noscript>` визуализируется, в противном случае он пропускается.

Рассмотрим простой пример:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Example HTML Page</title>
    <script type="text/javascript" defer="defer" src="example1.js"></script>
    <script type="text/javascript" defer="defer" src="example2.js"></script>
  </head>
```

```
  <body>
    <noscript>
      <p> This page requires a JavaScript-enabled browser.</p>
    </noscript>
  </body>
</html>
```

Если поддержка сценариев недоступна, этот код выводит на экран следующее сообщение, извещающее, что для вывода страницы нужен браузер с поддержкой JavaScript:

```
This page requires a JavaScript-enabled browser
```

В браузерах с поддержкой сценариев сообщение не выводится, хотя и является частью страницы.

Резюме

JavaScript-код вставляется в HTML-страницы с помощью элемента `<script>`. Он позволяет встраивать сценарии непосредственно в разметку HTML-страницы или использовать сценарии из внешних файлов.

- Чтобы включить в страницу внешний JavaScript-файл, назначьте атрибуту `src` URL-адрес файла, который может находиться как на сервере со страницей-контейнером, так и в другом домене.
- Все элементы `<script>` интерпретируются в том порядке, в котором они расположены на странице. Если не используются атрибуты `defer` и `async`, браузер должен полностью интерпретировать содержимое одного элемента `<script>`, прежде чем сможет перейти к следующему.
- В случае неотложенных сценариев браузер должен завершить интерпретацию кода внутри элемента `<script>` перед продолжением визуализации остальной части страницы. По этой причине элементы `<script>` обычно располагают ближе к концу страницы: после основного контента и непосредственно перед закрывающим тегом `</body>`.
- С помощью атрибута `defer` можно отложить выполнение сценария до завершения визуализации страницы. Отложенные сценарии всегда выполняются в том порядке, в котором они указаны.
- С помощью атрибута `async` можно указать, что сценарий не должен дожидаться других сценариев и блокировать визуализацию документа. Асинхронные сценарии могут выполняться не в том порядке, в котором они расположены на странице.

Элемент `<noscript>` позволяет задать контент, выводимый на экран только в том случае, если браузер не поддерживает сценарии. Если сценарии в браузере поддерживаются, содержимое элемента `<noscript>` не визуализируется.

Основы языка

- Обзор синтаксиса
- Типы данных
- Управляющие инструкции
- Функции

Ядром любого языка является спецификация его работы на самом базовом уровне. Как правило, она определяет синтаксис, операторы, типы данных и встроенный функционал, на основе которых можно создавать сложные решения. В стандарте ECMA-262 все эти элементы определены для JavaScript в форме псевдоязыка, который называется ECMAScript.

В большинстве веб-браузеров реализована версия ECMAScript из третьей редакции ECMA-262. На очереди пятая редакция, которая на конец 2011 года ни в одном браузере не реализована полностью. Сведения в этой главе основаны преимущественно на третьей редакции ECMAScript, а отличия пятой редакции описаны в выносках.

Синтаксис

Синтаксис ECMAScript во многом похож на C и другие C-подобные языки, такие как Java и Perl. Если вы знакомы с ними, вам будет легко привыкнуть к более свободному синтаксису ECMAScript.

Чувствительность к регистру

В ECMAScript все элементы, включая имена переменных, функций и операторов, чувствительны к регистру. Например, переменные `test` и `Test` различны, а ключевое слово `typeof` не может быть именем функции, тогда как `typeOf` — нормальное имя.

Идентификаторы

Идентификатор (identifier) — это имя переменной, функции, свойства или аргумента функции. Идентификаторы могут состоять из одного или нескольких знаков, удовлетворяющих двум условиям:

- первый знак должен быть буквой, знаком подчеркивания (_) или знаком доллара ($);
- все остальные знаки могут быть буквами, знаками подчеркивания, знаками доллара или цифрами.

В идентификаторах можно использовать буквы из расширенного набора ASCII или из Юникода, такие как À и Æ, но это не рекомендуется.

В ECMAScript-идентификаторах применяется «верблюжья» нотация. Это означает, что первая буква является строчной, а первые буквы всех последующих слов — прописными, например:

```
firstSecond
myCar
doSomethingImportant
```

Хотя это не является требованием, рекомендуется следовать этому правилу, чтобы не отступать от формата встроенных функций и объектов ECMAScript.

Ключевые слова, зарезервированные слова и значения true, false и null не могут быть идентификаторами (см. далее раздел «Ключевые и зарезервированные слова»).

Комментарии

ECMAScript поддерживает однострочные и блочные комментарии в стиле C. Для ввода однострочного комментария используются две косые черты:

```
// однострочный комментарий
```

Блочный комментарий начинается с косой черты и звездочки (/*), а заканчивается ими же в обратном порядке (*/):

```
/*
 * Это многострочный
 * комментарий
 */
```

Звездочки во второй и третьей строках в этом примере необязательны и добавлены исключительно для удобства чтения (такой формат обычно применяется в корпоративных приложениях).

Строгий режим

В ECMAScript 5 представлена концепция *строгого режима* (strict mode) — особой модели синтаксического анализа и выполнения JavaScript-кода, в которой исправлены некоторые неправильные аспекты работы ECMAScript и генерируются ошибки при небезопасных действиях. Чтобы включить строгий режим для всего сценария, добавьте в начало файла следующую команду:

```
"use strict"
```

Хотя она выглядит как строка, которую забыли присвоить переменной, на самом деле это директива, переводящая JavaScript в строгий режим. Такой синтаксис был выбран специально, чтобы исключить конфликты с ECMAScript 3.

Строгий режим можно включить и для отдельной функции, добавив эту директиву в начало тела функции:

```
function doSomething(){
    "use strict";
    // тело функции
}
```

В строгом режиме выполнение JavaScript-кода заметно меняется, и мы не раз с этим столкнемся. Строгий режим поддерживается в Internet Explorer 10+, Firefox 4+, Safari 5.1+, Opera 12+ и Chrome.

Инструкции

Инструкции в ECMAScript завершаются точками с запятой, хотя синтаксический анализатор сам способен определить конец инструкции, например:

```
var sum = a + b   // правильно даже без точки с запятой, но не рекомендуется
var diff = a - b; // правильно и рекомендуется
```

Хотя точки с запятой в конце инструкций необязательны, рекомендуется всегда добавлять их. Это предотвращает некоторые ошибки, например незавершенный ввод, и позволяет сжимать ECMAScript-код за счет удаления пустых мест (без точек с запятой это приводит к синтаксическим ошибкам). Кроме того, это препятствует снижению быстродействия, потому что синтаксические анализаторы пытаются исправлять предполагаемые ошибки, добавляя недостающие точки с запятой.

Как и в C, при помощи фигурных скобок ({}) несколько инструкций можно объединить в блок кода:

```
if (test){
    test = false;
    alert(test);
}
```

В управляющих инструкциях вроде `if` блоки требуются, только если инструкций несколько, но на практике рекомендуется создавать блок даже для одной инструкции:

```
if (test)
    alert(test);  // допустимо, но чревато ошибками и не рекомендуется

if (test){        // предпочтительный способ
    alert(test);
}
```

Использование блоков кода с управляющими инструкциями проясняет намерения программиста и предотвращает ошибки при изменении кода.

Ключевые и зарезервированные слова

Стандарт ECMA-262 определяет набор *ключевых слов* (keywords), служащих для решения специализированных задач, таких как указание начала или конца управляющей инструкции или выполнение специфической операции. Ключевые слова нельзя использовать как идентификаторы или имена свойств. Вот их полный список (ключевое слово со звездочкой было добавлено в пятой редакции):

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

Кроме того, ECMA-262 содержит набор *зарезервированных слов* (reserved words), которые также нельзя использовать как идентификаторы или имена свойств. Хотя эти слова не имеют специфического применения в языке, они зарезервированы на будущее как потенциальные ключевые слова. Вот полный список зарезервированных слов из третьей редакции ECMA-262:

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

В пятой редакции список зарезервированных слов в нестрогом режиме сокращается до следующего:

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

В строгом режиме в пятой редакции в этот список добавляются следующие слова:

```
implements    package       public
interface     private       static
let           protected     yield
```

Обратите внимание, что слова `let` и `yield` были зарезервированы в пятой редакции, а все остальные — в третьей. Ради совместимости рекомендуется брать за ориентир список из третьей редакции, добавляя в него слова `let` и `yield`.

Попытка использовать ключевое слово как имя идентификатора в реализациях ECMAScript 3 приводит к ошибке «Identifier Expected» (ожидается идентификатор). Применение зарезервированного слова с этой же целью в одних реализациях допускается, а в других приводит к ошибке.

В пятой редакции немного изменены правила употребления ключевых и зарезервированных слов. Они по-прежнему не могут быть идентификаторами, но теперь их разрешено использовать как имена свойств в объектах. В общем, для обеспечения совместимости с прошлыми и будущими редакциями ECMAScript лучше не использовать ключевые и зарезервированные слова как идентификаторы и имена свойств.

Кроме ключевых и зарезервированных слов в пятой редакции ECMA-262 налагаются ограничения на имена `eval` и `arguments`. В строгом режиме они не могут быть идентификаторами и именами свойств, иначе возникнет ошибка.

Переменные

ECMAScript-переменные типизированы слабо, то есть могут содержать данные любого типа. Каждая переменная — это просто именованный заполнитель для значения. Для определения переменной используется оператор `var` (заметьте, что это одно из ключевых слов), после которого указывается имя (идентификатор) переменной, например:

```
var message;
```

Здесь определяется переменная с именем `message`, которая может содержать любое значение (без инициализации она содержит специальное значение `undefined`, описанное в следующем разделе). ECMAScript поддерживает инициализацию переменных, то есть можно одновременно определить переменную и присвоить ей значение, например:

```
var message = "hi";
```

Здесь определяется переменная `message` для хранения строки `"hi"`. Инициализация не превращает переменную в строковую, она просто присваивает ей значение. После

инициализации можно не только изменить хранящееся в переменной значение, но и тип этого значения, например:

```
var message = "hi";
message = 100;          // допустимо, но не рекомендуется
```

В этом примере переменная `message` сначала определяется как строковое значение `"hi"`, а затем перезаписывается числовым значением 100. Хотя изменять тип данных, содержащихся в переменной, не рекомендуется, в ECMAScript это возможно.

Важно отметить, что при определении переменной с помощью оператора `var` она становится локальной в текущей области видимости. Например, если определить переменную с оператором `var` внутри функции, она будет уничтожена при выходе из функции:

```
function test(){
    var message = "hi";  // локальная переменная
}
test();
alert(message);          // ошибка!
```

Здесь переменная `message` определяется с помощью оператора `var` в функции `test()`. При создании переменной ей присваивается значение, но сразу же после этого она уничтожается, из-за чего в последней строке возникает ошибка. Однако переменную можно определить глобально, просто опустив оператор `var`:

```
function test(){
    message = "hi";      // глобальная переменная
}
test();
alert(message);          // "hi"
```

Теперь переменная `message` определена как глобальная. При вызове функции `test()` она инициализируется и становится доступна вне функции.

Определять глобальные переменные, опуская оператор var, не рекомендуется. Код с глобальными переменными, определенными локально, трудно разбирать и сопровождать, потому что непонятно, пропущен оператор var специально или случайно. В строгом режиме при попытке присвоить значение необъявленной переменной возникает ошибка ReferenceError.

В одной инструкции можно определить сразу несколько переменных, разделив их (с инициализацией или без нее) запятыми:

```
var message = "hi",
    found = false,
    age = 29;
```

Здесь объявляются и инициализируются три переменные. Поскольку ECMAScript типизирован слабо, в одной инструкции переменные можно инициализировать значениями разных типов. Чтобы облегчить чтение кода, можно разделить строку на несколько и добавить отступы, но это не требуется.

В строгом режиме определить переменную с именем `eval` или `arguments` нельзя. Попытка сделать это приведет к синтаксической ошибке.

Типы данных

В ECMAScript есть пять простых типов данных, также называемых *примитивными типами* (primitive types): неопределенный (undefined), нулевой (null), логический (boolean), числовой (number) и строковый (string). Есть также один сложный тип данных (object), который представляет собой неупорядоченный список пар имен и значений. Поскольку определить собственные типы данных в ECMAScript нельзя, все значения представляются с помощью одного из этих шести типов. Может показаться, что этого недостаточно, но у типов данных в ECMAScript есть динамические аспекты, благодаря которым каждый из них работает сразу за нескольких.

Оператор typeof

Поскольку ECMAScript типизирован слабо, для работы с ним необходим какой-то способ определения типа данных переменной. Для получения этой информации можно применить к значению оператор `typeof`, который возвращает одну из следующих строк:

- `"undefined"`, если значение не определено;
- `"boolean"`, если значение имеет логический тип;
- `"string"`, если значение является строкой;
- `"number"`, если значение является числом;
- `"object"`, если значение является объектом (отличным от функции) или значением `null`;
- `"function"`, если значение является функцией.

Оператор `typeof` вызывается следующим образом:

Листинг TypeofExample01.htm


```
var message = "some string";
alert(typeof message);     // "string"
alert(typeof(message));    // "string"
alert(typeof 95);          // "number"
```

Здесь оператору `typeof` передаются переменная (`message`) и числовой литерал. Поскольку `typeof` — это оператор, а не функция, заключать операнды в скобки не требуется (хотя можно).

Иногда `typeof` возвращает странные, но технически правильные значения. Так, `typeof null` возвращает строку `"object"`, потому что специальное значение `null` считается ссылкой на пустой объект. В Safari до версии 5 (включительно) и Chrome до версии 7 (включительно) оператор `typeof` возвращает для регулярного выражения значение `"function"`, а во всех остальных браузерах — `"object"`.

Технически функции считаются в ECMAScript объектами, а не отдельным типом данных. Однако они имеют некоторые специальные свойства, вследствие чего оператор typeof отличает их от других объектов.

Тип Undefined

Неопределенный тип (undefined) содержит единственное специальное значение `undefined`. Такое значение имеет переменная, объявленная с помощью оператора `var`, но не инициализированная:

Листинг UndefinedExample01.htm


```
var message;
alert(message == undefined);    // true
```

Здесь переменная `message` объявляется без инициализации. Сравнение с литеральным значением `undefined` показывает, что они равны. Этот пример идентичен следующему:

Листинг UndefinedExample02.htm

```
var message = undefined;
alert(message == undefined);    // true
```

Теперь переменная `message` явно инициализируется значением `undefined`, но это не требуется, потому что по умолчанию любая переменная без инициализации получает значение `undefined`.

Вообще говоря, переменным не следует явно присваивать значение undefined. Оно предоставляется преимущественно для сравнения и было добавлено только в третьей редакции ECMA-262, чтобы формализовать различие между указателем на пустой объект (null) и неинициализированной переменной.

Имейте в виду, что переменная со значением `undefined` отличается от переменной, которая вообще не определена. Рассмотрим пример:

Листинг UndefinedExample03.htm

```
var message;        // переменная объявляется, но имеет значение undefined

// следующая переменная не объявляется
// var age

alert(message);     // "undefined"
alert(age);         // ошибка
```

В этом примере в первом оповещении выводится значение переменной `message`, то есть `"undefined"`. Во втором случае в функцию `alert()` передается необъявленная переменная `age`, что приводит к ошибке. Для необъявленной переменной возможна только одна полезная операция: вызов оператора `typeof` (вызов `delete` для необъявленной переменной не приведет к ошибке в нестрогом режиме, но пользы от этого никакой).

И для неинициализированной, и для необъявленной переменной оператор `typeof` возвращает значение `"undefined"`, что немного запутывает. Взгляните на следующий пример:

Листинг UndefinedExample04.htm


```
var message;    // переменная объявляется, но имеет значение undefined

// следующая переменная не объявляется
// var age

alert(typeof message);  // "undefined"
alert(typeof age);      // "undefined"
```

В обоих случаях `typeof` возвращает строку `"undefined"`. Определенный смысл в этом есть, потому что никаких реальных операций нельзя выполнить ни для одной из переменных, хотя технически они совершенно разные.

Несмотря на то что неинициализированные переменные автоматически получают значение undefined, рекомендуется всегда выполнять инициализацию. Так вы будете знать, что оператор typeof возвращает строку «undefined» из-за того, что переменная не была объявлена, а не потому, что она просто не инициализирована.

Тип Null

Нулевой тип (Null) также содержит единственный элемент — специальное значение `null`. Логически `null` — это указатель на пустой объект, поэтому оператор `typeof` возвращает для него строку `"object"`:

Листинг NullExample01.htm

```
var car = null;
alert(typeof car);    // "object"
```

При определении переменной, которая позднее будет содержать объект, рекомендуется инициализировать ее именно значением `null`. Это позволяет явно проверять, была ли назначена переменной ссылка на объект, например:

```
if (car != null){
    // какие-то действия с car
}
```

Значение `undefined` является производным от `null`, так что в ECMA-262 они определены как нестрого равные:

Листинг NullExample02.htm

```
alert(null == undefined);  // true
```

При сравнении значений `null` и `undefined` с помощью оператора `==` всегда возвращается `true`, но помните, что этот оператор преобразует свои операнды (см. далее).

Несмотря на то что значения `null` и `undefined` связаны, используются они поразному. Как уже отмечалось, никогда не следует явно присваивать переменной значение `undefined`, но к `null` это не относится. Каждый раз, когда нужный объект недоступен, вместо него следует использовать `null`. Это отражает тот факт, что значение `null` было введено как указатель на пустой объект, и подчеркивает его отличие от `undefined`.

Тип boolean

Логический тип (boolean) — один из наиболее часто используемых в ECMAScript типов данных и имеет только два литеральных значения: `true` и `false`. Они отличаются от числовых значений: `true` не равно 1, а `false` не равно 0. Присвоить логические значения переменным можно следующим образом:

```
var found = true;
var lost = false;
```

Имейте в виду, что литералы `true` и `false` чувствительны к регистру, так что `True` и `False` (и эти же слова с другими сочетаниями прописных и строчных букв) являются допустимыми идентификаторами, но не логическими значениями.

Хотя литеральных логических значений всего два, в ECMAScript логические эквиваленты есть у всех значений. Для преобразования значения в его логический эквивалент используется специальная функция приведения типов `Boolean()`:

Листинг BooleanExample1.htm

```
var message = "Hello world!";
var messageAsBoolean = Boolean(message);
```

В этом примере строка `message` преобразуется в логическое значение и сохраняется в переменной `messageAsBoolean`. Функция `Boolean()` может принимать и данные других типов, но всегда возвращает логическое значение. Правила преобразования значения в `true` или `false` зависят как от самого значения, так и от его типа (см. таблицу).

Тип данных	Значения, преобразуемые в True	Значения, преобразуемые в False
boolean	true	false
string	Любая непустая строка	"" (пустая строка)
number	Любое ненулевое число (включая бесконечность)	0, NaN (см. раздел «NaN» далее в этой главе)
object	Любой объект	null
undefined	—	undefined

Важно понимать эти преобразования, потому что управляющие инструкции вроде `if` выполняют их автоматически, например:

Листинг BooleanExample02.htm


```
var message = "Hello world!";
if (message){
    alert("Value is true");
}
```

В этом примере оповещение выводится на экран, потому что строка `message` автоматически преобразуется в логический эквивалент (`true`). Внимательно следите за тем, какие переменные используются в управляющих инструкциях. Ошибочное указание объекта вместо логического значения может радикально изменить логику приложения.

Тип number

Пожалуй, наиболее интересным типом данных в ECMAScript является числовой (number). Он служит для представления целых чисел и чисел с плавающей точкой (которые в ряде языков называются числами с двойной точностью) в формате IEEE-754. Для поддержки чисел разных типов предусмотрено несколько разных форматов числовых литералов.

Самый простой из них — формат десятичного числа, которое можно ввести непосредственно:

```
var intNum = 55;    // целое число
```

Целые числа также можно представлять как восьмеричные или шестнадцатеричные литералы. В восьмеричном литерале первой цифрой является нуль (0), за которым

следует последовательность восьмеричных цифр (от 0 до 7). Если в литерале обнаруживается цифра не из этого диапазона, начальный нуль игнорируется и число интерпретируется как десятичное, например:

```
var octalNum1 = 070;  // 56 в восьмеричном формате
var octalNum2 = 079;  // недопустимое восьмеричное значение -
                      // интерпретируется как 79
var octalNum3 = 08;   // недопустимое восьмеричное значение -
                      // интерпретируется как 8
```

В строгом режиме попытка использовать восьмеричные литералы приведет к синтаксической ошибке.

Шестнадцатеричный литерал содержит знаки 0x в первых двух позициях (не чувствительны к регистру), а затем — любое количество шестнадцатеричных цифр (от 0 до 9 и от A до F). Буквы могут быть и строчными, и прописными, например:

```
var hexNum1 = 0xA;     // 10 в шестнадцатеричном формате
var hexNum2 = 0x1f;    // 31 в шестнадцатеричном формате
```

Числа, созданные в восьмеричном или шестнадцатеричном формате, во всех арифметических операциях используются как десятичные.

Способ хранения чисел в JavaScript позволяет представить положительный нуль (+0) и отрицательный нуль (–0). Они всегда эквивалентны, но помечаются в книге знаками для ясности.

Значения с плавающей точкой

Чтобы определить значение с плавающей точкой, необходимо ввести десятичную точку и как минимум одну цифру после нее. Нуль перед десятичной точкой необязателен, но лучше его указывать. Вот некоторые примеры:

```
var floatNum1 = 1.1;
var floatNum2 = 0.1;
var floatNum3 = .1;    // допустимо, но не рекомендуется
```

Из-за того что для хранения значений с плавающей точкой требуется вдвое больше памяти, чем для целых чисел, ECMAScript по возможности преобразует значения в целые числа. Если после десятичной точки нет разрядов, число становится целым. Если значение не имеет дробной части (например, 1.0), оно также преобразуется в целое число, например:

```
var floatNum1 = 1.;    // нет разрядов после десятичной точки -
                       // интерпретируется как целое число 1
var floatNum2 = 10.0;  // нет дробной части -
                       // интерпретируется как целое число 10
```

Очень большие и очень малые числа с плавающей точкой можно представлять в *экспоненциальном формате* (e-notation), в котором значения умножаются на 10 в соответствующей степени. В ECMAScript значение в экспоненциальном формате состоит из числа (целого или с плавающей точкой), прописной или строчной буквы E и показателя степени числа 10, например:

```
var floatNum = 3.125e7  // 31250000
```

В этом примере в экспоненциальном формате записано число 31 250 000. Можете понимать эту запись как умножение числа 3.125 на 10^7.

Экспоненциальный формат можно также использовать для представления очень малых чисел, таких как 0.00000000000000003, что можно сокращенно записать как 3e-17. По умолчанию ECMAScript переводит в экспоненциальный формат любые значения с плавающей точкой, содержащие как минимум шесть нулей после точки (например, 0.0000003 преобразуется в 3e-7).

Значения с плавающей точкой представляются с точностью до 17-го десятичного разряда, но все равно они гораздо менее точны в арифметических вычислениях, чем целые числа. Например, сложение 0.1 и 0.2 дает в результате 0.30000000000000004 вместо 0.3. Такие небольшие ошибки округления затрудняют проверку конкретных значений с плавающей точкой, например:

```
if (a + b == 0.3){      // не делайте так!
    alert("You got 0.3.");
}
```

Здесь сумма двух чисел проверяется на равенство 0.3. Этот код правильно сработает, если сложить 0.05 и 0.25 или 0.15 и 0.15, но для чисел 0.1 и 0.2 будет получен чуть больший результат и оповещение не появится. Никогда не проверяйте, равно ли значение с плавающей точкой конкретному числу.

Ошибки округления — это побочный эффект арифметических операций над числами с плавающей точкой формата IEEE-754, а не особенность ECMAScript. В других языках, в которых используется этот формат, наблюдаются такие же проблемы.

Диапазон значений

Из-за ограничений памяти в ECMAScript используется не весь диапазон чисел. Наименьшее и наибольшее числа, которые могут быть представлены в ECMAScript, хранятся в свойствах `Number.MIN_VALUE` и `Number.MAX_VALUE` и равны в большинстве браузеров 5e-324 и 1.7976931348623157e+308 соответственно. Результат вычисления, не попадающий в диапазон чисел JavaScript, автоматически приравнивается к специальному значению `Infinity`. Любое отрицательное число, которое не может быть представлено, считается отрицательной бесконечностью (`-Infinity`), а положительное — положительной бесконечностью (`Infinity`).

Если вычисление возвращает одну из бесконечностей, это значение не может использоваться ни в каких дальнейших вычислениях, потому что `Infinity` не имеет числового представления. Чтобы определить, конечно ли значение, можно задействовать функцию `isFinite()`. Она возвращает `true`, только если ее аргумент находится между минимальным и максимальным значениями:

```
var result = Number.MAX_VALUE + Number.MAX_VALUE;
alert(isFinite(result));    // false
```

Хотя значения, не попадающие в диапазон конечных чисел, применяются в вычислениях редко, это все же случается. Работая с очень большими или очень малыми числами, будьте внимательны.

Положительную и отрицательную бесконечности можно получить с помощью свойств Number.NEGATIVE_INFINITY и Number.POSITIVE_INFINITY, которые возвращают значения –Infinity и Infinity соответственно.

NaN

Специальное числовое значение `NaN`, то есть *не число* (Not a Number), указывает, что операция, которая должна была возвратить число, не сделала этого (но и не сгенерировала ошибку). Например, в большинстве других языков деление на 0 приводит к критической ошибке, но в ECMAScript вместо этого возвращается значение `NaN`, что позволяет продолжить работу.

Значение `NaN` имеет два уникальных свойства. Во-первых, любая операция с `NaN` (например, `NaN`/10) всегда возвращает `NaN`, что может быть проблемой, если вычисление выполняется в несколько этапов. Во-вторых, `NaN` не равно никакому значению, в том числе и другому `NaN`. Например, следующее выражение возвращает `false`:

```
alert(NaN == NaN);    // false
```

По этой причине ECMAScript предоставляет функцию `isNaN()`, которая принимает один аргумент любого типа и определяет, является ли он «не числом». При передаче значения в `isNaN()` предпринимается попытка преобразовать его в число. Некоторые нечисловые значения, например строка `"10"` или логическое значение, без проблем преобразуются в числа, а для аргументов, которые не могут быть преобразованы в числа, эта функция возвращает `true`, например:

Листинг NumberExample03.htm


```
alert(isNaN(NaN));      // true
alert(isNaN(10));       // false - 10 является числом
alert(isNaN("10"));     // false - может быть преобразовано в число 10
alert(isNaN("blue"));   // true  - не может быть преобразовано в число
alert(isNaN(true));     // false - может быть преобразовано в число 1
```

В этом примере проверяются пять разных значений. Первая проверка выполняется для самого значения `NaN` и, конечно, возвращает `true`. В следующих двух выражениях

проверяются число 10 и строка "10", и в обоих случаях возвращается false, потому что оба значения эквивалентны 10. Строка "blue" не может быть преобразована в число, поэтому следующий вызов возвращает false. Наконец, логическое значение true может быть преобразовано в число 1, поэтому последний вызов возвращает false.

Функцию isNaN() можно вызывать и для объектов, хотя обычно так не делают. В этом случае сначала вызывается метод valueOf(), чтобы определить, может ли возвращенное значение быть преобразовано в число. Если нет, вызывается метод toString() и проверяется возвращенное им значение. Таков типичный способ работы встроенных в ECMAScript функций и операторов (подробности см. в разделе «Операторы»).

Преобразование чисел

Есть три функции преобразования нечисловых значений в числа: Number(), parseInt() и parseFloat(). Функцию приведения типов Number() можно использовать с любым типом данных, а две другие функции служат для преобразования строк в числа. Каждая из них обрабатывает один и тот же ввод по-своему.

Функция Number()преобразует значения по следующим правилам.

- Логические значения true и false преобразуются в 1 и 0 соответственно.
- Числа возвращаются без изменений.
- Значение null преобразуется в 0.
- Значение undefined преобразуется в NaN.

Для строк действуют особые правила:

- Если строка содержит только цифры с начальным знаком «плюс» или «минус» либо без знака, она всегда преобразуется в десятичное число. Так, "1" преобразуется в 1, "123" — в 123, а "011" — в 11 (начальные нули игнорируются).
- Если строка содержит значение с плавающей точкой в правильном формате, такое как "1.1", она преобразуется в соответствующее число с плавающей точкой (начальные нули также игнорируются).
- Если строка содержит шестнадцатеричное значение в правильном формате, такое как "0xf", она преобразуется в соответствующее целое число.
- Если строка пуста (не содержит знаков), она преобразуется в 0.
- Если строка содержит что-то отличное от предыдущих вариантов, она преобразуется в NaN.
- Для объектов вызывается метод valueOf(), а возвращенное им значение преобразуется по предыдущим правилам. Если это преобразование дает результат NaN, вызывается метод toString() и применяются правила преобразования строк.

Уже по количеству правил для функции Number() ясно, что преобразование различных типов данных в числа — непростая задача. Вот несколько примеров:

Листинг NumberExample04.html


```
var num1 = Number("Hello world!");  // NaN
var num2 = Number("");              // 0
var num3 = Number("000011");        // 11
var num4 = Number(true);            // 1
```

Строка `"Hello world!"` преобразуется в `NaN`, потому что ей не соответствует никакое числовое значение, а для пустой строки возвращается 0. Из строки `"000011"` получается число 11, потому что начальные нули игнорируются, а из значения `true` — число 1.

Унарный оператор «плюс», описываемый далее в разделе «Операторы», работает так же, как функция Number().

Из-за сложных правил преобразования строк функцией `Number()` для получения целых чисел из строк лучше использовать функцию `parseInt()`. Она тщательнее проверяет строку, выясняя, соответствует ли она числовому шаблону. Начальные пробелы в строке игнорируются до первого символа, отличного от пробельного. Если этот первый символ не является числом, знаком «минус» или «плюс», функция `parseInt()` всегда возвращает `NaN` (в том числе для пустой строки, в отличие от функции `Number()`, которая возвращает 0). Если же первым символом является число, знак «плюс» или «минус», функция переходит ко второму символу, третьему, и так далее вплоть до конца строки или до нечислового символа. Например, строка `"1234blue"` преобразуется в 1234, потому что слово `"blue"` игнорируется. Аналогично `"22.5"` преобразуется в 22, потому что десятичная точка не используется в целых числах.

Если первый знак в строке — цифра, функция `parseInt()` также распознает различные форматы целых чисел (десятичный, восьмеричный и шестнадцатеричный). Так, строка, которая начинается с префикса `"0x"`, интерпретируется как шестнадцатеричное целое число, а если строка начинается с префикса `"0"`, после чего следует цифра, значение обрабатывается как восьмеричное.

Вот несколько примеров преобразования, поясняющих, что происходит:

Листинг NumberExample05.htm


```
var num1 = parseInt("1234blue");    // 1234
var num2 = parseInt("");            // NaN
var num3 = parseInt("0xA");         // 10 в шестнадцатеричном формате
var num4 = parseInt(22.5);          // 22
var num5 = parseInt("70");          // 70 в десятичном формате
var num6 = parseInt("0xf");         // 15 в шестнадцатеричном формате
```

Редакции ECMAScript 3 и 5 расходятся в вопросе применения функции `parseInt()` со строками, которые соответствуют восьмеричным литералам, например:

```
// 56 (восьмеричное значение) в ECMAScript 3; 0 (десятичное) в ECMAScript 5
var num = parseInt("070");
```

В реализациях ECMAScript 3 значение `"070"` обрабатывается как восьмеричный литерал и становится десятичным значением 56. В реализациях ECMAScript 5 функция `parseInt()` больше не поддерживает синтаксический анализ (parsing) восьмеричных значений, поэтому из-за начального нуля значение считается недействительным и интерпретируется как `"0"`, что дает в результате десятичное значение 0. Это верно даже для нестрогого режима ECMAScript 5.

Чтобы было проще следить за числовыми форматами, можно использовать с функцией `parseInt()` второй аргумент: основание системы счисления (иначе говоря, количество цифр в ней). Если вы знаете, что число является шестнадцатеричным, можете передать в функцию вместе с ним значение 16, чтобы число наверняка было обработано правильно:

```
var num = parseInt("0xAF", 16);    // 175
```

В этом случае можно опустить начальные знаки `"0x"`:

Листинг NumberExample06.htm

```
var num1 = parseInt("AF", 16);     // 175
var num2 = parseInt("AF");         // NaN
```

Здесь первое преобразование выполняется правильно, а второе нет, потому что первая функция `parseInt()` получает указание, что строка содержит шестнадцатеричное число. Вторая же функция выясняет, что первый знак не является цифрой, и автоматически прекращает работу.

Передача основания системы счисления может радикально изменить результат преобразования, например:

Листинг NumberExample07.html


```
var num1 = parseInt("10", 2);      // двоичное число 2
var num2 = parseInt("10", 8);      // восьмеричное число 8
var num3 = parseInt("10", 10);     // десятичное число 10
var num4 = parseInt("10", 16);     // шестнадцатеричное число 16
```

Поскольку при единственном аргументе функция `parseInt()` сама решает, как его интерпретировать, рекомендуется всегда передавать ей основание системы счисления во избежание ошибок.

Чаще всего вы будете передавать в функцию parseInt() десятичные числа, указывая 10 в качестве второго аргумента.

Функция `parseFloat()` работает подобно `parseInt()`, считывая каждый знак с нулевой позиции до конца строки или знака, которого не может быть в числе с плавающей точкой. Это означает, что одна десятичная точка допустима, но при обнаружении второй точки остальная часть строки игнорируется. Например, строка `"22.34.5"` преобразуется в 22.34.

Другое отличие функции `parseFloat()` в том, что она всегда пропускает начальные нули. Она распознает любые форматы чисел с плавающей точкой, описанные ранее, а также десятичный формат (начальные нули всегда игнорируются). Шестнадцатеричные числа всегда преобразуются в 0. Поскольку функция `parseFloat()` анализирует только десятичные значения, основание системы счисления она не принимает. Наконец, если строка содержит значение без десятичной точки или с нулем после нее, функция `parseFloat()` возвращает целое число. Вот несколько примеров:

Листинг NumberExample08.html

```
var num1 = parseFloat("1234blue");    // 1234 (целое число)
var num2 = parseFloat("0xA");         // 0
var num3 = parseFloat("22.5");        // 22.5
var num4 = parseFloat("22.34.5");     // 22.34
var num5 = parseFloat("0908.5");      // 908.5
var num6 = parseFloat("3.125e7");     // 31250000
```

Тип string

Строковый тип (string) — это последовательности 16-разрядных знаков Юникода (в том числе пустые). Строки могут быть заключены в двойные (") или одинарные (') кавычки:

```
var firstName = "Nicholas";
var lastName = 'Zakas';
```

В отличие от языка PHP, в котором интерпретация строки зависит от типа кавычек, в ECMAScript эти два варианта синтаксиса одинаковы, но кавычки в начале и конце строки не должны различаться. Например, такое выражение вызовет синтаксическую ошибку:

```
var firstName = 'Nicholas";    // синтаксическая ошибка - разные кавычки
```

Символьные литералы

Строковый тип данных поддерживает несколько литералов, представляющих полезные символы, в том числе непечатаемые. Они указаны в таблице.

Литерал	Значение
\n	Перевод строки
\t	Табуляция
\b	Возврат на одну позицию
\r	Возврат каретки
\f	Перевод страницы

Литерал	Значение
\\	Обратная косая черта (\)
\'	Одинарная кавычка ('). Используется, если строка заключена в одинарные кавычки, например: 'He said, \'hey.\''
\"	Двойная кавычка ("). Используется, если строка заключена в двойные кавычки, например: "He said, \"hey.\""
\xnn	Знак с шестнадцатеричным кодом nn (где n — шестнадцатеричный знак 0-F). Пример: \x41 is equivalent to "A"
\unnnn	Знак Юникода с шестнадцатеричным кодом nnnn (где n — шестнадцатеричный знак 0-F). Пример: \u03a3 is equivalent to the Greek character Σ

Символьные литералы могут находиться в любом месте строки и интерпретируются как один символ, например:

```
var text = "This is the letter sigma: \u03a3.";
```

В этом примере переменная `text` содержит 28 символов, хотя одна только экранирующая последовательность состоит из 6 символов. Однако она представляет один символ и считается одним символом.

Длину любой строки можно узнать с помощью свойства `length`:

```
alert(text.length);    //19
```

Это свойство возвращает количество 16-разрядных символов в строке. Если строка содержит символы удвоенной разрядности, свойство `length` может возвратить неправильный результат.

Природа строк

После создания строки в ECMAScript изменить ее значение невозможно. Для изменения строковой переменной первоначальная строка уничтожается, а затем переменной присваивается другая строка с новым значением:

```
var lang = "Java";
lang = lang + "Script";
```

Здесь переменная `lang` определяется со значением `"Java"`. В следующей строке она переопределяется как конкатенация строк `"Java"` и `"Script"` и получает значение `"JavaScript"`. Для этого создается новая строка с местом, достаточным для хранения 10 символов, и затем оно заполняется фрагментами `"Java"` и `"Script"`. Наконец, строки `"Java"` и `"Script"` уничтожаются, потому что они больше не нужны. Все это происходит за кулисами, и именно поэтому старые браузеры (например, Firefox до версии 1.0 и Internet Explorer 6.0) очень медленно выполняли конкатенацию строк. Более поздние версии этих браузеров обрабатывают строки эффективнее.

Преобразование значения в строку

Преобразовать значение в строку можно двумя способами. Первый — использовать метод `toString()`, который есть почти у любого значения (см. главу 5). Он просто возвращает строковый эквивалент значения, например:

Листинг StringExample01.htm


```
var age = 11;
var ageAsString = age.toString();      // строка "11"
var found = true;
var foundAsString = found.toString();  // строка "true"
```

Метод `toString()` доступен для чисел, логических значений, объектов и строк (да, у каждой строки есть метод `toString()`, который просто возвращает копию строки). Для значений `null` и `undefined` вызвать его нельзя.

В большинстве случаев у метода `toString()` нет аргументов, но при использовании с числами он может принимать один аргумент: основание целевой системы счисления. По умолчанию метод `toString()` всегда возвращает строковое представление числа в десятичном формате, но если задано основание, он может вывести значение с двоичным, восьмеричным, шестнадцатеричным и любым другим допустимым основанием, например:

Листинг StringExample02.htm

```
var num = 10;
alert(num.toString());    // "10"
alert(num.toString(2));   // "1010"
alert(num.toString(8));   // "12"
alert(num.toString(10));  // "10"
alert(num.toString(16));  // "a"
```

Этот пример показывает, как изменяется вывод метода `toString()` для чисел при изменении основания. Значение 10 может выводиться в самых разных числовых форматах, но по умолчанию (без аргумента) используется основание 10.

Если переменная способна принимать значение `null` или `undefined`, вы можете использовать функцию приведения типов `String()`, которая всегда возвращает строку независимо от полученного значения. Она работает следующим образом:

- если у значения есть метод `toString()`, он вызывается (без аргументов), а затем возвращается результат;
- для значения `null` возвращается строка `"null"`;
- для значения `undefined` возвращается строка `"undefined"`.

Рассмотрим следующий пример:

Листинг StringExample03.htm


```
var value1 = 10;
var value2 = true;
```

```
var value3 = null;
var value4;

alert(String(value1));    // "10"
alert(String(value2));    // "true"
alert(String(value3));    // "null"
alert(String(value4));    // "undefined"
```

Здесь в строки преобразуются число, логическое значение, значения `null` и `undefined`. Для числа и логического значения возвращается такой же результат, как если бы был вызван метод `toString()`. Поскольку для значений `"null"` и `"undefined"` он недоступен, для них метод `String()` просто возвращает текстовые литералы.

Также можно преобразовать значение в строку, добавив к нему пустую строку (" ") с помощью оператора «плюс» (см. далее раздел «Операторы»).

Тип Object

В ECMAScript объекты создаются как неспецифические сочетания данных и функциональности. Чтобы добавить в программу объект, нужно ввести оператор `new` и указать тип объекта. Для создания собственных объектов разработчики обычно создают экземпляры типа `Object`, а затем добавляют к ним свойства и (или) методы, например:

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

Этот синтаксис похож на Java, хотя в ECMAScript скобки нужны только при передаче аргументов в конструктор. Если аргументов нет, скобки можно опускать (однако это не рекомендуется):

```
var o = new Object;    // допустимо, но не рекомендуется
```

Сами по себе экземпляры типа `Object` не очень полезны, но важно понимать основы их работы, потому что подобно типу `java.lang.Object` в Java тип `Object` в ECMAScript является родительским для всех остальных объектов. Все его свойства и методы есть у других, более специфичных объектов.

Каждый экземпляр `Object` имеет свойства и методы из приведенного списка.

- `constructor` — функция, которая была использована для создания объекта. В предыдущем примере это функция `Object()`.
- `hasOwnProperty(имяСвойства)` — указывает, есть ли у объекта (не у прототипа) данное свойство. Имя свойства должно быть указано как строка (например, `o.hasOwnProperty("name")`).
- `isPrototypeOf(объект)` — определяет, является ли объект прототипом другого объекта (прототипы обсуждаются в главе 5).

- `propertyIsEnumerable(имяСвойства)` — указывает, можно ли перебирать данное свойство в инструкции `for-in` (см. далее). Как и в случае метода `hasOwnProperty()`, имя свойства должно быть строкой.
- `toLocaleString()` — возвращает строковое представление объекта в соответствии с региональными настройками среды выполнения.
- `toString()` — возвращает строковое представление объекта.
- `valueOf()` — возвращает строковый, численный или логический эквивалент объекта, часто совпадающий с результатом вызова `toString()`.

Поскольку тип `Object` является родительским для всех объектов в ECMAScript, эти базовые свойства и методы есть у каждого объекта. Подробные сведения об этом см. в главах 5 и 6.

Технически принципы работы объектов в ECMA-262 относятся не ко всем объектам в JavaScript. Объекты в среде браузера, например BOM- и DOM-объекты, предоставляются и определяются средой. На них не распространяются требования ECMA-262, а потому они могут не наследоваться от типа Object.

Операторы

Для работы с данными ECMA-262 предоставляет набор *операторов* (operators), которые варьируются от математических (таких, как сложение и вычитание) и поразрядных до операторов отношения и сравнения. ECMAScript-операторы уникальны в том смысле, что их можно задействовать со многими разными значениями, включая строки, числа, логические значения и даже объекты. Если операторы используются с объектами, для получения операндов обычно вызывается метод `valueOf()` и (или) `toString()`.

Унарные операторы

Операторы, работающие с единственным значением, называются *унарными* (unary operators). Это самые простые операторы в ECMAScript.

Инкремент и декремент

Операторы инкремента и декремента взяты непосредственно из C и имеют две версии: префиксную и постфиксную. Префиксные версии указываются перед операндами, постфиксные — после. Чтобы добавить 1 к числовой переменной с помощью оператора префиксного инкремента, введите перед именем переменной два знака «плюс» (`++`):

```
var age = 29;
++age;
```

В этом примере значение age изменяется на 30 (к предыдущему значению 29 добавляется 1), что эквивалентно следующему коду:

```
var age = 29;
age = age + 1;
```

Префиксный декремент работает похоже, вычитая 1 из числа. Чтобы использовать префиксный декремент, введите перед именем переменной два знака «минус» (--):

```
var age = 29;
--age;
```

Здесь переменная age уменьшается до 28 (из 29 вычитается 1).

При использовании префиксного инкремента или декремента значение переменной изменяется до вычисления выражения — в программировании это обычно называют *побочным эффектом* (side effect). Взгляните на следующий код:

Листинг IncrementDecrementExample01.htm


```
var age = 29;
var anotherAge = --age + 2;

alert(age);         // 28
alert(anotherAge);  // 30
```

В этом примере переменная anotherAge инициализируется суммой уменьшенного на единицу значения age и числа 2. Поскольку декремент выполняется первым, age получает значение 28, а затем к нему добавляется 2, что дает в итоге 30.

Префиксный инкремент и декремент имеют в инструкциях одинаковый приоритет и выполняются слева направо, например:

Листинг IncrementDecrementExample02.htm


```
var num1 = 2;
var num2 = 20;
var num3 = --num1 + num2;     // 21
var num4 = num1 + num2;       // 21
```

Переменная num3 равна 21, потому что значение num1 уменьшается на 1 перед сложением. Переменная num4 также равна 21, потому что суммируются уже измененные значения.

Постфиксные операторы инкремента и декремента имеют такой же синтаксис (соответственно ++ и --), но указываются после переменной, а не перед ней. Они отличаются от префиксных версий тем, что выполняются после вычисления инструкции, которая их содержит. Иногда это не имеет значения, например:

```
var age = 29;
age++;
```

Преобразование префиксного инкремента в постфиксный ничего не изменило, потому что эти инструкции больше ничего не делают. Однако если добавить другие операции, разница становится очевидной:

Листинг IncrementDecrementExample03.htm

```
var num1 = 2;
var num2 = 20;
var num3 = num1-- + num2;     // 22
var num4 = num1 + num2;       // 21
```

Как видите, аналогичное преобразование привело к изменению результата. В примере с префиксной версией переменные `num3` и `num4` были равны 21, тогда как здесь `num3` имеет значение 22, а `num4` — 21. Дело в том, что при вычислении `num3` слагаемое `num1` имеет первоначальное значение (2), а в инструкции с `num4` используется уменьшенное значение (1).

Все операторы инкремента и декремента работают не только с целыми числами, но и с любыми другими значениями: строками, логическими значениями, числами с плавающей точкой и объектами. При этом применяются такие правила.

- Если операнд — строка, представляющая допустимое число, она преобразуется в число, к которому применяется оператор. Строковая переменная становится числовой.
- Если операнд — строка, которая не является допустимым числом, переменная получает значение `NaN` (см. главу 4) и становится числовой.
- Если операнд — логическое значение `false`, оно преобразуется в 0, а затем применяется оператор. Логическая переменная становится числовой.
- Если операнд — логическое значение `true`, оно преобразуется в 1, а затем применяется оператор. Логическая переменная становится числовой.
- Если операнд — число с плавающей точкой, оно увеличивается или уменьшается на 1.
- Если операнд — объект, вызывается его метод `valueOf()` (см. главу 5) для получения допустимого операнда, после чего применяются другие правила. Если в результате получается `NaN`, вызывается метод `toString()` и снова применяются другие правила. Объект становится числовой переменной.

Некоторые из этих правил продемонстрированы в следующем примере:

Листинг IncrementDecrementExample04.htm


```
var s1 = "2";
var s2 = "z";
var b = false;
var f = 1.1;
var o = {
    valueOf: function() {
        return -1;
```

```
    }
};

s1++;  // результат – число 3
s2++;  // результат - NaN
b++;   // результат – число 1
f--;   // результат – 0.10000000000000009 (из-за неточностей округления)
o--;   // результат – число -2
```

Унарные плюс и минус

Унарные операторы плюс и минус известны всем со школы и работают в ECMAScript точно так же. Унарный плюс (+) указывается перед переменной, и если это число, он ничего не делает:

```
var num = 25;
num = +num    // по-прежнему 25
```

Если унарный плюс применяется к нечисловому значению, оно преобразуется так же, как и при вызове функции приведения типов `Number()`: логические значения `false` и `true` изменяются на 0 и 1, строковые значения анализируются согласно набору правил, а для объектов вызываются их методы `valueOf()` и (или) `toString()`.

Следующий пример поясняет применение унарного оператора плюс к разным типам данных:

Листинг UnaryPlusMinusExample01.htm

```
var s1 = "01";
var s2 = "1.1";
var s3 = "z";
var b = false;
var f = 1.1;
var o = {
    valueOf: function() {
        return -1;
    }
};

s1 = +s1;  // результат – число 1
s2 = +s2;  // результат – число 1.1
s3 = +s3;  // результат – NaN
b = +b;    // результат – число 0
f = +f;    // число не изменяется (1.1)
o = +o;    // результат – число -1
```

Унарный минус чаще всего используется для изменения знака числа, например:

```
var num = 25;
num = -num;    // -25
```

При использовании с числовым значением унарный минус просто изменяет его знак, как в этом примере. Если значение не является числом, применяются

те же правила, что и для унарного оператора плюс, а затем меняется знак результата:

Листинг UnaryPlusMinusExample02.htm


```
var s1 = "01";
var s2 = "1.1";
var s3 = "z";
var b = false;
var f =  1.1;
var o = {
    valueOf: function() {
        return -1;
    }
};

s1 = -s1;  // результат – число -1
s2 = -s2;  // результат – число -1.1
s3 = -s3;  // результат – NaN
b = -b;    // результат – число 0
f = -f;    // результат – число -1.1
o = -o;    // результат – число 1
```

Унарные операторы плюс и минус обычно применяются в простых арифметических операциях, но, как показывают примеры, могут использоваться и для преобразований.

Поразрядные операторы

Операторы из этого раздела работают с числами на самом низком уровне — уровне отдельных битов. Все числа в ECMAScript хранятся в 64-разрядном формате IEEE-754, но поразрядные операции не работают непосредственно с этим представлением. Вместо этого значение преобразуется в 32-разрядное число, для него выполняется нужная операция, а затем результат преобразуется обратно в 64-разрядный формат. Поскольку 64-разрядный формат прозрачен, для разработчика все выглядит так, как если бы существовали только 32-разрядные целые числа, которые мы сейчас и обсудим.

В целых числах со знаком все биты, кроме 32-го, представляют само значение, тогда как 32-й бит определяет знак числа: 0 для положительных чисел и 1 для отрицательных. Значение этого бита, называемого *знаковым* (sign bit), определяет формат остальной части числа. Положительные числа хранятся в настоящем двоичном формате, в котором все биты, кроме знакового, представляют степени двойки: первый бит (бит 0) соответствует 2^0, второй — 2^1, и т. д. Если какие-либо биты не используются, они считаются равными нулю и, по сути, игнорируются. Например, число 18 представляется как 00000000000000000000000000010010, или, сокращенно, как 10010. Эти пять значимых битов и определяют фактическое значение числа (рис. 3.1).

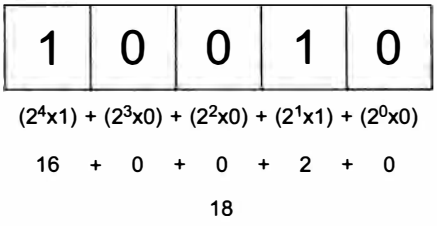

Рис. 3.1

Отрицательные числа также хранятся в двоичном коде, но в формате, который называется *дополнительным кодом* (two's complement). Он вычисляется в три этапа.

1. Определяется двоичное представление абсолютного значения числа (например, для числа –18 сначала определяется двоичное представление 18).
2. Находится обратный код числа. Это означает, что каждый ноль заменяется единицей и наоборот.
3. К результату добавляется 1.

Определим двоичное представление числа –18. Начнем с абсолютного значения (18):

```
0000 0000 0000 0000 0000 0000 0001 0010
```

Далее определим обратный код:

```
1111 1111 1111 1111 1111 1111 1110 1101
```

Наконец, добавим 1 к обратному коду числа:

```
1111 1111 1111 1111 1111 1111 1110 1101
                                      1
----------------------------------------
1111 1111 1111 1111 1111 1111 1110 1110
```

Итак, двоичным эквивалентом –18 является 11111111111111111111111111101110. Помните, что при работе с целыми числами со знаком бит 31 недоступен.

ECMAScript делает все возможное, чтобы скрыть от вас всю эту «кухню». Например, при выводе отрицательного числа в виде двоичной строки вы получаете двоичный код абсолютного значения со знаком «минус»:

```
var num = -18;
alert(num.toString(2));    // "-10010"
```

При преобразовании числа –18 в двоичную строку в результате получается –10010, поскольку предполагается, что этот формат более понятен, чем дополнительный код.

Когда поразрядные операторы применяются к числам, 64-разрядные числа преобразуются в 32-разрядные, выполняется операция, а затем 32-разрядный результат

снова сохраняется в 64-разрядном числе. Это создает иллюзию, что вы работаете с настоящими 32-разрядными числами, из-за чего двоичные операции в ECMAScript похожи на свои аналоги в других языках. У преобразований есть также любопытный побочный эффект: специальные значения `NaN` и `Infinity` в поразрядных операциях интерпретируются как 0.

По умолчанию все целые числа представляются в ECMAScript как числа со знаком. Существуют, однако, и целые числа без знака, в которых 32-й бит не указывает знак. Все такие числа положительные, но благодаря дополнительному биту они могут быть больше, чем целые числа со знаком.

Если поразрядный оператор применяется к нечисловому значению, оно автоматически преобразуется в число с помощью функции `Number()` и только затем выполняется операция. Ее результатом является число.

Поразрядное НЕ

Поразрядное НЕ (~) просто возвращает обратный код числа. Это один из немногих ECMAScript-операторов для двоичной математики. Рассмотрим пример:

Листинг BitwiseNotExample01.htm


```
var num1 = 25;     // двоичное число 00000000000000000000000000011001
var num2 = ~num1;  // двоичное число 11111111111111111111111111100110
alert(num2);       // -26
```

Здесь поразрядное НЕ применяется к числу 25, что дает в результате –26. По сути, число просто инвертируется, а затем новое значение уменьшается на 1. Тот же результат можно получить следующим образом:

```
var num1 = 25;
var num2 = -num1 - 1;
alert(num2);      // "-26"
```

Хотя результаты одинаковы, поразрядное НЕ выполняется гораздо быстрее, потому что оно работает на самом низком уровне.

Поразрядное И

Поразрядный оператор И (`&`) сопоставляет биты двух значений, а затем выполняет операцию И для каждой пары битов в одной позиции по правилам из следующей таблицы истинности:

Бит первого числа	Бит второго числа	Результат
1	1	1
1	0	0
0	1	0
0	0	0

Иначе говоря, бит результата равен 1, только если биты обоих операндов в этой позиции равны 1. Если хотя бы один из них равен 0, то и бит результата равен 0.

В следующем примере поразрядное И выполняется для чисел 25 и 3:

Листинг BitwiseAndExample01.htm


```
var result = 25 & 3;
alert(result);      // 1
```

В результате получается 1. Почему? Взгляните сами:

```
 25 = 0000 0000 0000 0000 0000 0000 0001 1001
  3 = 0000 0000 0000 0000 0000 0000 0000 0011
--------------------------------------------
AND = 0000 0000 0000 0000 0000 0000 0000 0001
```

Как видите, только в одной позиции биты обоих операндов равны 1. Из-за этого все остальные биты результата обнуляются, что в итоге дает 1.

Поразрядное ИЛИ

Поразрядный оператор ИЛИ (|) также работает с двумя операндами, при этом применяются правила из следующей таблицы истинности:

Бит первого числа	Бит второго числа	Результат
1	1	1
1	0	1
0	1	1
0	0	0

Поразрядное ИЛИ возвращает 1, если хотя бы один бит равен 1, и 0, если оба бита равны 0.

Рассмотрим выполнение поразрядного ИЛИ для чисел из предыдущего примера:

Листинг BitwiseOrExample01.htm


```
var result = 25 | 3;
alert(result);      // 27
```

Результат 27 получается следующим образом:

```
 25 = 0000 0000 0000 0000 0000 0000 0001 1001
  3 = 0000 0000 0000 0000 0000 0000 0000 0011
--------------------------------------------
 OR = 0000 0000 0000 0000 0000 0000 0001 1011
```

Каждый единичный бит любого из операндов переходит в результат. Двоичный код 11011 соответствует числу 27.

Поразрядное исключающее ИЛИ

Поразрядное исключающее ИЛИ (^) выполняется для двух операндов по правилам из следующей таблицы:

Бит первого числа	Бит второго числа	Результат
1	1	0
1	0	1
0	1	1
0	0	0

Оно отличается от обычного поразрядного ИЛИ тем, что возвращает 1, только если один бит равен 1 (если оба бита равны 1, возвращается 0).

Выполним исключающее ИЛИ для тех же чисел, 25 и 3:

Листинг BitwiseXorExample01.htm

```
var result = 25 ^ 3;
alert(result);      // 26
```

В результате получается 26:

```
 25 = 0000 0000 0000 0000 0000 0000 0001 1001
  3 = 0000 0000 0000 0000 0000 0000 0000 0011
--------------------------------------------
XOR = 0000 0000 0000 0000 0000 0000 0001 1010
```

Этот пример отличается от предыдущего только тем, что первый бит результата обнуляется, поскольку в обоих операндах он равен 1. Все остальные единичные биты переходят в результат, потому что у них нет пары во втором операнде. Двоичному коду 11010 соответствует число 26 (заметьте, что оно на единицу меньше, чем результат поразрядного ИЛИ).

Сдвиг влево

Оператор сдвига влево (<<) сдвигает все биты числа влево на указанное количество позиций. Например, если сдвинуть число 2 (10 в двоичном формате) на 5 битов влево, получится 64 (1000000 в двоичном формате):

Листинг LeftShiftExample01.htm

```
var oldValue = 2;              // 10 в двоичном формате
var newValue = oldValue << 5;  // 1000000 в двоичном формате (десятичное 64)
```

В этом примере при сдвиге числа влево справа остаются пять пустых битов, которые заполняются нулями, чтобы в результате получилось полное 32-разрядное число (рис. 3.2).

Рис. 3.2

Имейте в виду, что при сдвиге влево знак числа не меняется. Например, если сдвинуть –2 на пять битов влево, получится число –64, а не 64.

Сдвиг вправо с сохранением знака

Оператор сдвига вправо с сохранением знака (>>) в точности противоположен сдвигу влево. Например, если сдвинуть 64 вправо на 5 битов с сохранением знака, получится 2:

Листинг SignedRightShiftExample01.htm

```
var oldValue = 64;              // 1000000 в двоичном формате
var newValue = oldValue >> 5;  // 10 в двоичном формате или 2 в десятичном
```

При сдвиге вправо пустые биты появляются слева от числа, но справа от знакового бита (рис. 3.3). Чтобы получилось полное число, в них копируется знаковый бит.

Рис. 3.3

Сдвиг вправо с заполнением нулями

Для положительных чисел оператор сдвига вправо с заполнением нулями (>>>) эквивалентен сдвигу вправо с сохранением знака. Если сдвинуть 64 вправо на 5 битов с заполнением нулями, получится 2, как и в предыдущем примере:

Листинг UnsignedRightShiftExample01.htm

```
var oldVallue = 64;             // 1000000 в двоичном формате
var newValue = oldValue >>> 5;  // 10 в двоичном формате или 2 в десятичном
```

Отрицательные числа — совсем другое дело. В отличие от сдвига вправо с сохранением знака, теперь пустые биты заполняются нулями независимо от знака числа, что в случае отрицательных чисел дает другой результат. При сдвиге вправо с заполнением нулями отрицательное число в двоичном формате обрабатывается как положительное число. Поскольку отрицательное число является дополнением его абсолютного значения до двух, число становится очень большим, например:

Листинг UnsignedRightShiftExample02.htm

```
var oldValue = -64;       // двоичное число 11111111111111111111111111000000
var newValue = oldValue >>> 5;  // 134217726 в десятичном формате
```

Как видите, при сдвиге числа –64 вправо на 5 битов с заполнением нулями получается 134 217 726. Это происходит потому, что в двоичном формате –64 записывается как 11111111111111111111111111000000, которое в данном случае считается положительным числом 4 294 967 232. После сдвига мы получаем 0000011111111 11111111111111110, или десятичное число 134 217 726.

Логические операторы

Логические операторы относятся к важнейшим элементам любого языка программирования. Если бы нельзя было проверить, как соотносятся два значения, условные инструкции и циклы были бы бесполезны. Логических операторов три: НЕ, И и ИЛИ.

Логическое НЕ

Логический оператор НЕ (`!`) в ECMAScript можно применять к любым значениям. Он преобразует операнд в логическое значение, выполняет его отрицание и возвращает логическое значение, при этом действуют следующие правила:

- Если операнд — объект, возвращается `false`.
- Если операнд — пустая строка, возвращается `true`.
- Если операнд — непустая строка, возвращается `false`.
- Если операнд — число 0, возвращается `true`;
- Если операнд — любое число, отличное от 0 (включая `Infinity`), возвращается `false`.
- Если операнд — значение `null`, возвращается `true`.

- Если операнд — значение `NaN`, возвращается `true`.
- Если операнд — значение `undefined`, возвращается `true`.

Вот некоторые примеры:

Листинг LogicalNotExample01.htm


```
alert(!false);    // true
alert(!"blue");   // false
alert(!0);        // true
alert(!NaN);      // true
alert(!"");       // true
alert(!12345);    // false
```

С помощью двух логических НЕ можно также преобразовать значение в его логический эквивалент, что имитируют функцию приведения типов `Boolean()`. Первый оператор НЕ возвращает логическое значение независимо от типа операнда, а второй отрицает это значение, предоставляя логический аналог первоначальной переменной. Это дает тот же результат, что и вызов функции `Boolean()`:

Листинг LogicalNotExample02.htm


```
alert(!!"blue");  // true
alert(!!0);       // false
alert(!!NaN);     // false
alert(!!"");      // false
alert(!!12345);   // true
```

Логическое И

Логический оператор И (`&&`) применяется к двум значениям, например:

```
var result = true && false;
```

Он работает согласно следующей таблице истинности:

Операнд 1	Операнд 2	Результат
true	true	true
true	false	false
false	true	false
false	false	false

Логическое И действует с операндами любых типов. Если один из операндов не является примитивным логическим значением, логическое И не всегда возвращает логическое значение. Вместо этого применяются такие правила:

- Если первый операнд — объект, всегда возвращается второй операнд.
- Если второй операнд — объект, а первый эквивалентен значению `true`, возвращается этот объект.

- Если оба операнда — объекты, возвращается второй операнд.
- Если хотя бы один из операндов — значение `null`, возвращается `null`.
- Если хотя бы один из операндов — значение `NaN`, возвращается `NaN`.
- Если хотя бы один из операндов — значение `undefined`, возвращается `undefined`.

Логическое И поддерживает сокращенное вычисление: если первого операнда достаточно для определения результата, второй операнд не оценивается. Так, если первый операнд — `false`, то каким бы ни был второй операнд, результатом не может быть `true`. Рассмотрим пример:

Листинг LogicalAndExample01.htm


```
var found = true;
var result = (found && someUndeclaredVariable);    // ошибка
alert(result);    // эта строка никогда не выполняется
```

При выполнении логического И в этом коде возникает ошибка, потому что переменная `someUndeclaredVariable` не объявлена. Значение `found` равно `true`, поэтому интерпретатор переходит к оценке переменной `someUndeclaredVariable`, которая не объявлена, а потому не может использоваться в логическом И. Если изменить значение `found` на `false`, ошибка не возникает:

Листинг LogicalAndExample02.htm

```
var found = false;
var result = (found && someUndeclaredVariable);    // ошибки нет
alert(result);    // все работает
```

В этом случае оповещение выводится на экран. Переменная `someUndeclaredVariable` здесь тоже не определена, но она никогда не оценивается, потому что первый операнд равен `false` и, следовательно, результатом операции тоже может быть только `false`. Используя логическое И, не забывайте о сокращенной схеме его вычисления.

Логическое ИЛИ

Логический оператор ИЛИ (`||`) используется в ECMAScript следующим образом:

```
var result = true || false;
```

Он работает согласно следующей таблице истинности:

Операнд 1	Операнд 2	Результат
true	true	true
true	false	true
false	true	true
false	false	false

Если один из операндов не является логическим, логическое ИЛИ не всегда возвращает логическое значение. Вместо этого применяются такие правила:

- Если первый операнд — объект, возвращается он.
- Если первый операнд эквивалентен значению `false`, возвращается второй операнд.
- Если оба операнда — объекты, возвращается первый операнд.
- Если оба операнда — значения `null`, возвращается `null`.
- Если оба операнда — значения `NaN`, возвращается `NaN`.
- Если оба операнда — значения `undefined`, возвращается `undefined`.

Как и логическое И, логическое ИЛИ поддерживает сокращенные вычисления. В этом случае второй операнд не оценивается, если первый эквивалентен значению `true`, например:

Листинг LogicalOrExample01.htm


```
var found = true;
var result = (found || someUndeclaredVariable);    // ошибки нет
alert(result);    // все работает
```

Как и в предыдущем примере, переменная `someUndeclaredVariable` не определена, но благодаря тому, что переменная `found` равна `true`, значение `someUndeclaredVariable` никогда не оценивается и код выводит `"true"`. Если значение `found` изменить на `false`, возникнет ошибка:

Листинг LogicalOrExample02.htm

```
var found = false;
var result = (found || someUndeclaredVariable);    // ошибка
alert(result);    // эта строка никогда не выполняется
```

Используя эту схему, можно предотвратить присваивание переменной значения `null` или `undefined`, например:

```
var myObject = preferredObject || backupObject;
```

Здесь переменной `myObject` присваивается одно из двух значений. Переменная `preferredObject` содержит предпочтительное значение, но на тот случай, если оно окажется недоступным, предоставляется также резервная переменная `backupObject`. Если значение `preferredObject` не равно `null`, оно присваивается переменной `myObject`, но если оно равно `null`, переменная получает значение `backupObject`. Этот прием очень часто используется в ECMAScript, и вы еще не раз встретите его в книге.

Мультипликативные операторы

В ECMAScript к группе мультипликативных относятся три оператора: умножение, деление и деление по модулю. Они во многом похожи на свои аналоги в таких

языках, как Java, C и Perl, но автоматически преобразуют некоторые нечисловые значения. Если какой-либо из операндов этих операторов не является числом, он за кулисами преобразуется в число с помощью функции приведения типов `Number()`. Это означает, например, что пустая строка интерпретируется как 0, а логическое значение `true` — как 1.

Умножение

Как нетрудно догадаться, оператор умножения (`*`) умножает два числа. Его синтаксис такой же, как в C:

```
var result = 34 * 56;
```

Умножение специальных значений приводит к ряду уникальных режимов работы:

- Если множители — числа, выполняется обычное арифметическое умножение, при этом умножение двух положительных или двух отрицательных значений дает положительный результат, а умножение операндов с разными знаками — отрицательный. Если результат не может быть представлен в ECMAScript, возвращается значение `Infinity` или `-Infinity`.
- Если какой-либо из множителей — значение `NaN`, в результате получается `NaN`.
- Если `Infinity` умножается на 0, получается `NaN`.
- Если `Infinity` умножается на любое конечное число, отличное от 0, в результате получается или `Infinity`, или `-Infinity` в зависимости от знака второго множителя.
- Если `Infinity` умножается на `Infinity`, получается `Infinity`.
- Если какой-либо из множителей не является числом, он преобразуется в число с помощью функции `Number()`, затем применяются другие правила.

Деление

Оператор деления (`/`) делит первый операнд на второй:

```
var result = 66 / 11;
```

Как и умножение, деление специальных значений имеет ряд особенностей:

- Если операнды — числа, выполняется обычное арифметическое деление, при этом в случае одинаковых знаков операндов получается положительный результат, а в случае разных — отрицательный. Если результат не может быть представлен в ECMAScript, возвращается значение `Infinity` или `-Infinity`.
- Если какой-либо из операндов — значение `NaN`, в результате получается `NaN`.
- Если `Infinity` делится на `Infinity`, получается `NaN`.
- Если 0 делится на 0, получается `NaN`.

- Если ненулевое конечное число делится на 0, в результате получается `Infinity` или `-Infinity`, в зависимости от знака первого операнда.
- Если `Infinity` делится на любое число, в результате получается `Infinity` или `-Infinity`, в зависимости от знака второго операнда.
- Если какой-либо из операндов не является числом, он преобразуется в число с помощью функции `Number()`, затем применяются другие правила.

Деление по модулю

Оператор деления по модулю, или взятия остатка, (`%`) используется следующим образом:

```
var result = 26 % 5;    // 1
```

Он также имеет особенности, когда используется со специальными значениями:

- Если операнды — числа, выполняется обычное арифметическое деление и возвращается остаток.
- Если бесконечное число делится на конечное, в результате получается `NaN`.
- Если конечное число делится на 0, получается `NaN`.
- Если `Infinity` делится на `Infinity`, получается `NaN`.
- Если конечное число делится на бесконечное, возвращается делимое.
- Если 0 делится на число, не равное 0, в результате получается 0.
- Если какой-либо из операндов не является числом, он преобразуется в число с помощью функции `Number()`, затем применяются другие правила.

Операторы сложения и вычитания

Сложение и вычитание — одни из простейших математических операций в языках программирования, но в ECMAScript у них есть много нюансов. При сложении и вычитании также выполняется неявное преобразование типов данных, но его правила не так просты, как для умножения и деления.

Сложение

Оператор сложения (`+`) используется обычным образом:

```
var result = 1 + 2;
```

Если оба слагаемых являются числами, он выполняет арифметическое сложение и возвращает результат по указанным правилам:

- Если какое-либо из слагаемых — значение `NaN`, в результате получается `NaN`.

- Если суммируются значения `Infinity` и `Infinity`, получается `Infinity`.
- Если суммируются `-Infinity` и `-Infinity`, получается `-Infinity`.
- Если суммируются `Infinity` и `-Infinity`, получается `NaN`.
- Если суммируются +0 и +0, получается +0.
- Если суммируются –0 и +0, получается +0.
- Если суммируются –0 и –0, получается –0.

Если один из операндов является строкой, применяются другие правила:

- Если оба операнда — строки, вторая строка присоединяется к первой.
- Если только один операнд — строка, другой операнд преобразуется в строку и выполняется конкатенация строк.

Если какой-либо из операндов является объектом, числом или логическим значением, вызывается его метод `toString()` для получения строкового значения, а затем применяются правила для строк. Для значений `undefined` и `null` вызывается функция `String()`, которая возвращает `"undefined"` и `"null"` соответственно.

Рассмотрим следующий пример:

Листинг AddExample01.htm


```
var result1 = 5 + 5;      // два числа
alert(result1);           // 10
var result2 = 5 + "5";    // число и строка
alert(result2);           // "55"
```

Этот код поясняет различие между двумя режимами оператора сложения. В обычной ситуации 5 + 5 равно 10 (числовое значение), что подтверждают первые две строки кода. Однако если изменить один из операндов на строку `"5"`, результат изменяется на `"55"` (примитивное строковое значение), потому что другой операнд также преобразуется в строку `"5"`.

Невнимание к типам слагаемых часто приводит в ECMAScript к ошибкам, например:

Листинг AddExample02.htm

```
var num1 = 5;
var num2 = 10;
var message = "The sum of 5 and 10 is " + num1 + num2;
alert(message);  // выводится сообщение о том, что сумма 5 и 10 равна 510
```

В этом примере переменной `message` присваивается строка, включающая результат двух операций сложения. Предполагается, что ею должна быть строка `"The sum of 5 and 10 is 15"` (сумма 5 и 10 равна 15), но на экран выводится сообщение `"The sum of 5 and 10 is 510"` (сумма 5 и 10 равна 510). Это происходит из-за того, что каждое сложение выполняется отдельно. В первый раз строка складывается с числом 5, что дает в результате строку. Далее к ней добавляется число 10, и опять получается

строка. Чтобы сложить числа и присоединить результат к строке, просто добавьте скобки:

Листинг AddExample03.htm


```
var num1 = 5;
var num2 = 10;
var message = "The sum of 5 and 10 is " + (num1 + num2);
alert(message);    // Сумма 5 и 10 равна 15
```

Теперь интерпретатор сначала вычисляет сумму числовых переменных в скобках, а затем присоединяет ее к строке. Итоговая строка: `"The sum of 5 and 10 is 15"` (сумма 5 и 10 равна 15).

Вычитание

Оператор вычитания (`-`) также используется весьма часто. Вот пример с ним:

```
var result = 2 - 1;
```

Как и при сложении, при вычитании в ECMAScript действуют специальные правила преобразования типов:

- Если оба операнда — числа, выполняется арифметическое вычитание и возвращается результат.
- Если какой-либо из операндов — `NaN`, в результате получается `NaN`.
- Если `Infinity` вычитается из `Infinity`, получается `NaN`.
- Если `-Infinity` вычитается из `-Infinity`, получается `NaN`.
- Если `-Infinity` вычитается из `Infinity`, получается `Infinity`.
- Если `Infinity` вычитается из `-Infinity`, получается `-Infinity`.
- Если +0 вычитается из +0, получается +0.
- Если –0 вычитается из +0, получается –0.
- Если –0 вычитается из –0, получается +0.
- Если какой-либо из операндов — строка, логическое значение, `null` или `undefined`, он преобразуется в число с помощью функции `Number()`, а затем выполняется арифметическое вычитание по описанным правилам. Если операнд преобразуется в `NaN`, результат вычитания — `NaN`.
- Если какой-либо из операндов является объектом, вызывается его метод `valueOf()` для получения числового значения. Если это значение `NaN`, результат вычитания — `NaN`. Если для объекта не определен метод `valueOf()`, вызывается метод `toString()`, а полученная строка преобразуется в число.

Вот несколько примеров использования этих правил.

Листинг SubtractExample01.htm


```
var result1 = 5 - true;    // 4, потому что true преобразуется в 1
var result2 = NaN - 1;     // NaN
var result3 = 5 - 3;       // 2
var result4 = 5 - "";      // 5, потому что "" преобразуется в 0
var result5 = 5 - "2";     // 3, потому что "2" преобразуется в 2
var result6 = 5 - null;    // 5, потому что null преобразуется в 0
```

Операторы отношений

Операторы отношений «меньше» (<), «больше» (>), «меньше или равно» (<=) и «больше или равно» (>=) сравнивают значения так же, как в школьной математике. Каждый из них возвращает логическое значение, например:

```
var result1 = 5 > 3;    // true
var result2 = 5 < 3;    // false
```

Все примеры из этого раздела доступны в файле RelationalOperatorsExample01.htm на сайте Wrox.com.

Как и в других ECMAScript-операциях, при сравнении некоторые типы данных преобразуются по особым правилам:

- Если операнды — числа, выполняется числовое сравнение.
- Если операнды — строки, сравниваются коды знаков в одинаковых позициях.
- Если один операнд — число, другой операнд преобразуется в число и выполняется числовое сравнение.
- Если операнд — объект, вызывается метод `valueOf()` и его результат сравнивается с другим операндом по предыдущим правилам. Если метод `valueOf()` недоступен, вызывается метод `toString()` и полученное значение сравнивается по предыдущим правилам.
- Если операнд — логическое значение, он преобразуется в число и выполняется сравнение.

Если оператор отношения применяется к двум строкам, происходит кое-что интересное. Обычно предполагают, что «меньше» означает «ближе к началу алфавита», а «больше» — «ближе к концу алфавита», но это не так. Для строк каждый числовой код знака первой строки сравнивается с соответствующим кодом знака второй строки, после чего возвращается логическое значение. Проблема в том, что коды прописных букв меньше, чем строчных, из-за чего получается следующее:

```
var result = "Brick" < "alphabet";    // true
```

В этом примере строка `"Brick"` считается меньше, чем строка `"alphabet"`, потому что буква *B* имеет код 66, а буква *a* — 97. Чтобы сравнить строки по алфавиту,

необходимо перед сравнением преобразовать оба операнда в один регистр (не важно, верхний или нижний):

```
var result = "Brick".toLowerCase() < "alphabet".toLowerCase();  // false
```

После преобразования в нижний регистр слово `"alphabet"` правильно распознается как находящееся ближе к началу алфавита, чем `"Brick"`.

Другая непростая ситуация возникает при сравнении чисел в строковой форме, например:

```
var result = "23" < "3";  // true
```

Здесь утверждается, что `"23"` меньше, чем `"3"`. Почему? Потому что строковые операнды сравниваются по кодам знаков (код знака `"2"` — 50; код знака `"3"` — 51). Если один из операндов изменить на число, результат больше не удивляет:

```
var result = "23" < 3;    // false
```

В этом примере строка `"23"` перед сравнением преобразуется в число 23, что обеспечивает правильный результат. Когда строка сравнивается с числом, она преобразуется в число и выполняется числовое сравнение. Это хорошо работает в ситуациях вроде предыдущей, но что, если строку невозможно преобразовать в число? Взгляните на следующий пример:

```
var result = "a" < 3;    // false, потому что "a" преобразуется в NaN
```

Буква `"a"` не может быть осмысленно преобразована в число, поэтому она становится значением `NaN`. Как правило, в результате любой операции отношения с операндом `NaN` получается `false`, что имеет интересные следствия:

```
var result1 = NaN < 3;    // false
var result2 = NaN >=3;    // false
```

Обычно считается, что если одно значение не меньше другого, то оно должно быть больше или равно ему, но для значения `NaN` оба сравнения возвращают `false`.

Операторы эквивалентности

Определение эквивалентности двух переменных — одна из самых важных операций в программировании. Это довольно просто при работе со строками, числами и логическими значениями, но задача усложняется, когда дело доходит до объектов. Первоначально операторы равенства и неравенства в ECMAScript преобразовывали операнды перед сравнением в похожие типы, но затем был поставлен вопрос, а правильно ли это. В итоге в ECMAScript определили два набора операций: *равенство* (equal) и *неравенство* (not equal), которые преобразуют данные перед сравнением,

и *строгое равенство* (identically equal) и *строгое неравенство* (not identically equal), которые выполняют сравнение без преобразования.

Равенство и неравенство

Оператор равенства (`==`) возвращает `true`, если операнды равны. Оператор неравенства (`!=`) возвращает `true`, если операнды не равны. Для определения равенства операндов оба оператора при необходимости преобразуют их типы, что часто называют *приведением типов* (type coercion).

При преобразовании типов для операторов равенства и неравенства применяются свои правила:

- Если операнд — логическое значение, перед проверкой на равенство оно преобразуется в число. Значение `false` преобразуется в 0, а `true` — в 1.
- Если операнды — строка и число, перед сравнением предпринимается попытка преобразовать строку в число.
- Если один из операндов — объект, для него вызывается метод `valueOf()`, чтобы получить примитивное значение, которое затем сравнивается по предыдущим правилам.

При сравнении применяются свои правила:

- Значения `null` и `undefined` равны.
- Значения `null` и `undefined` не преобразуются для сравнения ни в какие другие значения.
- Если одним из операндов является значение `NaN`, оператор равенства возвращает `false`, а оператор неравенства — `true`. Даже если оба операнда — значения `NaN`, оператор равенства возвращает `false`, потому что по правилам `NaN` не равно `NaN`.
- Если оба операнда — объекты, они сравниваются, чтобы выяснить, один ли это объект. Если да, возвращается `true`, иначе — `false`.

Некоторые специальные сравнения и результаты приведены в таблице (см. файл EqualityOperatorsExample01.htm).

Выражение	Значение
null == undefined	true
"NaN" == NaN	false
5 == NaN	false
NaN == NaN	false
NaN != NaN	true
false == 0	true

Выражение	Значение
true == 1	true
true == 2	false
undefined == 0	false
null == 0	false
"5" == 5	true

Строгое равенство и строгое неравенство

Операторы строгих равенства и неравенства делают то же самое, что и обычные операторы равенства и неравенства, но не преобразуют операнды перед сравнением. Оператор строго равенства (`===`) возвращает `true`, только если операнды равны без преобразования, например:

Листинг EqualityOperatorsExample02.htm

```
var result1 = ("55" == 55);    // true - равно благодаря преобразованию
var result2 = ("55" === 55);   // false - не равно из-за разных типов данных
```

В первом случае здесь с помощью оператора равенства сравниваются строка `"55"` и число 55, что дает в результате `true`. Как уже отмечалось, это происходит потому, что строка `"55"` преобразуется в число 55, которое затем сравнивается с другим числом 55. Во втором случае строка и число сравниваются без преобразования, и конечно, возвращается `false`, потому что строка не равна числу.

Оператор строгого неравенства (`!==`) возвращает `true`, если без преобразования операнды не равны, например:

Листинг EqualityOperatorsExample03.htm

```
var result1 = ("55" != 55);    // false - равно благодаря преобразованию
var result2 = ("55" !== 55);   // true - не равно из-за разных типов данных
```

Здесь в первом сравнении используется оператор равенства, который преобразует строку `"55"` в число 55. Оно равно второму операнду, поэтому выражение получает значение `false`. Во втором сравнении используется оператор строгого неравенства, который возвращает `true`, потому что строка `"55"` отличается от числа 55.

Запомните, что выражение `null == undefined` истинно, потому что значения равны, но `null === undefined` ложно, потому что типы этих значений разные.

Из-за проблем преобразования типов, возникающих при работе с операторами равенства и неравенства, рекомендуется использовать вместо них операторы строгих равенства и неравенства. Это помогает поддерживать целостность типов данных в коде.

Условный оператор

Условный оператор используется в ECMAScript в самых разных ситуациях и работает так же, как и в Java:

```
переменная = логическое_выражение ? значение_если_true : значение_если_false
```

С помощью этого оператора можно присваивать переменной разные значения в зависимости от логического выражения. Если оно истинно, переменной присваивается первое значение, иначе — второе:

```
var max = (num1 > num2) ? num1 : num2
```

В этом примере переменной `max` присваивается большее число. Если `num1` больше, чем `num2`, то переменная `max` получает значение `num1`. Если же выражение в скобках ложно (то есть `num1` меньше или равно `num2`), она становится равной `num2`.

Операторы присваивания

Простое присваивание выполняется с помощью знака равенства (`=`), при этом значение справа от него просто присваивается переменной слева, например:

```
var num = 10;
```

Составное присваивание выполняется с помощью одного из арифметических операторов или операторов сдвига, за которым следует знак равенства. Это сокращает код в некоторых популярных сценариях, например:

```
var num = 10;
num = num + 10;
```

Этот код эквивалентен следующему:

```
var num = 10;
num += 10;
```

Составные операторы присваивания есть для всех основных математических операций и для нескольких других. Вот они:

- умножение с присваиванием (`*=`);
- деление с присваиванием (`/=`);
- деление по модулю с присваиванием (`%=`);
- сложение с присваиванием (`+=`);
- вычитание с присваиванием (`-=`);
- сдвиг влево с присваиванием (`<<=`);

- сдвиг вправо с сохранением знака и присваиванием (>>=);
- сдвиг вправо с заполнением нулями и присваиванием (>>>=).

Эти операторы только сокращают объем кода, но не увеличивают его быстродействие.

Оператор «запятая»

Оператор «запятая» позволяет выполнить в одной инструкции более одной операции:

```
var num1 = 1, num2 = 2, num3 = 3;
```

Чаще всего он используется в объявлениях переменных, но с его помощью можно также присваивать значения. В этом случае он всегда возвращает последний элемент выражения, например:

```
var num = {5, 1, 4, 8, 0};  // num равно 0
```

В этом примере переменная `num` получает значение 0, потому что это последний элемент выражения. Так запятые используются редко, однако полезно знать, что это возможно.

Инструкции

ECMA-262 включает несколько инструкций, называемых также *управляющими инструкциями* (flow-control statements), которые составляют основную часть синтаксиса ECMAScript и обычно решают специфическую задачу с помощью одного или нескольких ключевых слов. Сложность инструкций варьируется в широких пределах: от тривиального выхода из функции до блоков многократно выполняемых команд.

Инструкция if

Инструкция `if` часто используется почти во всех языках программирования. Она имеет следующий синтаксис:

```
if (условие) инструкция1 else инструкция2
```

Условие может быть любым выражением. Оно даже может не относиться к логическому типу, потому что ECMAScript автоматически преобразует результат выражения в логическое значение, вызывая для него функцию `Boolean()`. Если условие эквивалентно `true`, выполняется инструкция 1, в противном случае — инструкция 2. Любая из инструкций может быть одной строкой или блоком кода (группой строк в фигурных скобках), например:

Листинг IfStatementExample01.htm


```
if (i > 25)
    alert("Greater than 25.");              // однострочная инструкция
else {
    alert("Less than or equal to 25.");     // блочная инструкция
}
```

Рекомендуется всегда использовать блочные инструкции, даже если нужно выполнить всего одну строку кода. Это ясно показывает, что должно быть выполнено в каждом случае.

Инструкции `if` можно сцеплять друг с другом:

```
if (условие1) инструкция1 else if (условие2) инструкция2 else инструкция3
```

Вот пример:

Листинг IfStatmentExample02.htm

```
if (i > 25) {
    alert("Greater than 25.");
} else if (i < 0) {
    alert("Less than 0.");
} else {
    alert("Between 0 and 25, inclusive.");
}
```

Инструкция do-while

Инструкция `do-while` создает цикл с постусловием, в котором условие выхода из цикла проверяется только после выполнения кода внутри него. Тело цикла выполняется как минимум один раз перед оценкой выражения. Вот синтаксис цикла:

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

А вот пример его использования:

Листинг DoWhileStatementExample01.htm


```
var i = 0;
do {
   i +=2;
} while (i < 10);
```

Этот цикл продолжается, пока переменная `i` меньше 10. Она равна 0 в начале цикла и увеличивается на 2 на каждой итерации.

Циклы с постусловием чаще всего используются, если тело цикла должно быть выполнено хотя бы один раз.

Инструкция while

Инструкция `while` создает цикл с предусловием. Это означает, что условие выхода из цикла проверяется перед выполнением кода внутри него. Возможно, что тело цикла не будет выполнено ни разу. Синтаксис этого цикла таков:

```
while(выражение) инструкция
```

А вот пример его использования:

Листинг WhileStatementExample01.htm

```
var i = 0;
while (i < 10) {
   i += 2;
}
```

Переменная `i` равна 0 перед началом цикла и увеличивается на 2 на каждой итерации. Пока она меньше 10, цикл продолжается.

Инструкция for

Инструкция `for` — это вариант цикла с предусловием, позволяющий инициализировать переменную перед началом цикла и указать код, выполняемый после цикла. Она имеет следующий синтаксис:

```
for (инициализация; выражение; выражение после цикла) инструкция
```

Пример цикла `for`:

Листинг ForStatementExample01.htm


```
var count = 10;
for (var i=0; i < count; i++){
    alert(i);
}
```

В этом фрагменте определяется переменная `i` с нулевым значением. Цикл `for` начинается, только если результатом условного выражения (`i < count`) является значение `true`, то есть тело цикла может быть не выполнено ни разу. Если тело цикла выполняется, вслед за ним в выражении после цикла увеличивается значение `i`. Этот цикл `for` эквивалентен следующему:

```
var count = 10;
var i = 0;
while (i < count){
    alert(i);
    i++;
}
```

С помощью инструкции `for` нельзя сделать ничего такого, что невозможно было бы выполнить в цикле `while`, она просто группирует связанный с циклом код в одном месте.

Использовать ключевое слово `var` в разделе инициализации цикла `for` не требуется. Это можно сделать и вне цикла:

Листинг ForStatementExample02.htm

```
var count = 10;
var i;
for (i=0; i < count; i++){
    alert(i);
}
```

Этот код эквивалентен объявлению переменной в разделе инициализации цикла. В ECMAScript нет переменных с блочной областью видимости (см. главу 4), так что переменная, определенная внутри цикла, доступна и вне его, например:

Листинг ForStatementExample03.htm

```
var count = 10;
for (var i=0; i < count; i++){
    alert(i);
}
alert(i);    // 10
```

В этом примере в окне оповещения выводится 10, то есть окончательное значение `i` после завершения цикла. Это возможно потому, что переменная `i` все еще доступна, хотя и была определена внутри цикла.

Инициализация, управляющее выражение и выражение после цикла не обязательны. Если опустить все три части, получится бесконечный цикл:

```
for (;;) {      // бесконечный цикл
    doSomething();
}
```

Добавив только управляющее выражение, можно преобразовать цикл `for` в цикл `while`:

Листинг ForStatementExample04.htm


```
var count = 10;
var i = 0;
for (; i < count; ){
    alert(i);
    i++;
}
```

Благодаря такой гибкости инструкция `for` в ECMAScript является одной из наиболее востребованных.

Инструкция for-in

Инструкция `for-in` используется для перебора свойств объектов и имеет следующий синтаксис:

```
for (свойство in выражение) инструкция
```

Пример с инструкцией `for-in`:

Листинг ForInStatementExample01.htm

```
for (var propName in window) {
     document.write(propName);
}
```

Здесь инструкция `for-in` используется для вывода из объектной модели браузера на экран всех свойств объекта `windows`. При каждой итерации цикла переменной `propName` присваивается имя очередного свойства. Это продолжается, пока не будут перебраны все доступные свойства. Как и в цикле `for`, оператор `var` в управляющем выражении здесь не обязателен, но рекомендуется использовать его, чтобы переменная была локальной.

Свойства объектов в ECMAScript не упорядочены, поэтому порядок возврата их имен в цикле `for-in` предсказать нельзя. Все перечислимые свойства будут возвращены, но порядок их вывода может зависеть от браузера.

Если переменная, представляющая перебираемый объект, равна `null` или `undefined`, инструкция `for-in` генерирует ошибку, а в ECMAScript 5 вместо этого тело цикла просто не выполняется. Для обеспечения совместимости рекомендуется проверять значение объекта перед циклом `for-in` на предмет того, не равно ли оно `null` или `undefined`.

В Safari до версии 3 инструкция for-in могла по ошибке возвращать некоторые свойства дважды.

Метки инструкций

Инструкции можно помечать, чтобы затем ссылаться на них. Синтаксис меток таков:

```
метка: инструкция
```

Вот пример кода с меткой:

```
start: for (var i=0; i < count; i++) {
    alert(i);
}
```

В этом примере на метку `start` можно ссылаться позднее в инструкции `break` или `continue`. Помеченные инструкции обычно используются с вложенными циклами.

Инструкции break и continue

Инструкции `break` и `continue` обеспечивают более точный контроль над выполнением кода в цикле. Инструкция `break` немедленно завершает цикл, передавая управление следующей инструкции после цикла, а `continue` завершает только текущую итерацию цикла, начиная новую. Рассмотрим пример:

Листинг BreakStatementExample01.htm


```
var num = 0;

for (var i=1; i < 10; i++) {
    if (i % 5 == 0) {
        break;
    }
    num++;
}

alert(num);    // 4
```

В цикле `for` переменная `i` увеличивается с 1 до 10. В теле цикла инструкция `if` с помощью оператора деления по модулю проверяет, делится ли значение `i` без остатка на 5. Если да, выполняется инструкция `break` и цикл завершается. Переменная `num` подсчитывает количество итераций цикла. После `break` выводится оповещение со значением 4. Цикл выполняется 4 раза потому, что когда `i` равно 5, инструкция `break` завершает цикл до очередного увеличения значения `num`. Если изменить `break` на `continue`, получится другой результат:

Листинг ContinueStatementExample01.htm


```
var num = 0;

for (var i=1; i < 10; i++) {
    if (i % 5 == 0) {
        continue;
    }
    num++;
}

alert(num);    // 8
```

В этот раз итоговое количество итераций равно 8. Когда `i` достигает значения 5, итерация цикла завершается до увеличения переменной `num`, но цикл продолжается со следующей итерации со значением `i`, равным 6. Затем цикл выполняется до естественного завершения при значении `i`, равном 10. Окончательное значение `num` равно 8, а не 9, потому что одна операция инкремента пропускается из-за инструкции `continue`.

И `break`, и `continue` можно использовать вместе с помеченными инструкциями для возврата к конкретному месту в коде. Обычно это делается во вложенных циклах, например:

Листинг BreakStatementExample02.htm

```
var num = 0;

outermost:
for (var i=0; i < 10; i++) {
    for (var j=0; j < 10; j++) {
        if (i == 5 && j == 5) {
            break outermost;
        }
        num++;
    }
}

alert(num);    // 55
```

В этом примере для первой инструкции `for` добавлена метка `outermost`. Каждый цикл включает 10 итераций, то есть инструкция `num++` предположительно должна быть выполнена 100 раз, после чего переменная `num` должна быть равна 100. Инструкция `break` получает здесь в качестве аргумента метку для перехода, вследствие чего она завершает не только внутренний цикл `for` (с переменной `j`), но и внешний (с переменной `i`). Окончательное значение `num` равно 55, потому что циклы завершаются, когда `i` и `j` равны 5. Инструкция `continue` используется аналогично:

Листинг ContinueStatementExample02.htm


```
var num = 0;

outermost:
for (var i=0; i < 10; i++) {
    for (var j=0; j < 10; j++) {
        if (i == 5 && j == 5) {
            continue outermost;
        }
        num++;
    }
}

alert(num);    //95
```

В этом случае инструкция `continue` завершает выполнение внутреннего цикла, начиная новую итерацию внешнего. Она выполняется, когда `j` равно 5, то есть пропускаются пять итераций внутреннего цикла, из-за чего `num` в итоге имеет значение 95.

Использование помеченных инструкций вместе с `break` и `continue` — очень эффективный прием, но не злоупотребляйте им, иначе будет трудно отлаживать код. Всегда назначайте меткам описательные имена и не создавайте циклов с большим количеством уровней вложенности.

Инструкция with

Инструкция `with` делает областью видимости кода конкретный объект. Вот ее синтаксис:

```
with (выражение) инструкция;
```

Инструкция `with` была создана ради удобства для тех случаев, когда имя одного объекта приходится вводить снова и снова, например:

```
var qs = location.search.substring(1);
var hostName = location.hostname;
var url = location.href;
```

Чтобы не указывать в каждой строке объект `location`, этот код можно переписать следующим образом:

Листинг WithStatementExample01.htm

```
with(location){
    var qs = search.substring(1);
    var hostName = hostname;
    var url = href;
}
```

В этой версии кода, где инструкция `with` используется с объектом `location`, каждая переменная внутри блока сначала считается локальной. Если выясняется, что она не является локальной, выполняется поиск свойства с тем же именем в объекте `location`. Если оно обнаруживается, переменная интерпретируется как свойство объекта `location`.

В строгом режиме инструкция `with` не поддерживается. Попытка использовать ее приведет к синтаксической ошибке.

Использование инструкции with в окончательном коде считается плохой практикой, потому что это снижает быстродействие и затрудняет отладку кода.

Инструкция switch

С `if` тесно связана управляющая инструкция `switch`, также заимствованная из других языков. Синтаксис `switch` в ECMAScript напоминает аналоги в других C-подобных языках программирования:

```
switch (выражение) {
  case значение: инструкция
    break;
  case значение: инструкция
    break;
  case значение: инструкция
    break;
  case значение: инструкция
    break;
  default: инструкция
}
```

Если выражение равно конкретному значению в блоке `switch`, выполняется соответствующая инструкция. Ключевое слово `break` вызывает выход из блока `switch`, в противном случае просто выполнялась бы следующая инструкция в списке. Ключевое слово `default` указывает код, который выполняется, если выражение не равно ни одному значению (по сути, оно аналогично `else`).

Инструкция `switch` избавляет от необходимости писать код вроде этого:

```
if (i == 25){
  alert("25");
} else if (i == 35) {
  alert("35");
} else if (i == 45) {
  alert("45");
} else {
  alert("Other");
}
```

Этот фрагмент эквивалентен следующему:

Листинг SwitchStatementExample01.htm


```
switch (i) {
    case 25:
        alert("25");
        break;
    case 35:
        alert("35");
        break;
    case 45:
        alert("45");
        break;
    default:
        alert("Other");
}
```

Лучше всегда добавлять слово `break` в конце каждого раздела `case`, чтобы управление не «проваливалось» в следующий раздел. Если же именно это и нужно, добавьте комментарий, поясняющий, что инструкция `break` опущена умышленно:

Листинг SwitchStatementExample02.htm

```
switch (i) {
    case 25:
         /* переход к следующему разделу */
    case 35:
        alert("25 или 35");
        break;
    case 45:
        alert("45");
        break;
    default:
        alert("Other");
}
```

Хотя инструкция switch позаимствована из других языков, в ECMAScript она имеет некоторые особенности. Во-первых, она работает со всеми типами данных (во многих языках только с числами), так что ее можно использовать со строками и даже с объектами. Во-вторых, значения для сравнения с выражением могут быть не только константами, но и переменными и даже выражениями. Рассмотрим пример:

Листинг SwitchStatementExample03.htm

```
switch ("hello world!") {
    case "hello" + " world!":
        alert("Greeting was found.");
        break;
    case "goodbye":
        alert("Closing was found.");
        break;
    default:
        alert("Unexpected message was found.");
}
```

В этом примере в инструкции switch используется строковое значение, которое в первом же разделе сравнивается с выражением — результатом конкатенации строк. Поскольку объединенная строка равна аргументу switch, выводится сообщение "Greeting was found." (это приветствие). Возможность использовать выражения как селекторы разделов case позволяет писать такой код:

Листинг SwitchStatementExample04.htm


```
var num = 25;
switch (true) {
    case num < 0:
        alert("Less than 0.");
        break;
    case num >=0 && num <= 10:
        alert("Between 0 and 10.");
        break;
    case num > 10 && num <= 20:
        alert("Between 10 and 20.");
        break;
    default:
        alert("More than 20.");
}
```

Здесь переменная num определена вне инструкции switch. В switch передается значение true, которое сравнивается с логическими значениями — результатами вычисления условий case. Каждое значение оценивается по порядку, пока не будет обнаружено совпадение или пока не встретится инструкция default (как в этом примере).

В инструкции switch значения сравниваются с помощью оператора строгого равенства, то есть без преобразования типов (например, строка "10" не равна числу 10).

Функции

Функции — это базовые элементы любого языка, с помощью которых можно инкапсулировать группы инструкций и в любое время выполнять их в других местах кода. В ECMAScript для объявления функции используется ключевое слово `function`, за которым следуют аргументы и тело функции:

```
function имяФункции(арг0, арг1, ..., аргN) {
    инструкции
}
```

Вот пример функции:

Листинг FunctionExample01.htm

```
function sayHi(name, message) {
    alert("Hello " + name + ", " + message);
}
```

После определения функции ее можно вызывать по имени, указывая в скобках после него аргументы, разделенные запятыми. Например, функция `sayHi()` вызывается так:

```
sayHi("Nicholas", "how are you today?");
```

В результате в окне оповещения будет выведена строка `"Hello Nicholas, how are you today?"` (Здравствуй, Николас, как дела?), составленная из приветствия и аргументов `name` и `message`.

Для функций в ECMAScript можно не указывать, возвращают ли они значения. Из любой функции можно возвратить значение в любое время с помощью инструкции `return`, например:

Листинг FunctionExample02.htm


```
function sum(num1, num2) {
    return num1 + num2;
}
```

Эта функция суммирует два значения и возвращает результат. Обратите внимание, что кроме инструкции `return` нет никакого специального объявления о том, что функция возвращает значение. Эту функцию можно вызвать следующим образом:

```
var result = sum(5, 10);
```

Помните, что при достижении инструкции `return` функция сразу завершается. Никакой код после `return` никогда не выполняется, например:

```
function sum(num1, num2) {
    return num1 + num2;
    alert("Hello world!");  // никогда не выполняется
}
```

В этом примере оповещение никогда не появится, потому что вызов `alert` расположен после `return`.

Функция может содержать несколько инструкций `return`:

Листинг FunctionExample03.htm

```
function diff(num1, num2) {
    if (num1 < num2) {
        return num2 - num1;
    } else {
        return num1 - num2;
    }
}
```

Эта функция находит разность двух чисел. Если первое число меньше, оно вычитается из второго, в противном случае второе число вычитается из первого. Каждая ветвь кода содержит отдельную инструкцию `return` с правильным выражением.

Инструкцию `return` можно также использовать без указания возвращаемого значения. В этом случае функция немедленно завершается, возвращая значение `undefined`. Это обычно делается для преждевременного завершения функций, которые не возвращают значение. В следующем примере оповещение не выводится:

Листинг FunctionExample04.htm


```
function sayHi(name, message) {
    return;
    alert("Hello " + name + ", " + message);  // никогда не вызывается
}
```

Рекомендуется либо возвращать значение из функции всегда, либо не возвращать никогда. Функцию, которая возвращает значение иногда, труднее понять, особенно на этапе отладки.

В строгом режиме на функции налагаются следующие ограничения:

- функции не могут называться `eval` или `arguments`;
- именованные параметры не могут называться `eval` или `arguments`;
- именованные параметры не могут иметь одинаковые имена.

Если какое-либо из этих условий нарушено, возникает синтаксическая ошибка и код не выполняется.

Аргументы функций

Аргументы функций в ECMAScript работают не так, как в большинстве других языков. В ECMAScript функции все равно, сколько аргументов в нее передается и каковы их типы. То, что вы определили функцию с двумя аргументами, не означает,

что ей можно передать только два аргумента. Можно передать ей один аргумент, три аргумента или не передать ни одного, интерпретатор жаловаться не будет. Это объясняется тем, что аргументы в ECMAScript внутренне представляются как массив. В функцию всегда передается массив, и ее не интересует, что в нем содержится (и содержится ли вообще). Если в нем нет элементов, функцию все устраивает; если элементов больше, чем нужно, она тоже довольна. Получить значения каждого переданного аргумента внутри функции позволяет объект `arguments`.

Объект `arguments` напоминает массив, хотя и не является экземпляром типа `Array`. Так, для доступа к его элементам можно использовать квадратные скобки, при этом первым аргументом является `arguments[0]`, вторым — `arguments[1]`, и т. д. Определить, как много аргументов было передано в функцию, можно с помощью свойства `length`. В предыдущем примере первый аргумент функции `sayHi()` называется `name`, но то же самое значение доступно как `arguments[0]`. Таким образом, функцию можно переписать, не называя аргументы явно:

Листинг FunctionExample05.htm


```
function sayHi() {
    alert("Hello " + arguments[0] + ", " + arguments[1]);
}
```

В этой версии функции нет именованных аргументов, но несмотря на то, что аргументы `name` и `message` были удалены, функция работает как прежде. Это иллюстрирует важную особенность функций в ECMAScript: именованные аргументы в них — это удобство, а не необходимость. В отличие от других языков, указание именованных аргументов в ECMAScript не определяет сигнатуру функции, которую позднее нужно соблюдать, — с ними ничто не сопоставляется.

С помощью свойства `length` объекта `arguments` можно узнать, сколько аргументов передано в функцию. Следующий код выводит количество аргументов, переданных в функцию при каждом вызове:

Листинг FunctionExample06.htm

```
function howManyArgs() {
    alert(arguments.length);
}

howManyArgs("string", 45);  // 2
howManyArgs();              // 0
howManyArgs(12);            // 1
```

В этом примере выводится оповещение с числами 2, 0 и 1 (именно в таком порядке). Так можно обрабатывать в функции любое количество параметров. Рассмотрим пример:

Листинг FunctionExample07.htm

```
function doAdd() {
    if(arguments.length == 1) {
```

```
        alert(arguments[0] + 10);
    } else if (arguments.length == 2) {
        alert(arguments[0] + arguments[1]);
    }
}

doAdd(10);      // 20
doAdd(30, 20);  // 50
```

Функция `doAdd()` прибавляет 10 к числу, только если аргумент один; если аргументов два, они суммируются и выводится их сумма. Таким образом, `doAdd(10)` выводит 20, а `doAdd(30, 20)` — 50. Пусть этот прием не так хорош, как перегрузка функций, но чтобы обойти это ограничение в ECMAScript, его достаточно.

Имейте также в виду, что объект `arguments` можно использовать вместе с именованными аргументами, например:

Листинг FunctionExample08.htm


```
function doAdd(num1, num2) {
    if(arguments.length == 1) {
        alert(num1 + 10);
    } else if (arguments.length == 2) {
        alert(arguments[0] + num2);
    }
}
```

В этой версии функции `doAdd()` два именованных аргумента используются вместе с объектом `arguments`. Аргумент `num1` содержит то же значение, что и `arguments[0]`, так что они взаимозаменяемы (как и пара `num2` и `arguments[1]`).

Объект `arguments` также интересен тем, что его значения всегда синхронизированы с соответствующими именованными параметрами, например:

Листинг FunctionExample09.htm

```
function doAdd(num1, num2) {
    arguments[1] = 10;
    alert(arguments[0] + num2);
}
```

Эта версия `doAdd()` всегда перезаписывает второй аргумент значением 10. Поскольку изменения значений в объекте `arguments` автоматически отражаются на соответствующих именованных аргументах, при изменении `arguments[1]` аргумент `num2` также получает значение 10. Однако это не означает, что они занимают одно место в памяти. Нет, они хранятся по отдельности, но синхронизируются. Этот эффект односторонний: изменение именованного аргумента *не* приводит к обновлению соответствующего значения в массиве `arguments`. Имейте также в виду, что если передать в функцию только один аргумент, присвоение значения элементу `arguments[1]` не отразится на этом аргументе. Это объясняется тем, что длина объекта `arguments` определяется по количеству переданных аргументов, а не именованных аргументов, указанных для функции.

Любой именованный аргумент, для которого в функцию не было передано значение, автоматически получает значение `undefined`. Это похоже на объявление переменной без ее инициализации. Например, если в функцию `doAdd()` передать только один аргумент, `num2` будет иметь значение `undefined`.

В строгом режиме возможности объекта `arguments` другие. Во-первых, присваивание, как в предыдущем примере, больше не работает. Аргумент `num2` сохраняет значение `undefined`, даже когда элементу `arguments[1]` присваивается значение 10. Во-вторых, попытка перезаписать значение `arguments` приводит к синтаксической ошибке (код не выполняется).

Все аргументы в ECMAScript передаются по значению. Передать аргумент по ссылке невозможно.

Никакой перегрузки

ECMAScript-функции нельзя перегружать в традиционном смысле. В других языках, таких как Java, можно создать два определения функции, если их сигнатуры (тип и количество принимаемых аргументов) различны. В ECMAScript функции не имеют сигнатур, потому что их аргументы представлены как массив, а без сигнатур настоящая перегрузка функций невозможна.

Если определить в ECMAScript две функции с одним именем, это имя получит функция, определенная последней, например:

Листинг FunctionExample10.htm


```
function addSomeNumber(num) {
    return num + 100;
}

function addSomeNumber(num) {
    return num + 200;
}

var result = addSomeNumber(100);  // 300
```

Здесь функция `addSomeNumber` определена дважды. В первой ее версии к аргументу прибавляется 100, во второй — 200. Когда функция вызывается, она возвращает 300, потому что вторая версия перезаписала первую.

Как было отмечено ранее, перегрузку методов можно имитировать, проверяя тип и количество переданных в функцию аргументов и выбирая в зависимости от них дальнейшие действия.

Резюме

Фундаментальные возможности JavaScript определены в ECMA-262 как псевдоязык с именем ECMAScript. Он содержит все основные синтаксические элементы,

операторы, типы данных и объекты, необходимые для выполнения базовых вычислительных задач, но не предоставляет средств ввода или вывода данных. Знание ECMAScript и его особенностей важно для полного понимания реализаций JavaScript в веб-браузерах. В большинстве браузеров используется версия ECMAScript, определенная в третьей редакции ECMA-262, хотя разработчики уже приступили к реализации пятой редакции. Перечислим некоторые основные элементы этого языка.

- К базовым ECMAScript-типам данных относятся неопределенный (undefined), нулевой (null), логический (boolean), числовой (number) и строковый (string) типы.
- В отличие от других языков, в ECMAScript нет отдельных типов данных для целых чисел и чисел с плавающей точкой. Все числа представляются числовым типом.
- Также имеется сложный тип данных Object, который является базовым для всех объектов в языке.
- В строгом режиме налагаются ограничения на некоторые возможности языка, часто приводящие к ошибкам.
- ECMAScript содержит много базовых операторов, доступных в C и других C-подобных языках, в том числе арифметические и логические операторы, операторы отношений, эквивалентности и присваивания.
- В языке доступны управляющие инструкции, типичные для многих других языков, такие как `if`, `for` и `switch`.

Функции в ECMAScript работают иначе, чем функции в других языках. Вот некоторые их особенности.

- Указывать возвращаемое значение функции не требуется, поскольку любая функция может возвратить любое значение в любое время.
- Функции без возвращаемого значения на самом деле возвращают специальное значение `undefined`.
- Сигнатур у функций нет, потому что аргументы передаются в функцию как массив.
- В функции можно передавать любое количество аргументов. Они доступны как элементы объекта `arguments`.
- Перегрузка функций невозможна из-за отсутствия у них сигнатур.

4 Переменные, область видимости и память

- Примитивные и ссылочные значения
- Контекст выполнения
- Сборка мусора

Переменные в спецификации ECMA-262 во многом отличаются от переменных в других языках. Будучи слабо типизированными, они просто определяют имена для конкретных значений в конкретный момент времени. Поскольку никакие правила не определяют тип данных переменной, ее значение и тип со временем могут меняться. Кроме этой интересной, полезной и иногда спорной особенности у переменных есть много других нетривиальных аспектов.

Примитивные и ссылочные значения

ECMAScript-переменные могут содержать значения двух видов: примитивные и ссылочные. *Примитивные значения* (primitive values) — это просто атомарные элементы данных, в то время как *ссылочные значения* (reference values) — это объекты, которые могут состоять из нескольких значений.

Когда значение присваивается переменной, интерпретатор JavaScript должен определить, примитивное оно или ссылочное. Пять примитивных типов были рассмотрены в предыдущей главе: неопределенный (undefined), нулевой (null), логический (boolean), числовой (number) и строковый (string). Доступ к переменным этих

типов осуществляется *по значению* (by value), то есть вы работаете с фактическим значением, хранящимся в переменной.

Ссылочные значения — это объекты, хранящиеся в памяти. В отличие от других языков, в JavaScript невозможен прямой доступ к памяти, в том числе занимаемой объектами. Вместо этого при выполнении каких-либо действий над объектом вы на самом деле работаете не с самим объектом, а со *ссылкой* (reference) на него. Говорят, что доступ к таким значениям осуществляется *по ссылке*.

Во многих языках строки представляются объектами и соответственно считаются ссылочными типами. ECMAScript не следует этой традиции.

Динамические свойства

Примитивные и ссылочные значения определяются схожим образом: вы создаете переменную и присваиваете ей значение. Однако действия, которые можно выполнять со значениями переменных, различаются. При работе со ссылочными значениями можно в любое время добавлять, изменять и удалять их свойства и методы, например:

Листинг DynamicPropertiesExample01.htm


```
var person = new Object();
person.name = "Nicholas";
alert(person.name);    // "Nicholas"
```

Здесь мы создаем объект и сохраняем его в переменной `person`. Далее к объекту добавляется свойство `name`, которому присваивается строковое значение `"Nicholas"`. Новое свойство можно использовать, пока объект не будет уничтожен или пока свойство не будет явно удалено.

К примитивным значениям добавить свойства нельзя, хотя это и не является ошибкой.

Листинг DynamicPropertiesExample02.htm

```
var name = "Nicholas";
name.age = 27;
alert(name.age);    // неопределенное значение
```

Здесь для строки `name` определяется свойство `age`, которому затем присваивается значение 27, но уже в третьей инструкции свойство недоступно. Динамически добавленные свойства остаются доступны только у ссылочных значений.

Копирование значений

Примитивные и ссылочные значения не только хранятся, но и копируются поразному. Когда одна переменная с примитивным значением присваивается другой,

создается копия значения, хранящегося в объекте переменных, а затем она записывается по адресу новой переменной, например:

```
var num1 = 5;
var num2 = num1;
```

Здесь num1 содержит значение 5. Когда переменная num2 инициализируется значением num1, она также получает значение 5. Она никак не связана с num1, потому что содержит копию значения. Затем эти переменные можно использовать по отдельности без побочных эффектов (рис. 4.1).

Рис. 4.1

Когда ссылочное значение одной переменной присваивается другой, значение в объекте переменных также копируется в расположение новой переменной, но в этот раз оно является указателем на объект в куче. После копирования обе переменные указывают на один объект, поэтому изменения одной из них отражаются на другой:

```
var obj1 = new Object();
var obj2 = obj1;
obj1.name = "Nicholas";
alert(obj2.name);    // "Nicholas"
```

В этом примере переменной obj1 присваивается новый объект, после чего значение переменной копируется в obj2. Теперь обе переменные указывают на один объект, и когда для obj1 задается свойство name, оно становится доступным через obj2. На рис. 4.2 показаны отношения между переменными в объекте переменных и объектом в куче.

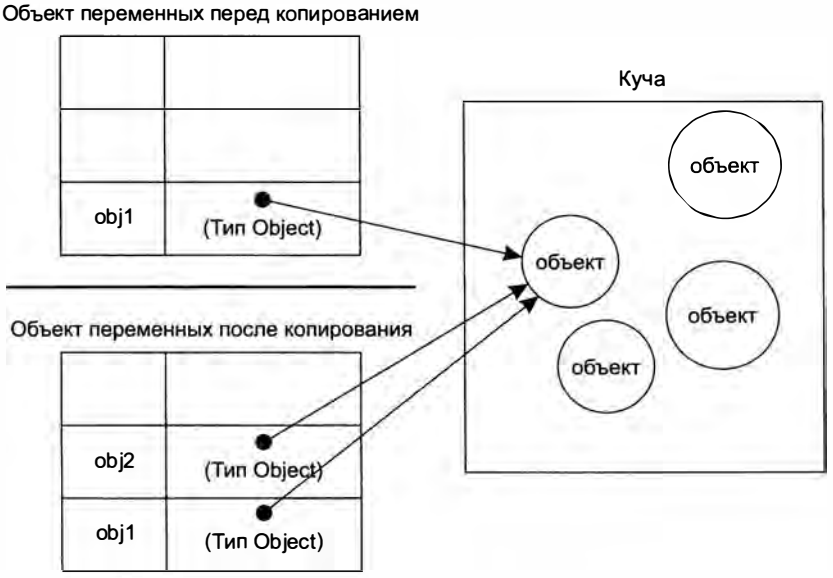

Рис. 4.2

Передача аргументов

Все аргументы функций в ECMAScript передаются по значению. Это означает, что значения извне функции копируются в аргументы внутри функции так же, как копируются значения переменных: примитивные — по одному сценарию, ссылочные — по другому. В то время как доступ к переменным возможен и по значению, и по ссылке, многие разработчики плохо понимают, что аргументы передаются только по значению.

Когда аргумент передается по значению, его значение просто копируется в локальную переменную (которой в ECMAScript соответствуют именованный аргумент и ячейка объекта `arguments`). При передаче аргумента по ссылке в локальной переменной сохраняется расположение его значения в памяти. Это означает, что изменения локальной переменной отражаются вне функции (в ECMAScript это невозможно). Рассмотрим следующий пример:

Листинг FunctionArgumentsExample01.htm


```
function addTen(num) {
    num +=10;
    return num;
}

var count = 20;
var result = addTen(count);
alert(count);     // 20 - без изменений
alert(result);    // 30
```

Здесь определяется функция addTen() с аргументом num, который, по сути, является локальной переменной. При вызове функции ей передается переменная count со значением 20, которое копируется в аргумент num для использования в функции addTen(). В функции к значению num прибавляется 10, но это не изменяет значение count вне функции. Аргумент num и переменная count не знают друг о друге, они просто имеют одинаковые значения. Если бы значение num было передано по ссылке, переменная count также стала бы равной 30 согласно изменению внутри функции. Все это очевидно при работе с примитивными значениями вроде чисел, но не с объектами. Взгляните на этот пример:

Листинг FunctionArgumentsExample02.htm

```
function setName(obj) {
    obj.name = "Nicholas";
}

var person = new Object();
setName(person);
alert(person.name);    // "Nicholas"
```

В этом коде переменной person назначается созданный объект, который затем передается в функцию setName, при этом он копируется в локальную переменную obj. Внутри функции переменные obj и person указывают на один и тот же объект, вследствие чего доступ к объекту с помощью obj выполняется по ссылке, хотя он и был передан в функцию по значению. При задании свойства name в функции это изменение отражается вне функции, потому что объект, на который указывает obj, находится в куче и доступен глобально. Многие разработчики ошибаются, когда думают, что если локальное изменение объекта проявляется глобально, значит, аргумент был передан по ссылке. Взгляните на измененный код, позволяющий убедиться, что объекты передаются по значению:

```
function setName(obj) {
    obj.name = "Nicholas";
    obj = new Object();
    obj.name = "Greg";
}

var person = new Object();
setName(person);
alert(person.name);    // "Nicholas"
```

Эта функция setName() содержит две дополнительные строки кода, которые переопределяют obj как новый объект с другим именем. Когда в функцию setName() передается объект person, его свойству name присваивается значение "Nicholas". Затем переменной obj назначается новый объект, а его свойству name присваивается значение "Greg". Если бы переменная person была передана по ссылке, она автоматически стала бы указывать на объект со свойством "Greg". Однако при выводе свойства person.name снова отображается значение "Nicholas", то есть первоначальная ссылка остается нетронутой, хотя значение аргумента внутри функции изменилось. При

перезаписи переменной obj в функции она становится указателем на локальный объект, который уничтожается сразу по завершении функции.

Аргументы функций в ECMAScript можно считать обычными локальными переменными.

Проверка типа

Чтобы определить, относится ли переменная к одному из примитивных типов, лучше всего использовать оператор typeof, упомянутый в предыдущей главе. Точнее говоря, это наилучший способ определить, является ли переменная строкой, числом, логическим значением или значением undefined. Для объектов и значения null оператор typeof возвращает "object":

Листинг DeterminingTypeExample01.htm


```
var s = "Nicholas";
var b = true;
var i = 22;
var u;
var n = null;
var o = new Object();

alert(typeof s);    // строка
alert(typeof i);    // число
alert(typeof b);    // логическое значение
alert(typeof u);    // неопределенное значение
alert(typeof n);    // объект
alert(typeof o);    // объект
```

Хотя оператор typeof вполне подходит для примитивных значений, при работе со ссылочными значениями от него мало пользы. Обычно вас не интересует, является ли значение объектом, — вам нужно знать, какого объект типа. Чтобы помочь с этим, ECMAScript предоставляет оператор instanceof со следующим синтаксисом:

```
результат = переменная instanceof конструктор
```

Оператор instanceof возвращает true, если переменная — экземпляр конкретного ссылочного типа, определяемого по его цепочке прототипов (см. главу 6):

```
alert(person instanceof Object);   // относится ли person к типу Object?
alert(colors instanceof Array);    // относится ли colors к типу Array?
alert(pattern instanceof RegExp);  // относится ли pattern к типу RegExp?
```

Все ссылочные значения по определению являются экземплярами типа Object, так что для ссылочных значений и конструктора Object оператор instanceof всегда возвращает true. Соответственно, для примитивных значений он всегда возвращает false, потому что примитивные значения — не объекты.

Оператор typeof также возвращает значение «function» для функций. Кроме того, в Safari до версии 5 включительно и Chrome до версии 7 включительно он из-за особенностей реализации возвращает «function» для регулярных выражений. В ECMA-262 указано, что оператор typeof должен возвращать значение «function» для любых объектов, реализующих внутренний метод [[Call]]. Поскольку регулярные выражения реализуют его в этих браузерах, typeof возвращает «function». В Internet Explorer и Firefox оператор typeof возвращает для регулярных выражений значение «object».

Контекст выполнения и область видимости

Концепция контекста выполнения, или просто *контекста* (context), очень важна в JavaScript. От контекста выполнения переменной или функции зависит, какие другие данные ей доступны и как она должна работать. С каждым контекстом выполнения связан *объект переменных* (variable object), содержащий все переменные и функции контекста. Он недоступен в коде, но используется за кулисами для обработки данных.

Наиболее общим является глобальный контекст выполнения. В зависимости от среды выполнения для ECMAScript-реализации он может представляться разными объектами. В веб-браузерах глобальным контекстом считается контекст объекта `window` (см. главу 8), так что все глобальные переменные и функции создаются как свойства и методы объекта `window`. Когда весь код в контексте выполнен, он уничтожается вместе со всеми определенными в нем переменными и функциями (глобальный контекст существует вплоть до завершения приложения, например до закрытия веб-страницы или веб-браузера).

У каждого вызова функции имеется свой контекст выполнения. При передаче управления в функцию ее контекст помещается в стек контекста, а при выходе из функции он извлекается из стека, при этом управление возвращается в прежний контекст. Так осуществляется контроль над последовательностью операций в ECMAScript-программе.

При выполнении кода в контексте создается *цепочка областей видимости* (scope chain) объектов переменных, которая обеспечивает упорядоченный доступ ко всем переменным и функциям, доступным в контексте выполнения. Первым звеном цепочки областей видимости всегда является объект переменных контекста, код которого выполнятся. Если контекст — функция, в качестве объекта переменных используется *объект активации* (activation object), который начинает существование с единственной переменной `arguments` (у глобального контекста ее нет). Каждый последующий объект переменных в цепочке относится к все более внешнему контексту, пока не достигается глобальный контекст. Объект переменных глобального контекста всегда последний в цепочке областей видимости.

При разрешении идентификатора его имя ищется в цепочке областей видимости. Поиск всегда осуществляется в направлении от первого звена к последнему, пока не обнаруживается идентификатор (если найти его не удается, обычно возникает ошибка).

Рассмотрим следующий код:

Листинг ExecutionContextExample01.htm


```
var color = "blue";

function changeColor(){
    if (color === "blue"){
        color = "red";
    } else {
        color = "blue";
    }
}

changeColor();
```

В этом простом примере функция `changeColor()` имеет цепочку областей видимости с двумя объектами: собственным объектом переменных (в котором определен объект `arguments`) и объектом переменных глобального контекста. Переменная `color` доступна в функции, потому что она имеется в цепочке областей видимости.

В локальном контексте вместе с глобальными переменными можно использовать локальные, например:

```
var color = "blue";

function changeColor(){
    var anotherColor = "red";

    function swapColors(){
        var tempColor = anotherColor;
        anotherColor = color;
        color = tempColor;

        // здесь доступны переменные color, anotherColor и tempColor
    }

    // здесь доступны переменные color и anotherColor, но не tempColor
    swapColors();
}

// здесь доступна только переменная color
changeColor();
```

В этом коде три контекста выполнения: глобальный контекст, локальный контекст функции `changeColor()` и локальный контекст функции `swapColors()`. В глобальном контексте определены переменная `color` и функция `changeColor()`. В локальном контексте `changeColor()` определены переменная `anotherColor` и функция `swapColors()`, но в нем также доступна переменная `color` из глобального контекста. Локальный контекст `swapColors()` содержит одну переменную `tempColor`, которая доступна только в этой функции, но не в глобальном контексте и не в локальном контексте `changeColor()`. В контексте `swapColors()` также полностью доступны переменные из двух других

контекстов выполнения, потому что они являются родительскими по отношению к нему. Цепочка областей видимости для этого примера представлена на рис. 4.3.

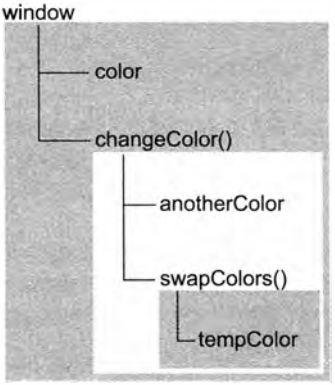

Рис. 4.3

На этом рисунке разные контексты выполнения показаны прямоугольниками. Внутреннему контексту по цепочке областей видимости доступно все из обоих внешних контекстов, но тем во внутреннем контексте недоступно ничего. Связь между контекстами линейна и упорядочена. Каждый контекст может искать переменные и функции в цепочке областей видимости по направлению наружу, но не внутрь. В цепочке областей видимости локального контекста `swapColors()` три объекта: объекты переменных функций `swapColors()` и `changeColor()` и глобальный объект переменных. При обращении к переменной или функции в локальном контексте `swapColors()` начинается поиск ее имени в локальном объекте переменных. Если найти имя не удается, выполняется поиск в следующем объекте цепочки. Цепочка областей видимости контекста `changeColor()` содержит его собственный объект переменных и глобальный объект переменных, а контекст `swapColors()` в нем недоступен.

Аргументы функций считаются переменными и подчиняются тем же правилам доступа, что и любые другие переменные в контексте выполнения.

Приращение цепочки областей видимости

Основных типов контекстов выполнения два, глобальный контекст и контекст функции (есть также третий внутри вызова `eval()`), но существуют и другие способы приращения цепочки областей видимости. Некоторые инструкции временно добавляют области видимости к началу цепочки, которые затем удаляются. Это происходит в двух случаях:

- при входе в блок `catch` инструкции `try-catch`;
- при выполнении инструкции `with`.

Обе эти инструкции добавляют объект переменных к началу цепочки областей видимости. В случае инструкции `with` к цепочке областей видимости добавляется переданный инструкции объект, а в случае блока `catch` создается новый объект переменных с объявлением объекта ошибки. Рассмотрим пример:

Листинг ExecutionContextExample03.htm


```
function buildUrl() {
    var qs = "?debug=true";

    with(location){
        var url = href + qs;
    }

    return url;
}
```

В этом примере инструкция `with` применяется к объекту `location`, который добавляется к цепочке областей видимости. В функции `buildUrl()` определена единственная переменная `qs`. При обращении к переменной `href` на самом деле используется переменная `location.href`, которая относится к отдельному объекту переменных. При ссылке на `qs` используется переменная, определенная в функции `buildUrl()` и относящаяся к объекту переменных контекста `buildUrl()`. Также в инструкции `with` объявляется переменная `url`, которая становится частью контекста функции и потому может быть возвращена как значение функции.

В Internet Explorer до версии 8 включительно ошибка, перехваченная в блоке catch, добавляется в объект переменных контекста выполнения, а не блока catch, из-за чего она доступна даже вне блока catch. Это отклонение от правила устранено в Internet Explorer 9.

Отсутствие блочных областей видимости

Блочных областей видимости в JavaScript нет, о чем часто забывают. В других C-подобных языках блоки кода в фигурных скобках имеют собственные области видимости (или контексты выполнения в терминах ECMAScript), что делает возможным условное определение переменных. Например, следующий JavaScript-код работает не так, как его аналоги во многих других языках:

```
if (true) {
    var color = "blue";
}

alert(color);  // "blue"
```

Здесь в инструкции `if` определяется переменная `color`. В языках вроде C, C++ и Java она была бы уничтожена после выполнения `if`, однако в JavaScript объявляемая переменная добавляется в текущий контекст выполнения (глобальный в данном

случае). Об этом важно помнить при создании циклов for, которые обычно выглядят так:

```
for (var i=0; i < 10; i++){
    doSomething(i);
}

alert(i);    // 10
```

В языках с блочными областями видимости раздел инициализации в цикле for определяет переменные, существующие только в контексте цикла. В JavaScript переменная i, созданная в инструкции for, продолжает существовать вне цикла после его выполнения.

Объявление переменной

При объявлении переменной с помощью ключевого слова var она автоматически добавляется в текущий контекст. В функции им является локальный контекст, а в инструкции with — контекст функции. Если переменная инициализируется без предварительного объявления, она автоматически добавляется в глобальный контекст. Рассмотрим следующий пример:

Листинг ExecutionContextExample04.htm


```
function add(num1, num2) {
    var sum = num1 + num2;
    return sum;
}

var result = add(10, 20);  // 30
alert(sum);                // ошибка, потому что sum недействительна
```

В функции add() определяется локальная переменная sum, которой присваивается результат сложения. Затем это значение возвращается из функции, но сама переменная sum недоступна вне функции. Если опустить ключевое слово var, переменная sum станет доступна после вызова add():

Листинг ExecutionContextExample05.htm


```
function add(num1, num2) {
    sum = num1 + num2;
    return sum;
}

var result = add(10, 20);  // 30
alert(sum);                // 30
```

Здесь переменная sum инициализируется без объявления. При вызове add() она создается в глобальном контексте и продолжает существовать даже после завершения функции.

Инициализация JavaScript-переменных без их объявления часто ошибочна. Во избежание проблем рекомендуется всегда объявлять переменные до их инициализации. В строгом режиме инициализация необъявленной переменной считается ошибкой.

Поиск идентификатора

При доступе к идентификатору для чтения или записи в конкретном контексте необходимо выяснить, какую переменную он представляет. Для этого начинается поиск идентификатора с конкретным именем с начала цепочки областей видимости. Если имя идентификатора обнаруживается в локальном контексте, поиск прекращается и переменная используется. Если имя переменной в локальном контексте отсутствует, запускается поиск в следующем элементе цепочки (имейте в виду, что у объектов в цепочке областей видимости также есть цепочка прототипов, так что поиск может охватывать и цепочку прототипов каждого объекта). Это продолжается, пока не будет достигнут объект переменных глобального контекста. Если и в нем нет идентификатора, значит, он не был объявлен.

Рассмотрим следующий пример:

Листинг ExecutionContextExample06.htm


```
var color = "blue";

function getColor(){
    return color;
}

alert(getColor());  // "blue"
```

Здесь при вызове функции `getColor()` выполняется обращение к переменной `color`, при этом начинается поиск, включающий два этапа. Сначала идентификатор `color` ищется в объекте переменных функции `getColor()`. Если найти его не удается, выполняется поиск в следующем объекте переменных (из глобального контекста). Поскольку переменная `color` определена в нем, поиск завершается. Этот процесс иллюстрирует рис. 4.4.

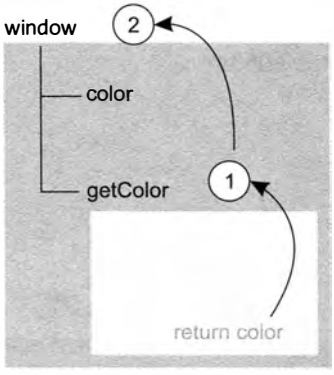

Рис. 4.4

Таким образом, доступ к локальной переменной автоматически блокирует продолжение поиска в другом объекте переменных. Это означает, что идентификатор в родительском контексте не используется, если в локальном контексте есть идентификатор с тем же именем, например:

Листинг ExecutionContextExample07.htm

```
var color = "blue";

function getColor(){
    var color = "red";
    return color;
}

alert(getColor());  // "red"
```

В этом измененном коде при вызове функции `getColor()` объявляется локальная переменная `color`. При выполнении второй строки функции начинается поиск в локальном контексте, в котором обнаруживается переменная `color` со значением `"red"`. На этом поиск прекращается и значение локальной переменной возвращается из функции. В коде после объявления локальной переменной `color` глобальная переменная `color` недоступна без квалификатора области видимости (`window.color`).

Поиск переменных имеет свою цену. Доступ к локальным переменным выполняется быстрее, чем к глобальным, потому что для этого не нужно просматривать цепочку областей видимости. С другой стороны, алгоритмы поиска идентификаторов становятся в интерпретаторах JavaScript все эффективнее, так что это различие может со временем стать несущественным.

Сборка мусора

JavaScript — это язык со сборкой мусора, то есть за управление памятью при работе сценариев отвечает среда выполнения. В языках вроде C и C++ слежение за использованием памяти относится к важнейшим задачам и является источником многих проблем. JavaScript освобождает разработчиков от забот по управлению памятью, автоматически выделяя сценариям нужную память и возвращая в среду память, которая больше не используется. Идея сборки мусора проста: нужно выяснить, какие переменные больше не потребуются, и освободить связанную с ними память. Сборщик мусора запускается периодически с заданной частотой или в предопределенные моменты выполнения кода.

При нормальном жизненном цикле локальной переменной она создается в ходе выполнения функции. В этот момент для нее выделяется память в стеке (и, возможно, в куче). Далее переменная используется в функции, и рано или поздно функция завершается. После этого переменная больше не нужна, поэтому ее память может быть освобождена для повторного использования. В описанном случае ясно, что переменная стала ненужной, но не все ситуации так очевидны. Сборщик мусора

должен отслеживать, какие переменные могут потребоваться, а какие нет, чтобы можно было выявить те из них, которые допускают освобождение памяти. Способ выявления неиспользуемых переменных зависит от реализации, но в браузерах традиционно применяются две стратегии.

Отслеживание и очистка

Наиболее популярным способом сборки мусора в JavaScript является *отслеживание и очистка* (mark-and-sweep). Когда переменная появляется в контексте, например при объявлении внутри функции, она помечается как находящаяся в контексте. Память переменных в контексте не должна освобождаться, потому что они могут потребоваться, пока продолжается выполнение кода в этом контексте. Когда переменная покидает контекст, она помечается как находящаяся вне контекста.

Переменные могут помечаться разными способами — например, с помощью бита-переключателя или списков переменных «в контексте» и «вне контекста». Реализация пометок не имеет значения, важен принцип.

При запуске сборщика мусора он тем или иным способом маркирует все переменные, находящиеся в памяти, а затем снимает пометки со всех переменных в контексте и переменных, на которые они ссылаются. Переменные, оставшиеся помеченными после этого, считаются готовыми к удалению, потому что они недостижимы ни для каких переменных, находящихся в контексте. Затем сборщик мусора очищает память, уничтожая все помеченные переменные и возвращая связанную с ними память в среду.

На 2008 год в Internet Explorer, Firefox, Opera, Chrome и Safari для сборки мусора использовался метод отслеживания и очистки (или его вариации), но время сборки мусора определялось в них по-разному.

Подсчет ссылок

Еще одним способом сборки мусора, который оказался не столь востребованным, является *подсчет ссылок* (reference counting). Его идея в том, что каждое значение отслеживает количество ссылок на него. Когда переменная объявляется и ей присваивается ссылочное значение, счетчик ссылок равен единице. Если это же значение присваивается другой переменной, счетчик ссылок увеличивается, а если переменная со ссылкой на значение перезаписывается другим значением, он уменьшается. Когда счетчик ссылок на значение достигает нуля, оно становится недоступным и занимаемую им память можно безопасно освободить, что и делает сборщик мусора при следующем запуске.

Подсчет ссылок первоначально использовался в Netscape Navigator 3.0, но при этом сразу возникла проблема циклических ссылок. *Циклическая ссылка* (circular reference) имеет место, если объект A указывает на объект B, а B — на A, например:

```
function problem(){
    var objectA = new Object();
    var objectB = new Object();

    objectA.someOtherObject = objectB;
    objectB.anotherObject = objectA;
}
```

В этом примере объекты `objectA` и `objectB` ссылаются друг на друга посредством своих свойств, и у каждого из них счетчик ссылок равен двум. В системе с отслеживанием и очисткой это не было бы проблемой, потому что оба объекта покинули бы область видимости при завершении функции. Однако в системе с подсчетом ссылок оба объекта продолжают существовать после выполнения функции, потому что их счетчики ссылок никогда не достигают нуля. Если эта функция вызывается многократно, большие объекты памяти так и не освобождаются. По этой причине в Netscape Navigator 4.0 сборка мусора с подсчетом ссылок была заменена алгоритмом отслеживания и очистки, но проблемы подсчета ссылок на этом не закончились.

Не все объекты в Internet Explorer 8 и более ранних версий являются естественными объектами JavaScript. BOM- и DOM-объекты реализованы как объекты COM (Component Object Model — объектная модель программных компонентов) на C++ и используют для сборки мусора подсчет ссылок. Несмотря на то что интерпретатор JavaScript в Internet Explorer использует отслеживание и очистку, из-за подсчета ссылок для COM-объектов может возникать знакомая проблема циклических ссылок, например:

```
var element = document.getElementById("some_element");
var myObject = new Object();
myObject.element = element;
element.someObject = myObject;
```

В этом примере формируется циклическая ссылка между DOM-элементом `element` и JavaScript-объектом `myObject`. Свойство `element` переменной `myObject` указывает на DOM-элемент, а его свойство `someObject`, в свою очередь, указывает на `myObject`. Из-за этой циклической ссылки память, занимаемая DOM-элементом, не освобождается, даже если он удаляется со страницы.

Для предотвращения подобных проблем с циклическими ссылками следует разрывать связи между естественными JavaScript-объектами и DOM-элементами по завершении работы с ними. Например, в следующем коде циклическая ссылка из предыдущего примера уничтожается:

```
myObject.element = null;
element.someObject = null;
```

Присвоение переменной значения `null` разрывает связь между ней и значением, на которое она ссылалась до этого. При следующем запуске сборщика мусора он может удалить это значение и возвратить память в среду.

Чтобы решить некоторые из этих проблем, в Internet Explorer 9 BOM- и DOM-объекты были сделаны настоящими JavaScript-объектами. Это унифицировало алгоритмы сборки мусора и устранило частые причины утечек памяти.

Циклические ссылки могу возникать и в ряде других сценариев, которые мы рассмотрим в следующих главах.

Производительность

Периодическая сборка мусора может требовать много ресурсов, если в памяти находится большое количество переменных, поэтому важно подбирать для нее подходящее время. Когда-то в Internet Explorer это было серьезной проблемой. Сборка мусора запускалась в этом браузере согласно количеству выделенных фрагментов памяти, а именно — по достижении 256 переменных, 4096 литералов объектов/массивов и ячеек массивов или 64 Кбайт строковых данных. Проблема заключалась в том, что если для решения некой задачи в сценарии требовалось такое большое количество переменных, обычно эти пределы затем достигались снова и снова, что каждый раз запускало сборку мусора. Это существенно снижало быстродействие и вынудило изменить алгоритм сборки мусора в Internet Explorer 7.

В сборщике мусора Internet Explorer 7 была реализована динамическая настройка порогов выделения памяти для переменных, литералов и ячеек массивов. Сначала в Internet Explorer 7 действуют те же ограничения, что и в Internet Explorer 6. Если при сборке мусора освобождается менее 15 % выделенных фрагментов памяти, порог для переменных, литералов и ячеек массивов удваивается. Если освобождается более 85 % выделенных фрагментов, восстанавливаются пороговые значения, предлагаемые по умолчанию. Это простое изменение значительно повысило быстродействие браузера при обработке веб-страниц с большим объемом JavaScript-кода.

В некоторых браузерах можно инициировать сборку мусора вручную, но делать это не рекомендуется. В Internet Explorer для немедленного запуска сборки мусора используется метод window.CollectGarbage(), а в Opera 7 и более поздних версий — метод window.opera.collect().

Управление памятью

В средах программирования со сборкой мусора разработчикам обычно не приходится волноваться об управлении памятью. Однако интерпретаторы JavaScript работают в среде, уникальной в плане управления памятью и сборки мусора. Как правило, объем памяти, доступной в веб-браузерах, гораздо меньше, чем в обычных приложениях. Он ограничивается в основном ради безопасности, чтобы сценарии JavaScript в веб-страницах не могли вызвать сбой операционной системы, израсходовав всю системную память. Ограничения памяти влияют не только на выделение памяти для переменных, но и на стек вызовов и количество инструкций, выполняемых в одном потоке.

Экономия памяти в JavaScript-сценариях ускоряет обработку страниц, а наилучший способ оптимизировать использование памяти — это хранить в коде только те данные, которые требуются для его выполнения. Если данные больше не нужны, лучше всего присвоить соответствующей переменной значение `null`, чтобы разорвать ее связь с данными. Этот совет относится преимущественно к глобальным переменным и свойствам глобальных объектов. В случае локальных переменных ссылки на данные удаляются автоматически, когда переменные покидают контекст, например:

```
function createPerson(name){
    var localPerson = new Object();
    localPerson.name = name;
    return localPerson;
}

var globalPerson = createPerson("Nicholas");

// какие-то действия с globalPerson

globalPerson = null;
```

В этом коде переменной `globalPerson` присваивается значение, возвращенное функцией `createPerson()`. Внутри функции создается объект `localPerson`, к которому добавляется свойство `name`. Затем этот объект возвращается из функции и назначается переменной `globalPerson`. Переменная `localPerson` покидает контекст при завершении функции `createPerson()`, так что разрывать ее связь со значением не требуется. Что касается `globalPerson`, то это глобальная переменная, поэтому когда она больше не требуется, ей присваивается значение `null`.

Помните, что разрыв связи переменной со значением не приводит к автоматическому освобождению занятой им памяти. Значение просто выводится из контекста, что делает возможным возвращение памяти при следующей сборке мусора.

Резюме

JavaScript-переменные могут содержать значения двух типов: примитивные и ссылочные. Примитивные значения относятся к одному из пяти примитивных типов данных: неопределенному (undefined), нулевому (null), логическому (boolean), числовому (number) и строковому (string). Перечислим некоторые характеристики JavaScript-значений.

- Примитивные значения имеют фиксированный размер и хранятся в памяти в стеке.
- При копировании примитивного значения из одной переменной в другую создается его копия.
- Ссылочные значения — это объекты, которые хранятся в памяти в куче.

- Переменная со ссылочным значением на самом деле содержит не объект, а лишь указатель на него.
- При копировании ссылочного значения копируется только указатель, так что в результате обе переменные ссылаются на один объект.
- Для определения типа примитивных и ссылочных значений используются операторы `typeof` и `instanceof` соответственно.

И примитивные и ссылочные переменные существуют в том или ином контексте выполнения (называемом также областью видимости), который определяет время жизни переменной, а также те части кода, в которых она доступна.

- Контекст выполнения может быть глобальным или локальным. В последнем случае он ограничен функцией.
- При входе в каждый новый контекст выполнения создается цепочка областей видимости, служащая для поиска переменных и функций.
- В локальном контексте функции доступны переменные не только из текущей области видимости, но и из всех внешних контекстов, включая глобальный.
- В глобальном контексте доступны только переменные и функции из глобальной области видимости, а непосредственного доступа к каким-либо данным в локальных контекстах нет.
- Контексты выполнения переменных помогают управлять освобождением памяти.

В среде программирования JavaScript применяется сборка мусора, то есть разработчику не нужно беспокоиться о выделении или освобождении памяти. Механизм сборки мусора в JavaScript работает следующим образом.

- Значения, покидающие область видимости, автоматически помечаются как подлежащие удалению и удаляются во время сборки мусора.
- Наиболее популярный алгоритм сборки мусора — отслеживание и очистка. Он помечает неиспользуемые значения, а затем освобождает занимаемую ими память.
- В алгоритме подсчета ссылок отслеживается количество ссылок на конкретное значение. В интерпретаторах JavaScript он больше не применяется, но Internet Explorer использует его при доступе к сторонним объектам (таким, как DOM-элементы).
- Подсчет ссылок приводит к проблемам при наличии циклических ссылок в коде.
- Разрыв связей переменных со значениями не только помогает при наличии циклических ссылок, но и повышает эффективность сборки мусора в целом. Когда глобальные объекты и их свойства становятся ненужными, присваивайте им значение `null`, чтобы оптимизировать возвращение памяти системе.

5 Ссылочные типы

- Работа с объектами
- Создание и использование массивов
- Базовые типы данных в JavaScript
- Работа с примитивными типами и оболочками примитивных типов

Ссылочное значение (объект) — это экземпляр конкретного *ссылочного типа* (reference type). В ECMAScript ссылочные типы представляют собой структуры, которые используют для группировки данных и функций и часто ошибочно называют *классами* (classes). Хотя технически ECMAScript относится к объектно-ориентированным языкам, в нем нет некоторых традиционных базовых конструкций объектно-ориентированного программирования, в том числе классов и интерфейсов. Ссылочные типы также иногда называют *определениями объектов* (object definitions), потому что они описывают свойства и методы, которые должны быть у объектов.

Хотя ссылочные типы похожи на классы, эти концепции не эквивалентны. Во избежание путаницы термин «класс» далее в книге не используется.

Объекты представляют собой *экземпляры* (instances) конкретного ссылочного типа и создаются с помощью оператора `new`, за которым следует *конструктор* (constructor). Конструктор — это просто функция, служащая для создания объектов, например:

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

Эта инструкция создает экземпляр ссылочного типа `Object` и сохраняет его в переменной `person`. Конструктор `Object()` создает простой объект, содержащий лишь свойства и методы, предлагаемые по умолчанию. ECMAScrit предоставляет ряд встроенных ссылочных типов, таких как `Object`, для решения типичных задач программирования.

Тип Object

До сих пор в большинстве примеров со ссылочными значениями фигурировал тип `Object` — один из наиболее востребованных ECMAScript-типов. Хотя экземпляры `Object` не могут похвастать широкими возможностями, они идеально подходят для хранения и передачи данных в приложении.

Есть два способа явного создания экземпляров `Object`. Первый — использовать оператор `new` с конструктором `Object`:

Листинг ObjectTypeExample01.htm


```
var person = new Object();
person.name = "Nicholas";
person.age = 29;
```

Второй способ — использовать *литерал объекта* (object literal). Так называется сокращенная форма определения объекта, которая упрощает создание объектов с большим количеством свойств. С ее помощью объект `person` из предыдущего примера можно определить так:

Листинг ObjectTypeExample02.htm

```
var person = {
    name : "Nicholas",
    age : 29
};
```

В этом примере левая фигурная скобка { указывает начало литерала объекта, потому что мы имеем дело с *контекстом выражения* (expression context). Так в ECMAScript называется контекст, в котором ожидается значение (выражение). Справа от оператора присваивания должно находиться значение, поэтому левая фигурная скобка после знака равенства определяет начало выражения. Та же левая фигурная скобка в *контексте инструкции* (statement context), например после условия в инструкции `if`, определяет начало блочной инструкции.

После скобки указано свойство `name`, а за ним — двоеточие и значение свойства. В литерале объекта свойства разделяются запятыми. В нашем случае запятая указывается после строки `"Nicholas"`, но не после значения 29, потому что `age` — это последнее свойство объекта. Добавление запятой после значения последнего свойства вызывает ошибку в Internet Explorer 7 и более ранних версий, а также в Opera.

В литералах объектов имена свойств могут быть строками и числами, например:

```
var person = {
    "name" : "Nicholas",
    "age" : 29,
    5: true
};
```

В этом примере создается объект со свойствами `name`, `age` и `"5"`. Имейте в виду, что числовые имена свойств автоматически преобразуются в строки.

Используя нотацию литералов объектов, можно создавать объекты, содержащие только свойства и методы, предлагаемые по умолчанию. Для этого следует оставить место в фигурных скобках пустым, например:

```
var person = {};      // то же, что new Object()
person.name = "Nicholas";
person.age = 29;
```

Этот код эквивалентен первому примеру в этом разделе, хотя и выглядит немного странно. Ради ясности рекомендуется использовать литералы объектов, только если вы собираетесь задавать в них свойства.

При определении объекта с помощью литерала конструктор Object на самом деле не вызывается (он вызывался в Firefox 2 и более ранних версий, но уже не в Firefox 3).

Хотя приемлем любой способ создания экземпляров `Object`, разработчики обычно предпочитают нотацию литералов объектов, потому что она более лаконична и визуально группирует все связанные данные. Кроме того, она часто применяется, если нужно передать в функцию много необязательных аргументов.

Листинг ObjectTypeExample04.htm


```
function displayInfo(args) {
    var output = "";

    if (typeof args.name == "string") {
        output += "Name: " + args.name + "\n";
    }

    if (typeof args.age == "number") {
        output += "Age: " + args.age + "\n";
    }

    alert(output);
}

displayInfo({
    name: "Nicholas",
    age: 29
});

displayInfo({
    name: "Greg"
});
```

Функция `displayInfo()` принимает единственный аргумент `args`, который может содержать только одно из свойств, `name` и `age`, или ни одного. Функция проверяет существование свойств с помощью оператора `typeof` и составляет сообщение

с имеющимися значениями. Она дважды вызывается с разными данными, которые передаются ей как литералы объектов, и оба раза выводится правильный результат.

Этот способ передачи аргументов в функции хорош при большом количестве необязательных аргументов. Вообще говоря, с именованными аргументами работать проще, но если их много, код становится громоздким. Лучше всего передавать обязательные значения как именованные аргументы и использовать литерал объекта для передачи многочисленных необязательных значений.

Для доступа к свойствам объектов обычно применяется *точечная нотация* (dot notation), типичная для многих объектно-ориентированных языков, но можно также использовать *скобочную нотацию* (bracket notation). В этом случае имя свойства указывается как строка в квадратных скобках, например:

```
alert(person["name"]);    // "Nicholas"
alert(person.name);       // "Nicholas"
```

Функционально эти подходы одинаковы. Главное преимущество скобочной нотации в том, что она позволяет использовать переменные для доступа к свойствам:

```
var propertyName = "name";
alert(person[propertyName]);  // "Nicholas"
```

Скобочная нотация также полезна, если имя свойства содержит ключевое или зарезервированное слово либо знак, вызывающий синтаксическую ошибку:

```
person["first name"] = "Nicholas";
```

Поскольку имя `"first name"` содержит пробел, вы не можете использовать для доступа к нему точечную нотацию. Благодаря скобочной нотации возможен доступ к свойствам, имена которых содержат не только алфавитно-цифровые знаки.

Если же доступ к свойствам по именам с помощью переменных не требуется, обычно точечная нотация предпочтительнее.

Тип Array

Пожалуй, второе место по популярности после типа `Object` в ECMAScript занимает тип `Array` (массив). ECMAScript-массивы сильно отличаются от массивов в большинстве других языков программирования. Как и в других языках, ECMAScript-массивы — это упорядоченные списки данных, но в отличие от других языков, они могут содержать значения разных типов. Это означает, что можно создать массив со строкой в первой позиции, числом во второй, объектом в третьей и т. д. ECMAScript-массивы имеют динамические размеры, автоматически увеличиваясь при добавлении данных.

Создать массив можно двумя способами. Первый — использовать конструктор `Array`:

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

Если известно количество элементов массива, можно передать его в конструктор, и тогда автоматически будет создано свойство `length` (длина массива) с этим значением. Например, следующий код создает массив со значением `length`, равным 20:

```
var colors = new Array(20);
```

В конструктор `Array` можно также передать элементы, которые нужно добавить в массив. Например, здесь создается массив с тремя строками:

```
var colors = new Array("red", "blue", "green");
```

Если в конструктор передается одно значение, все немного сложнее. При передаче числового аргумента всегда создается массив с указанным количеством элементов, но если аргумент имеет другой тип, создается массив с этим единственным элементом, например:

Листинг ArrayTypeExample01.htm

```
var colors = new Array(3);     // массив с тремя элементами
var names = new Array("Greg"); // массив с одним элементом, строкой "Greg"
```

При вызове конструктора `Array` можно опустить оператор `new` без изменения результата:

```
var colors = Array(3);         //массив с тремя элементами
var names = Array("Greg");     //массив с одним элементом, строкой "Greg"
```

Второй способ создать массив — использовать нотацию *литерала массива* (array literal), который представляет собой список разделенных запятыми элементов в квадратных скобках, например:

Листинг ArrayTypeExample02.htm

```
var colors = ["red", "blue", "green"];  // массив с тремя строками
var names = [];                         // пустой массив

// НЕ ДЕЛАЙТЕ ТАК! Создается массив с двумя или тремя элементами
var values = [1,2,];

// НЕ ДЕЛАЙТЕ ТАК! Создается массив с пятью или шестью элементами
var options = [,,,,,];
```

Здесь в первой строке создается массив с тремя строковыми переменными. Во второй строке с помощью квадратных скобок создается пустой массив. Следующая инструкция демонстрирует проблему, которая возникает, если ввести лишнюю запятую вслед за последним значением в литерале массива. Из-за ошибки в реализации

литералов ECMAScript-массивов в Internet Explorer 8 и более ранних версий в массив `values` будут добавлены три элемента со значениями 1, 2 и `undefined`, а во всех остальных браузерах `values` будет содержать два элемента со значениями 1 и 2. В четвертом примере показана другая разновидность этой ошибки: в Internet Explorer 9+, Firefox, Opera, Safari и Chrome эта инструкция создает массив с пятью элементами, а в Internet Explorer 8 и более ранних версий — с шестью. Если значения между запятыми отсутствуют, каждый элемент получает значение `undefined`, что логически эквивалентно вызову конструктора `Array` и передаче в него соответствующего количества элементов. Однако из-за того, что литералы массивов некорректно реализованы в ранних версиях Internet Explorer, использовать этот синтаксис не рекомендуется.

Как и в случае объектов, при создании массива с помощью литерала конструктор Array не вызывается (исключение — браузеры Firefox до версии 3).

Для получения и задания значений массива используется индекс с отсчетом от нуля в квадратных скобках:

```
var colors = ["red", "blue", "green"];  // определение массива строк
alert(colors[0]);                       // вывод первого элемента
colors[2] = "black";                    // изменение третьего элемента
colors[3] = "brown";                    // добавление четвертого элемента
```

Индекс в квадратных скобках указывает значение, к которому нужно обратиться. Если он меньше количества элементов в массиве, возвращается значение соответствующего элемента: например, для `colors[0]` в приведенном примере выводится строка `"red"`. Задаются значения точно так же, при этом значение в указанной позиции заменяется новым. Если при задании значения указан индекс за пределами массива, как в случае `colors[3]`, длина массива автоматически изменяется на значение этого индекса, увеличенное на единицу (для индекса 3 из примера она увеличивается до 4).

Количество элементов в массиве хранится в свойстве `length`, которое всегда возвращает неотрицательное число:

```
var colors = ["red", "blue", "green"];  // массив с тремя строками
var names = [];                         // пустой массив

alert(colors.length);    // 3
alert(names.length);     // 0
```

Свойство `length` уникально тем, что оно доступно не только для чтения. С его помощью можно легко удалять или добавлять конечные элементы массива.

Листинг ArrayFilterExample03.htm


```
var colors = ["red", "blue", "green"];  // массив с тремя строками
colors.length = 2;
alert(colors[2]);      // undefined
```

Массив colors сначала содержит три значения. Когда свойству length присваивается значение 2, последний элемент (в позиции 2) удаляется и становится недоступен как colors[2]. Если присвоить свойству length значение, превышающее количество элементов в массиве, новые элементы получат значения undefined:

Листинг ArrayFilterExample04.htm


```
var colors = ["red", "blue", "green"];  // массив с тремя строками
colors.length = 4;
alert(colors[3]);      // undefined
```

В этом коде свойству length массива colors присваивается значение 4, хотя массив содержит только три элемента. Позиции 3 в массиве нет, поэтому при обращении к этому значению возвращается специальное значение undefined.

Свойство length также полезно при добавлении элементов в конец массива:

Листинг ArrayFilterExample05.htm

```
var colors = ["red", "blue", "green"];  // массив с тремя строками
colors[colors.length] = "black"; // добавление черного цвета в позиции 3
colors[colors.length] = "brown"; // добавление коричневого цвета в позиции 4
```

Последний элемент массива всегда находится в позиции length – 1, так что индекс следующей ячейки равен length. Каждый раз при добавлении элемента за последним элементом массива свойство length автоматически обновляется. Это означает, что инструкция colors[colors.length] задает значение в позиции 3 во второй строке примера и в позиции 4 в последней строке. При добавлении элемента в позиции за пределами массива автоматически вычисляется его новая длина, для чего к позиции прибавляется 1, например:

Листинг ArrayFilterExample06.htm

```
var colors = ["red", "blue", "green"];  // массив с тремя строками
colors[99] = "black";    // добавление черного цвета в позиции 99
alert(colors.length);    // 100
```

В этом коде в массив colors добавляется значение в позиции 99, при этом длина становится равна 100 (99 + 1). Элементы с индексами от 3 до 98 не существуют, и при доступе к ним возвращается значение undefined.

Массивы могут содержать не более 4 294 967 295 элементов, чего достаточно для решения почти любых задач. Если попытаться добавить в массив еще больше элементов, возникнет исключение. Попытка создать массив с первоначальным размером, близким к этому максимуму, может вызвать ошибку из-за длительного выполнения сценария.

Идентификация массивов

Одной из классических проблем ECMAScript 3 является надежность определения того, является ли конкретный объект массивом. При работе с одной веб-страницей

(а значит, и с одной областью видимости) для этого можно использовать оператор `instanceof`:

```
if (value instanceof Array){
    // какие-то действия с массивом
}
```

К сожалению, оператор `instanceof` предполагает, что имеется единственный глобальный контекст выполнения. Если веб-страница содержит несколько фреймов, то глобальных контекстов выполнения тоже несколько и конструктор `Array` имеет как минимум две версии. Если бы нужно было передать массив из одного фрейма в другой, он имел бы иную функцию конструктора, чем массив, изначально созданный во втором фрейме.

Для обходного решения этой проблемы в ECMAScript 5 имеется метод `Array.isArray()`, позволяющий наверняка узнать, является ли конкретное значение массивом, независимо от того, в каком глобальном контексте выполнения оно было создано:

```
if (Array.isArray(value)){
    // какие-то действия с массивом
}
```

Метод `Array.isArray()` доступен в Internet Explorer 9+, Firefox 4+, Safari 5+, Opera 10.5+ и Chrome. Сведения о надежной идентификации массивов в браузерах, которые еще не поддерживают этот метод, можно найти в главе 22.

Методы преобразования массивов

Как уже было сказано, у всех объектов есть методы `toLocaleString()`, `toString()` и `valueOf()`. Методы `toString()` и `valueOf()`, будучи вызванными для массива, возвращают один и тот же результат, а именно — строку из строковых эквивалентов значений массива, разделенных запятыми. При составлении итоговой строки метод `toString()` вызывается для каждого элемента массива. Рассмотрим пример:

Листинг ArrayFilterExample07.htm


```
var colors = ["red", "blue", "green"];  // массив с тремя строками
alert(colors.toString());   // red,blue,green
alert(colors.valueOf());    // red,blue,green
alert(colors);              // red,blue,green
```

В этом коде методы `toString()` и `valueOf()` сначала вызываются явно, возвращая строковое представление массива, которое содержит его отдельные значения, разделенные запятыми. Затем в метод `alert()` передается сам массив. Поскольку метод `alert()` ожидает строку, при этом неявно вызывается метод `toString()`, благодаря чему выводится тот же результат, что и при непосредственном вызове `toString()`.

Метод `toLocaleString()` не всегда возвращает то же значение, что и методы `toString()` и `valueOf()`. Если вызвать его для массива, он также составляет строку значений массива, разделенных запятыми, но в отличие от двух других методов вызывает для получения значений метод `toLocaleString()` каждого элемента массива, а не метод `toString()`. Взгляните на следующий пример:

Листинг ArrayTypeExample08.htm


```
var person1 = {
    toLocaleString : function () {
        return "Nikolaos";
    },

    toString : function() {
        return "Nicholas";
    }
};

var person2 = {
    toLocaleString : function () {
        return "Grigorios";
    },

    toString : function() {
        return "Greg";
    }
};

var people = [person1, person2];
alert(people);                      // Nicholas,Greg
alert(people.toString());           // Nicholas,Greg
alert(people.toLocaleString());     // Nikolaos,Grigorios
```

Здесь определяются объекты `person1` и `person2` с методами `toString()` и `toLocaleString()`, которые возвращают разные значения. После этого создается массив `people`, содержащий оба объекта. При передаче массива в метод `alert()` выводится строка `"Nicholas,Greg"`, потому что для каждого элемента массива вызывается метод `toString()` (как и при явном вызове `toString()` в следующей строке кода). Когда для массива вызывается метод `toLocaleString()`, выводится результат `"Nikolaos,Grigorios"`, потому что в этом случае для каждого элемента массива вызывается метод `toLocaleString()`.

Каждый из унаследованных методов `toLocaleString()`, `toString()` и `valueOf()` возвращает элементы массива как строку значений, разделенных запятыми. При желании можно составить строку с другим разделителем, используя метод `join()`. Он принимает разделитель как единственный аргумент и возвращает строку, содержащую все элементы массива:

Листинг ArrayTypeJoinExample01.htm

```
var colors = ["red", "green", "blue"];
alert(colors.join(","));     // red,green,blue
alert(colors.join("||"));    // red||green||blue
```

При вызове с запятой в качестве аргумента метод `join()` повторяет вывод метода `toString()`, возвращая список значений массива `colors`, разделенных запятыми. После этого в него передаются две вертикальные черты, в результате чего выводится строка `"red||green||blue"`. Если в метод `join()` передать значение `undefined` или не передать ничего, в качестве разделителя используется запятая. В Internet Explorer 7 и более ранних версий в этом случае как разделитель ошибочно используется строка `"undefined"`.

Если элемент массива имеет значение null или undefined, в результатах методов join(), toLocaleString(), toString() и valueOf() он представляется пустой строкой.

Методы для работы с массивом как со стеком

Одной из интересных особенностей ECMAScript-массивов является то, что их можно использовать как другие структуры данных. Например, массив может работать как стек — одна из структур данных, которые ограничивают возможности добавления и удаления элементов. Стек работает по принципу *LIFO* (last-in-first-out — последним вошел, первым вышел), то есть последний добавленный элемент извлекается первым. Добавление (push) элементов в стек и *извлечение* (pop) их из стека выполняются только в одном месте: на его вершине. Специально для этих операций в ECMAScript-массивах реализованы методы `push()` и `pop()`.

Метод `push()` принимает любое количество аргументов и добавляет их в конец массива, возвращая его новую длину. Метод `pop()` извлекает последний элемент массива, уменьшает длину массива на 1 и возвращает элемент. Рассмотрим пример:

Листинг ArrayTypeExample09.htm


```
var colors = new Array();                  // создание массива
var count = colors.push("red", "green");  // включение двух элементов
alert(count);          // 2

count = colors.push("black");              // включение еще одного элемента
alert(count);          // 3

var item = colors.pop();                   // извлечение последнего элемента
alert(item);           // "black"
alert(colors.length);  // 2
```

В этом фрагменте создается массив, который затем используется как стек (обратите внимание, что специальный код для этого не требуется; `push()` и `pop()` — методы, по умолчанию доступные для массива). Сначала две строки добавляются в конец массива с помощью метода `push()`, а новая длина массива (2) присваивается переменной `count`. Затем в стек добавляется еще один элемент, а переменная `count` увеличивается до трех. Поскольку теперь в стеке три элемента, метод `push()` возвращает 3. Далее вызывается метод `pop()`, который возвращает последний элемент массива, строку `"black"`. После этого стек снова содержит два элемента.

Методы стека можно использовать вместе с любыми другими методами массива, например:

Листинг ArrayTypeExample10.htm


```
var colors = ["red", "blue"];
colors.push("brown");          // добавление элемента
colors[3] = "black";           // добавление элемента
alert(colors.length);  // 4

var item = colors.pop();       // получение последнего элемента
alert(item);                   // "black"
```

Здесь массив инициализируется двумя значениями. Затем в него добавляются еще два значения: третье — с помощью метода `push()`, а четвертое — путем непосредственного присваивания строки элементу с индексом 3. При вызове `pop()` возвращается строка `"black"`, которая была добавлена в массив последней.

Методы для работы с массивом как с очередью

Очереди ограничивают доступ к элементам порядком *FIFO* (first-in-first-out — первым вошел, первым вышел). Элементы добавляются в конец очереди и извлекаются из ее начала. Для добавления элементов в конец массива можно использовать метод `push()`, поэтому все, что нужно для имитации очереди, это способ получения первого элемента массива. Этот метод называется `shift()`. Он удаляет первый элемент массива, возвращая его и уменьшая длину массива на 1. Используя метод `shift()` вместе с `push()`, можно работать с массивом как с очередью:

Листинг ArrayTypeExample11.htm

```
var colors = new Array();                  // создание массива
var count = colors.push("red", "green");  // добавление двух элементов
alert(count);  // 2

count = colors.push("black");              // добавление еще одного элемента
alert(count);  // 3

var item = colors.shift();                 // извлечение первого элемента
alert(item);   // "red"
alert(colors.length);  // 2
```

В этом примере с помощью метода `push()` создается массив из трех цветов. В выделенной строке метод `shift()` служит для получения первого элемента массива — строки `"red"`. После его удаления в массиве остаются два элемента, при этом первое место занимает элемент `"green"`, а второе — `"black"`.

В ECMAScript у массивов есть также метод `unshift()`, обратный методу `shift()`: он добавляет любое количество элементов в начало массива и возвращает его новую длину. Используя `unshift()` вместе с `pop()`, можно имитировать обратную очередь, в которой значения добавляются в начало массива, а извлекаются с конца, например:

Листинг ArrayTypeExample12.htm

```
var colors = new Array();                    // создание массива
var count = colors.unshift("red", "green");  // добавление элементов
alert(count);  // 2

count = colors.unshift("black");  // добавление еще одного элемента
alert(count);  // 3

var item = colors.pop();          // извлечение элемента
alert(item);   // "green"
alert(colors.length);  // 2
```

В этом фрагменте созданный массив заполняется с помощью метода `unshift()`. Сначала в массив добавляются строки `"red"` и `"green"`, а затем `"black"`, в результате элементы располагаются в порядке `"black"`, `"red"`, `"green"`. При вызове метода `pop()` последний элемент (`"green"`) удаляется из массива и возвращается.

В Internet Explorer 7 и более ранних версий метод unshift() возвращает значение undefined, а не новую длину массива. В Internet Explorer 8 длина возвращается правильно, если только не выбран режим совместимости.

Методы изменения порядка следования элементов

Для изменения порядка следования элементов, уже находящихся в массиве, используются методы `reverse()` и `sort()`. Метод `reverse()` просто изменяет порядок следования элементов в массиве на обратный, например:

Листинг ArrayTypeExample13.htm


```
var values = [1, 2, 3, 4, 5];
values.reverse();
alert(values);    // 5,4,3,2,1
```

Массив `values` первоначально содержит значения 1, 2, 3, 4 и 5 в данном порядке. Вызов метода `reverse()` для массива изменяет порядок на 5, 4, 3, 2, 1. Этот метод прост, но негибок, поэтому вам также может пригодиться метод `sort()`.

По умолчанию метод `sort()` располагает элементы по возрастанию: наименьшее значение первым, а наибольшее последним. Чтобы отсортировать массив, он вызывает функцию приведения типов `String()` для каждого элемента, а затем сравнивает возвращенные строки. Это происходит, даже если массив содержит только числа, например:

Листинг ArrayTypeExample14.htm

```
var values = [0, 1, 5, 10, 15];
values.sort();
alert(values);    // 0,1,10,15,5
```

Хотя значения в этом примере сразу расположены в правильном числовом порядке, метод `sort()` сортирует их как строки. Например, строка `"10"` располагается

в итоговом массиве раньше, чем "5", хотя число 10 больше. Ясно, что во многих случаях требуется совсем не это, поэтому в метод sort() можно передать *функцию сравнения* (comparison function), которая упорядочивает два значения.

Функция сравнения принимает два аргумента и возвращает отрицательное число, если первый аргумент должен предшествовать второму, нуль, если аргументы равны, и положительное число, если первый аргумент должен следовать за вторым. Вот пример простой функции сравнения:

Листинг ArrayTypeExample15.htm


```
function compare(value1, value2) {
    if (value1 < value2) {
        return -1;
    } else if (value1 > value2) {
        return 1;
    } else {
        return 0;
    }
}
```

Эта функция сравнения работает с большинством типов данных и может использоваться как аргумент метода sort(), например:

```
var values = [0, 1, 5, 10, 15];
values.sort(compare);
alert(values);    // 0,1,5,10,15
```

При вызове метода sort() с функцией сравнения в качестве аргумента числа остаются в правильном порядке. Если поменять в ней местами возвращаемые значения, массив будет отсортирован по убыванию:

Листинг ArrayTypeExample16.htm

```
function compare(value1, value2) {
    if (value1 < value2) {
        return 1;
    } else if (value1 > value2) {
        return -1;
    } else {
        return 0;
    }
}

var values = [0, 1, 5, 10, 15];
values.sort(compare);
alert(values);    //15,10,5,1,0
```

В отличие от предыдущего примера, здесь функция сравнения возвращает 1, если первое значение меньше второго, и –1, если оно больше. Это означает, что в итоговом массиве бóльшие значения будут предшествовать меньшим, и массив будет отсортирован по убыванию. Конечно, если нужно просто изменить порядок

следования элементов на обратный, метод `reverse()` намного эффективнее сортировки.

Методы reverse() и sort() возвращают ссылку на массив, для которого они были вызваны.

Для сортировки чисел и объектов, метод `valueOf()` которых возвращает числа (например, объектов `Date`), в функции сравнения можно просто вычесть одно значение из другого:

```
function compare(value1, value2){
    return value1 - value2;
}
```

Возвращение разности двух аргументов полностью соответствует спецификации функции сравнения.

Методы манипулирования элементами

Над элементами массивов можно выполнять различные операции. Например, метод `concat()` позволяет создать новый массив на основе текущего. Сначала он создает копию массива, а затем добавляет аргументы в его конец и возвращает новый массив. Если метод `concat()` вызван без аргументов, он просто возвращает копию массива. Если передать в метод `concat()` один или несколько массивов, все их элементы будут добавлены в конец результата. Значения, которые не являются массивами, просто добавляются в конец итогового массива. Рассмотрим пример:

Листинг ArrayTypeConcatExample01.htm


```
var colors = ["red", "green", "blue"];
var colors2 = colors.concat("yellow", ["black", "brown"]);
alert(colors);     // red,green,blue
alert(colors2);    // red,green,blue,yellow,black,brown
```

Сначала массив `colors` содержит три значения. Далее для него вызывается метод `concat()`, которому передаются строка `"yellow"` и массив со значениями `"black"` и `"brown"`. В результате в массив `colors2` записываются строки `"red"`, `"green"`, `"blue"`, `"yellow"`, `"black"` и `"brown"`, а исходный массив `colors` остается неизменным.

Метод `slice()` создает массив с одним или более элементами, уже содержащимися в массиве. Он принимает один или два аргумента: начальную и конечную позиции элементов, которые нужно возвратить. Если аргумент только один, метод возвращает все элементы с этой позиции до конца массива. Если аргументов два, метод возвращает все элементы между начальной и конечной позициями, не включая конечный элемент. Эта операция никак не влияет на исходный массив. Рассмотрим пример:

Листинг ArrayTypeSliceExample01.htm


```
var colors = ["red", "green", "blue", "yellow", "purple"];
var colors2 = colors.slice(1);
var colors3 = colors.slice(1,4);

alert(colors2);    // green,blue,yellow,purple
alert(colors3);    // green,blue,yellow
```

Здесь массив `colors` содержит пять элементов. Метод `slice()` с аргументом 1 возвращает массив с четырьмя элементами без элемента `"red"`, потому что копирование начинается с позиции 1, или строки `"green"`. Итоговый массив `colors2` содержит строки `"green"`, `"blue"`, `"yellow"` и `"purple"`. Далее метод `slice()` вызывается с аргументами 1 и 4, то есть копируются элементы в позициях 1–3. В результате массив `colors3` содержит строки `"green"`, `"blue"` и `"yellow"`.

Если начальная или конечная позиция в slice() является отрицательным числом, оно вычитается из длины массива. Например, вызов slice(–2, –1) для массива с пятью элементами эквивалентен вызову slice(3, 4). Если конечная позиция меньше начальной, возвращается пустой массив.

Пожалуй, самым мощным методом для работы с массивами является `splice()`. Он используется в основном для вставки элементов в середину массива, но есть и два других способа его применения.

- **Удаление.** Из массива можно удалить любое количество элементов, указав позицию первого элемента, подлежащего удалению, и количество удаляемых элементов. Например, вызов `splice(0, 2)` удаляет первые два элемента.
- **Вставка.** Элементы можно вставить в массив в конкретной позиции, указав три или более аргументов: начальную позицию, 0 (количество удаляемых элементов) и элемент, который нужно вставить. С помощью четвертого, пятого и т. д. параметров можно вставить дополнительные элементы. Например, вызов `splice(2, 0, "red", "green»)` вставляет в массив строки `"red"` и `"green"` начиная с позиции 2.
- **Замена.** При вставке элементов в конкретной позиции можно одновременно удалить элементы, которые уже есть в массиве. Для этого нужно указать три или более аргументов: начальную позицию, количество удаляемых элементов и любое количество вставляемых элементов. Вставляемых элементов может быть больше или меньше, чем удаляемых. Например, вызов `splice(2, 1, "red", "green")` удаляет один элемент в позиции 2, а затем вставляет в этой же позиции строки `"red"` и `"green"`.

Метод `splice()` всегда возвращает массив, содержащий удаленные элементы (или пустой массив, если элементы не удалялись). Три способа его применения показаны в следующем примере:

Листинг ArrayTypeSpliceExample01.htm


```
var colors = ["red", "green", "blue"];

// удаление первого элемента
```

```
var removed = colors.splice(0,1);
alert(colors);     // green,blue
alert(removed);    // red - массив с одним элементом

// вставка двух элементов в позиции 1
removed = colors.splice(1, 0, "yellow", "orange");
alert(colors);     // green,yellow,orange,blue
alert(removed);    // пустой массив

// вставка двух значений и удаление одного
removed = colors.splice(1, 1, "red", "purple");
alert(colors);     // green,red,purple,orange,blue
alert(removed);    // yellow - массив с одним элементом
```

В начале примера массив `colors` содержит три элемента. Первый вызов `splice` удаляет первый элемент, оставляя в массиве `colors` строки `"green"` и `"blue"`. Второй вызов `splice()` вставляет два элемента в позиции 1, в результате получается массив со строками `"green"`, `"yellow"`, `"orange"` и `"blue"`. Никакие элементы при этом не удаляются, так что возвращается пустой массив. В последнем вызове `splice()` удаляется один элемент в позиции 1, а вместо него вставляются элементы `"red"` и `"purple"`. После выполнения всего кода массив `colors` будет содержать строки `"green"`, `"red"`, `"purple"`, `"orange"` и `"blue"`.

Методы поиска элементов

ECMAScript 5 предоставляет два метода поиска элементов в массивах: `indexOf()` и `lastIndexOf()`. Оба они принимают два аргумента: искомый элемент и необязательный индекс, с которого начинается поиск. Метод `indexOf()` ищет элемент с начала массива (индекс 0) до конца, а метод `lastIndexOf()` — в обратном порядке.

Оба метода возвращают позицию элемента в массиве или –1, если найти элемент не удалось. При сравнении первого аргумента с каждым элементом массива проверяется их идентичность, то есть они должны быть строго равны, как при использовании оператора `===`. Вот несколько примеров:

Листинг ArrayIndexOfExample01.htm

```
var numbers = [1,2,3,4,5,4,3,2,1];

alert(numbers.indexOf(4));          // 3
alert(numbers.lastIndexOf(4));      // 5

alert(numbers.indexOf(4, 4));       // 5
alert(numbers.lastIndexOf(4, 4));   // 3

var person = { name: "Nicholas" };
var people = [{ name: "Nicholas" }];
var morePeople = [person];

alert(people.indexOf(person));      // -1
alert(morePeople.indexOf(person));  // 0
```

Методы `indexOf()` и `lastIndexOf()` позволяют легко найти конкретные элементы в массиве. Они поддерживаются в Internet Explorer 9+, Firefox 2+, Safari 3+, Opera 9.5+ и Chrome.

Методы перебора элементов

В ECMAScript 5 определены пять методов перебора элементов массивов. Каждый из них принимает два аргумента: функцию, выполняемую для каждого элемента, и необязательный объект области, в которой будет выполняться функция (зависит от значения `this`). Функция, передаваемая в эти методы, принимает три аргумента: значение элемента массива, позицию элемента в массиве и сам объект массива. В зависимости от метода результаты выполнения этой функции могут не влиять на возвращаемое методом значение. Вот эти пять методов:

- `every()` — выполняет полученную функцию для каждого элемента массива и возвращает `true`, если она возвратила `true` для каждого элемента;
- `filter()` — выполняет полученную функцию для каждого элемента массива и возвращает массив всех элементов, для которых она возвратила `true`;
- `forEach()` — выполняет полученную функцию для каждого элемента массива, но не возвращает никакого значения;
- `map()` — выполняет полученную функцию для каждого элемента массива и возвращает массив с результатами каждого вызова функции;
- `some()` — выполняет полученную функцию для каждого элемента массива и возвращает `true`, если она возвратила `true` хотя бы для одного элемента.

Эти методы не изменяют значения, содержащиеся в массиве.

Из пяти методов наиболее похожи `every()` и `some()`, которые проверяют, соответствуют ли элементы массива некоторым условиям. Метод `every()` возвращает `true`, если переданная ему функция возвратила `true` для каждого элемента массива, в противном случае возвращается `false`. Метод `some()` возвращает `true`, если переданная ему функция возвратила `true` хотя бы для одного элемента, например:

Листинг ArrayEveryAndSomeExample01.htm


```
var numbers = [1,2,3,4,5,4,3,2,1];

var everyResult = numbers.every(function(item, index, array){
    return (item > 2);
});

alert(everyResult);    // false

var someResult = numbers.some(function(item, index, array){
    return (item > 2);
});

alert(someResult);     // true
```

В этом коде в методы `every()` и `some()` передается функция, которая возвращает `true`, если элемент больше 2. Метод `every()` возвращает `false`, потому что не все элементы соответствуют условию. Метод `some()` возвращает `true`, потому что как минимум один из элементов больше 2.

Метод `filter()` использует полученную функцию для определения элементов, которые нужно включить в возвращаемый массив. Например, следующий код возвращает массив всех чисел больше 2:

Листинг ArrayFilterExample01.htm


```
var numbers = [1,2,3,4,5,4,3,2,1];

var filterResult = numbers.filter(function(item, index, array){
    return (item > 2);
});

alert(filterResult);  // [3,4,5,4,3]
```

Здесь метод `filter()` создает и возвращает массив с элементами 3, 4, 5, 4 и 3, потому что переданная в него функция возвращает `true` для каждого из них. Этот метод полезен, если нужно запросить все элементы массива, соответствующие некоторым условиям.

Метод `map()` возвращает массив, каждый элемент которого является результатом выполнения полученной функции для соответствующего элемента исходного массива. Например, с его помощью можно умножить каждый элемент массива на 2:

Листинг ArrayMapExample01.htm

```
var numbers = [1,2,3,4,5,4,3,2,1];

var mapResult = numbers.map(function(item, index, array){
    return item * 2;
});

alert(mapResult);    // [2,4,6,8,10,8,6,4,2]
```

В этом примере возвращается массив удвоенных чисел. Метод `map()` полезен при создании массивов, элементы которых соответствуют друг другу.

Пятый метод, `forEach()`, просто выполняет полученную функцию для каждого элемента массива. Он ничего не возвращает и, по сути, аналогичен перебору массива с помощью цикла `for`, например:

```
var numbers = [1,2,3,4,5,4,3,2,1];

numbers.forEach(function(item, index, array){
    // какие-то действия
});
```

Методы перебора упрощают обработку массивов и поддерживаются в Internet Explorer 9+, Firefox 2+, Safari 3+, Opera 9.5+ и Chrome.

Методы редукции массивов

В ECMAScript 5 появились два метода редукции массивов: `reduce()` и `reduceRight()`. Оба они перебирают все элементы в массиве, формируя на их основе единственное возвращаемое значение. Метод `reduce()` делает это в направлении от первого элемента к последнему, а метод `reduceRight()` — в обратном порядке.

Оба метода принимают два аргумента: функцию, вызываемую для каждого элемента, и необязательное первоначальное значение результата редукции. Функция, передаваемая в методы `reduce()` и `reduceRight()`, принимает четыре аргумента: предыдущее значение, текущее значение, индекс элемента и объект массива. Любое значение, возвращаемое из этой функции, автоматически передается как первый аргумент в вызов функции для следующего элемента. Первая итерация выполняется для второго элемента массива, так что первым аргументом функции при этом является первый элемент массива.

Метод `reduce()` можно использовать для выполнения таких операций, как сложение всех чисел в массиве:

Листинг ArrayReductionExample01.htm


```
var values = [1,2,3,4,5];
var sum = values.reduce(function(prev, cur, index, array){
    return prev + cur;
});
alert(sum);     // 15
```

Когда функция обратного вызова выполняется в первый раз, аргументы `prev` и `cur` равны 1 и 2 соответственно. При второй итерации оба они равны 3 (`prev` как сумма 1 и 2, а `cur` как третий элемент массива). Этот процесс продолжается, пока не будут обработаны все элементы.

Метод `reduceRight()` делает то же самое, но в обратном направлении:

```
var values = [1,2,3,4,5];
var sum = values.reduceRight(function(prev, cur, index, array){
    return prev + cur;
});
alert(sum);     // 15
```

В этой версии при первом вызове функции аргументы `prev` и `cur` равны 5 и 4. Поскольку она просто складывает элементы массива, результат получается таким же, как и в предыдущем примере.

Выбор метода `reduce()` или `reduceRight()` зависит исключительно от того, в каком направлении нужно обработать элементы массива. В остальном эти методы одинаковы.

Методы редукции поддерживаются в Internet Explorer 9+, Firefox 3+, Safari 4+, Opera 10.5 и Chrome.

Тип Date

Тип `Date` в ECMAScript основан на ранней версии `java.util.Date`. Даты хранятся в нем как количество миллисекунд, прошедших с полуночи 1 января 1970 г. согласно универсальному временному коду (Universal Time Code, UTC). Благодаря такому формату с помощью типа `Date` можно точно представлять даты, отстоящие от 1 января 1970 г. на 285 616 лет.

Чтобы создать объект даты, используйте оператор `new` с конструктором `Date`:

Листинг DateTypeExample01.htm

```
var now = new Date();
```

Если конструктор `Date` вызывается без аргументов, создается объект с текущими значениями даты и времени. Чтобы создать объект `Date` с другой датой или временем, нужно передать в конструктор значение даты в миллисекундах, прошедших с полуночи 1 января 1970 г. согласно UTC. Чтобы упростить решение этой задачи, можно использовать методы `Date.parse()` и `Date.UTC()`.

Метод `Date.parse()` принимает строковое представление даты и пытается преобразовать его в дату в миллисекундах. В пятой редакции ECMA-262 сказано, что метод `Date.parse()` должен поддерживать следующие форматы данных:

- месяц/день/год (например, 6/13/2004);
- название_месяца день, год (например, January 12, 2004);
- день_недели название_месяца день год часы:минуты:секунды часовой_пояс (например, Tue May 25 2004 00:00:00 GMT-0700);
- ГГГГ-ММ-ДДТЧЧ:мм:сс.cccZ (например, 2004-05-25T00:00:00) — этот расширенный формат ISO 8601 работает только в реализациях, совместимых с ECMAScript 5.

Например, создать объект `Date` для 25 мая 2004 г. можно следующим образом:

Листинг DateTypeExample01.htm

```
var someDate = new Date(Date.parse("May 25, 2004"));
```

Если строка, переданная в метод `Date.parse()`, не представляет дату, он возвращает значение `NaN`. Если строка даты передается непосредственно в конструктор `Date`, он неявно вызывает метод `Date.parse()`, поэтому приведенный пример можно переписать так:

```
var someDate = new Date("May 25, 2004");
```

Эта инструкция идентична предыдущей.

Реализации типа Date в разных браузерах имеют много особенностей. Часто значения дат, не попадающие в допустимые диапазоны, заменяются правильными аналогами (например, некоторые браузеры интерпретируют строку «January 32, 2007» как «February 1, 2007»), тогда как Opera обычно подставляет текущий день текущего месяца, возвращая «January текущий_день, 2007». Иначе говоря, при обработке указанной даты 21 сентября будет возвращена строка «January 21, 2007».

Метод `Date.UTC()` также возвращает представление даты в миллисекундах, но создает его на основе других данных. Аргументами `Date.UTC()` являются год, месяц с отсчетом от нуля (январь — 0, февраль — 1, и т. д.), день месяца (от 1 до 31), часы (от 0 до 23), минуты, секунды и миллисекунды. Обязательны только первые два аргумента (год и месяц). Если не указан день месяца, предполагается, что он равен 1, тогда как все остальные опущенные аргументы считаются равными 0. Вот два примера использования метода `Date.UTC()`:

Листинг DateTypeUTCExample01.htm


```
// 1 января 2000 г., полночь (GMT)
var y2k = new Date(Date.UTC(2000, 0));

// 5 мая 2005 г., 17:55:55 (GMT)
var allFives = new Date(Date.UTC(2005, 4, 5, 17, 55, 55));
```

В этих примерах создаются две даты. Первая — полночь по Гринвичу 1 января 2000 г., чему соответствуют год 2000 и месяц 0 (январь). Поскольку вместо остальных аргументов подставляются значения, предлагаемые по умолчанию (1 как день месяца и нули вместо всего остального), в результате получается полночь первого дня месяца. Вторая дата представляет 5 мая 2005 г., 17:55:55 по Гринвичу. Обратите внимание, что месяц задан числом 4, потому что месяцы отсчитываются от нуля. В остальных аргументах нет ничего необычного.

Конструктор `Date` поддерживает также формат метода `Date.UTC()`, но с одним важным отличием: он создает дату и время в локальном часовом поясе, а не по Гринвичу. Аргументы в него передаются такие же, что и в метод `Date.UTC()`. Если первым аргументом является число, конструктор `Date` предполагает, что это год даты, второй аргумент — месяц, и т. д. Два предыдущих примера можно переписать следующим образом:

Листинг DateTypeConstructorExample01.htm

```
// 1 января 2000 г., полночь по локальному времени
var y2k = new Date(2000, 0);

// 5 мая 2005 г., 17:55:55 по локальному времени
var allFives = new Date(2005, 4, 5, 17, 55, 55);
```

Здесь создаются те же даты, что и в предыдущих примерах, но на этот раз они относятся к локальному часовому поясу, заданному в системных параметрах.

В ECMAScript 5 добавлен метод `Date.now()`, который возвращает дату и время его выполнения в миллисекундах. Это позволяет использовать объекты `Date` для профилирования кода:

```
// получение времени начала
var start = Date.now();

// вызов функции
doSomething();

// получение времени окончания
var stop = Date.now(),
    result = stop - start;
```

Метод `Date.now()` реализован в Internet Explorer 9+, Firefox 3+, Safari 3+, Opera 10.5 и Chrome. Для браузеров, которые его не поддерживают, можно смоделировать такое же поведение, преобразовав объекты `Date` в числа с помощью оператора сложения (`+`):

```
// получение времени начала
var start = +new Date();

// вызов функции
doSomething();

// получение времени окончания
var stop = +new Date(),
    result = stop - start;
```

Унаследованные методы

Как и другие ссылочные типы, тип `Date` переопределяет методы `toLocaleString()`, `toString()` и `valueOf()`, но у него эти методы возвращают разные значения. Метод `toLocaleString()` типа `Date` возвращает дату и время в региональном формате, заданном для браузера. Например, формат может включать обозначение AM или PM для часов и не содержать никаких сведений о часовом поясе (точный формат зависит от браузера). Метод `toString()` обычно возвращает дату и время со сведениями о часовом поясе, при этом время обычно указывается в 24-часовом формате (от 0 до 23). Далее показано, какие значения в разных браузерах возвращают методы `toLocaleString()` и `toString()` для полуночи 1 февраля 2007 г. по стандартному тихоокеанскому времени в региональном формате «en-US».

- Internet Explorer 8:

```
toLocaleString() – Thursday, February 01, 2007 12:00:00 AM
toString() – Thu Feb 1 00:00:00 PST 2007
```

- Firefox 3.5:

```
toLocaleString() – Thursday, February 01, 2007 12:00:00 AM
toString() – Thu Feb 01 2007 00:00:00 GMT-0800 (Pacific Standard Time)
```

- Safari 4:

```
toLocaleString() – Thursday, February 01, 2007 00:00:00
toString() – Thu Feb 01 2007 00:00:00 GMT-0800 (Pacific Standard Time)
```

- Chrome 4:

```
toLocaleString() – Thu Feb 01 2007 00:00:00 GMT-0800 (Pacific Standard Time)
toString() – Thu Feb 01 2007 00:00:00 GMT-0800 (Pacific Standard Time)
```

- Opera 10:

```
toLocaleString() – 2/1/2007 12:00:00 AM
toString() – Thu, 01 Feb 2007 00:00:00 GMT-0800
```

Как видите, форматы значений, возвращаемых этими методами в разных браузерах, довольно сильно различаются. По этой причине методы `toLocaleString()` и `toString()` полезны только для отладки, но не для вывода данных.

Метод `valueOf()` в типе `Date` переопределен и возвращает не строку, а представление даты в миллисекундах, чтобы операторы (такие, как «меньше» и «больше») правильно работали с датами. Рассмотрим пример:

Листинг DateTypeValueOfExample01.htm


```
var date1 = new Date(2007, 0, 1);  // 1 января 2007 г.
var date2 = new Date(2007, 1, 1);  // 1 февраля 2007 г.

alert(date1 < date2);  // true
alert(date1 > date2);  // false
```

1 января 2007 г. наступило раньше, чем 1 февраля 2007 г., поэтому вполне можно сказать, что первая дата меньше второй. Поскольку в миллисекундах 1 января 2007 г. меньше, чем 1 февраля 2007 г., оператор «меньше» возвращает при их сравнении `true`, что позволяет легко упорядочивать даты.

Методы форматирования дат

Тип `Date` включает несколько методов, с помощью которых можно форматировать даты как строки:

- `toDateString()` — выводит день недели, месяц, день месяца и год в формате, зависящем от реализации;
- `toTimeString()` — выводит часы, минуты, секунды и часовой пояс в формате, зависящем от реализации;
- `toLocaleDateString()` — выводит день недели, месяц, день месяца и год в формате, зависящем от реализации и региональных параметров;

- `toLocaleTimeString()` — выводит часы, минуты и секунды в формате, зависящем от реализации;
- `toUTCString()` — выводит полную UTC-дату в формате, зависящем от реализации.

Как и в случае методов `toLocaleString()` и `toString()`, выводимые этими методами данные во многом зависят от браузера, поэтому их не следует использовать для вывода дат в пользовательском интерфейсе.

Есть также метод toGMTString(), который эквивалентен методу toUTCString() и предоставляется ради обратной совместимости. В спецификации рекомендуется использовать в новом коде только метод toUTCString().

Методы для работы с компонентами даты/времени

Остальные методы типа `Date`, приведенные в таблице, получают и задают отдельные части даты и времени. Если в таблице упоминается стандарт UTC, это означает, что дата интерпретируется без смещения часового пояса (преобразуется в дату по Гринвичу).

Метод	Описание
getTime()	Возвращает представление даты в миллисекундах; то же, что и valueOf()
setTime(миллисекунды)	Задает представление даты в миллисекундах, изменяя тем самым всю дату
getFullYear()	Возвращает четырехзначный год (2007, а не просто 07)
getUTCFullYear()	Возвращает четырехзначный год даты в формате UTC
setFullYear(год)	Задает год даты. Значение года должно содержать четыре цифры (2007, а не просто 07)
setUTCFullYear(год)	Задает год даты в формате UTC. Значение года должно содержать четыре цифры (2007, а не просто 07)
getMonth()	Возвращает месяц даты (0 представляет январь, а 11 — декабрь)
getUTCMonth()	Возвращает месяц даты в формате UTC (0 представляет январь, а 11 — декабрь)
setMonth(месяц)	Задает месяц даты с отсчетом от 0. Указание числа больше 11 приводит к увеличению года
setUTCMonth(месяц)	Задает месяц даты в формате UTC с отсчетом от 0. Указание числа больше 11 приводит к увеличению года
getDate()	Возвращает день месяца даты (от 1 до 31)
getUTCDate()	Возвращает день месяца даты в формате UTC (от 1 до 31)

Метод	Описание
setDate(дата)	Задает день месяца для даты. Если значение даты больше, чем количество дней в месяце, значение месяца увеличивается
setUTCDate(дата)	Задает день месяца для даты в формате UTC. Если значение даты больше, чем количество дней в месяце, значение месяца увеличивается
getDay()	Возвращает день недели даты как число (0 представляет воскресенье, 6 — субботу)
getUTCDay()	Возвращает день недели даты в формате UTC как число (0 представляет воскресенье, 6 — субботу)
getHours()	Возвращает час даты как число от 0 до 23
getUTCHours()	Возвращает час даты в формате UTC как число от 0 до 23
setHours(часы)	Задает час даты. Указание числа больше 23 увеличивает день месяца
setUTCHours(часы)	Задает час даты в формате UTC. Указание числа больше 23 увеличивает день месяца
getMinutes()	Возвращает минуты даты как число от 0 до 59
getUTCMinutes()	Возвращает минуты даты в формате UTC как число от 0 до 59
setMinutes(минуты)	Задает минуты даты. Указание числа больше 59 увеличивает час
setUTCMinutes(минуты)	Задает минуты даты в формате UTC. Указание числа больше 59 увеличивает час
getSeconds()	Возвращает секунды даты как число от 0 до 59
getUTCSeconds()	Возвращает секунды даты в формате UTC как число от 0 до 59
setSeconds(секунды)	Задает секунды даты. Указание числа больше 59 увеличивает значение минут
setUTCSeconds(секунды)	Задает секунды даты в формате UTC. Указание числа больше 59 увеличивает значение минут
getMilliseconds()	Возвращает миллисекунды даты
getUTCMilliseconds()	Возвращает миллисекунды даты в формате UTC
setMilliseconds (миллисекунды)	Здает миллисекунды даты
setUTCMilliseconds (миллисекунды)	Задает миллисекунды даты в формате UTC
getTimeZoneOffset()	Возвращает количество минут, на которое локальный часовой пояс отстоит от UTC. Например, для восточного стандартного времени возвращается число 300. При переходе региона на летнее время это значение меняется

Тип RegExp

Тип `RegExp` реализует в ECMAScript регулярные выражения, которые можно с легкостью создавать, используя синтаксис, похожий на Perl:

```
var выражение = /шаблон/флаги;
```

Шаблоном может быть регулярное выражение любой сложности, включающее классы символов, квантификаторы, группировки, предпросмотр и обратные ссылки. Каждое выражение может иметь или не иметь флаги, указывающие режим сопоставления. Поддерживаются следующие флаги:

- `g` — включает глобальный режим, в котором шаблон применяется ко всей строке, то есть поиск не прекращается после обнаружения первого совпадения;
- `i` — включает режим без учета регистра, в котором при поиске совпадений регистры шаблона и строки игнорируются;
- `m` — включает многострочный режим, в котором поиск совпадений продолжается после достижения конца одной строки теста.

Регулярное выражение создается путем объединения шаблона и флагов, например:

```
/*
 * Поиск всех экземпляров "at" в строке.
 */
var pattern1 = /at/g;

/*
 * Поиск первого экземпляра "bat" или "cat" без учета регистра.
 */
var pattern2 = /[bc]at/i;

/*
 * Поиск всех трехсимвольных сочетаний, заканчивающихся на "at",
 * независимо от регистра.
 */
var pattern3 = /.at/gi;
```

Как и в других языках, все *метасимволы* (metacharacters) в шаблоне регулярного выражения нужно экранировать. Доступны следующие метасимволы:

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

Метасимволы могут использоваться в регулярных выражениях несколькими способами, поэтому если требуется сопоставить метасимвол со строкой, его нужно экранировать обратной косой чертой, например:

```
/*
 * Поиск первого экземпляра "bat" или "cat" без учета регистра.
 */
```

```
var pattern1 = /[bc]at/i;

/*
 * Поиск первого экземпляра "[bc]at" без учета регистра.
 */
var pattern2 = /\[bc\]at/i;

/*
 * Поиск всех трехсимвольных сочетаний, заканчивающихся на "at",
 * без учета регистра.
 */
var pattern3 = /.at/gi;

/*
 * Поиск всех экземпляров ".at" без учета регистра.
 */
var pattern4 = /\.at/gi;
```

В этом коде `pattern1` сопоставляется со всеми экземплярами `"bat"` и `"cat"` независимо от регистра. Чтобы выполнить сопоставление со строкой `"[bc]at"`, нужно экранировать обе квадратные скобки, как сделано в шаблоне `pattern2`. В шаблоне `pattern3` точка указывает, что символам `"at"` может предшествовать любой символ. Если требуется найти подстроку `".at"`, нужно экранировать точку, как в шаблоне `pattern4`.

Во всех предыдущих примерах регулярные выражения определены как литералы, но их также можно создавать с помощью конструктора `RegExp`. Он принимает два аргумента: строковый шаблон для сопоставления и необязательную строку флагов. Любое регулярное выражение, которое можно определить, используя синтаксис литералов, можно также создать с помощью конструктора, например:

```
/*
 * Поиск первого экземпляра "bat" или "cat" без учета регистра.
 */
var pattern1 = /[bc]at/i;

/*
 * То же, что и pattern1, но с использованием конструктора.
 */
var pattern2 = new RegExp("[bc]at", "i");
```

Здесь `pattern1` и `pattern2` определяют эквивалентные регулярные выражения. Обратите внимание, что оба аргумента конструктора `RegExp` являются строками (литералы регулярных выражений не следует передавать в конструктор `RegExp`). Поскольку шаблон передается в конструктор `RegExp` как строка, иногда его знаки могут требовать двойного экранирования. Это относится ко всем метасимволам, а также к знакам, которые уже экранированы, таким как `\n` (знак `\`, который обычно экранируется в строках как `\\`, повторяется в строке регулярного выражения четырежды: `\\\\`). В следующей таблице приведены некоторые шаблоны в форме литералов и эквивалентные строки для передачи в конструктор `RegExp`:

Литерал шаблона	Строковый эквивалент
/\[bc\]at/	"\\[bc\\]at"
/\.at/	"\\.at"
/name\/age/	"name\\/age"
/\d.\d{1,2}/	"\\d.\\d{1,2}"
/\w\\hello\\123/	"\\w\\\\hello\\\\123"

Помните, что определить регулярное выражение на основе литерала и создать его с помощью конструктора `RegExp` — это не совсем одно и то же. В ECMAScript 3 литералы регулярных выражений всегда относятся к одному экземпляру `RegExp`, тогда как вызов конструктора типа `RegExp` каждый раз создает его новый экземпляр. Рассмотрим следующий пример:

```
var re = null,
    i;

for (i=0; i < 10; i++){
    re = /cat/g;
    re.test("catastrophe");
}

for (i=0; i < 10; i++){
    re = new RegExp("cat", "g");
    re.test("catastrophe");
}
```

В первом цикле создается только один экземпляр `RegExp` для шаблона `/cat/`, несмотря на то, что он указан в теле цикла. Свойства экземпляра (см. следующий раздел) не сбрасываются, из-за чего метод `test()` каждый второй раз не может обнаружить шаблон `/cat/` в строке. При первом вызове `test()` шаблон обнаруживается, но в следующей итерации поиск начинается с индекса 3 (конец первого совпадения) и завершается ничем. После достижения конца строки и перехода к следующей итерации метод `test()` снова начинает поиск с начала строки.

Во втором цикле используется конструктор `RegExp`, который на каждой итерации создает новое регулярное выражение, поэтому каждый вызов метода `test()` возвращает `true`.

Уточнение, внесенное в ECMAScript 5, явно требует, чтобы для литералов регулярных выражений создавались новые экземпляры `RegExp`, как если бы непосредственно вызывался конструктор `RegExp`. Это изменение реализовано в Internet Explorer 9+, Firefox 4+ и Chrome.

Свойства экземпляра RegExp

У каждого экземпляра `RegExp` есть следующие свойства, позволяющие получить сведения о шаблоне:

- `global` — логическое значение, указывающее, задан ли флаг `g`;
- `ignoreCase` — логическое значение, указывающее, задан ли флаг `i`;
- `lastIndex` — целое число, указывающее позицию в исходной строке, где сопоставление будет выполнено в следующий раз (это значение всегда первоначально равно 0);
- `multiline` — логическое значение, указывающее, задан ли флаг `m`;
- `source` — исходная строка регулярного выражения, которая всегда возвращается в форме литерала (без открывающей и закрывающей косых черт), а не как строковый шаблон, переданный в конструктор.

С помощью этих свойств можно получить полезную информацию о регулярном выражении, но они используются редко, потому что эта информация доступна в объявлении шаблона, например:

Листинг RegExpInstancePropertiesExample01.htm


```
var pattern1 = /\[bc\]at/i;

alert(pattern1.global);        // false
alert(pattern1.ignoreCase);    // true
alert(pattern1.multiline);     // false
alert(pattern1.lastIndex);     // 0
alert(pattern1.source);        // "\[bc\]at"

var pattern2 = new RegExp("\\[bc\\]at", "i");

alert(pattern2.global);        // false
alert(pattern2.ignoreCase);    // true
alert(pattern2.multiline);     // false
alert(pattern2.lastIndex);     // 0
alert(pattern2.source);        // "\[bc\]at"
```

Заметьте, что значения свойства `source` у обоих шаблонов одинаковы, хотя первый представлен в формате литерала, а второй был передан в конструктор `RegExp`. Свойство `source` форматирует строку как литерал.

Методы экземпляра RegExp

Главный метод объекта `RegExp` называется `exec()` и предназначен для работы с группами захвата. Он принимает в качестве единственного аргумента строку, к которой нужно применить шаблон, и возвращает массив со сведениями о первом совпадении или значение `null`, если совпадения отсутствуют. Возвращенный массив является экземпляром `Array`, но содержит два дополнительных свойства: `index` — место в строке, где было зарегистрировано совпадение с шаблоном, и `input` — исходная строка для сопоставления с шаблоном. Первым элементом массива является строка, соответствующая всему шаблону, а любые дополнительные элементы представляют захваченные группы в выражении (если в шаблоне нет групп захвата, массив содержит только один элемент). Рассмотрим следующий пример:

Листинг RegExpExecExample01.htm


```
var text = "mom and dad and baby";
var pattern = /mom( and dad( and baby)?)?/gi;

var matches = pattern.exec(text);
alert(matches.index);    // 0
alert(matches.input);    // "mom and dad and baby"
alert(matches[0]);       // "mom and dad and baby"
alert(matches[1]);       // " and dad and baby"
alert(matches[2]);       // " and baby"
```

В этом примере у шаблона две группы захвата. Внутренняя сопоставляется со строкой `" and baby"`, а охватывающая ее — со строкой `" and dad"` или `" and dad and baby"`. При вызове метода `exec()` для строки обнаруживается совпадение. Поскольку шаблону соответствует вся строка, свойству `index` массива `matches` присваивается значение 0. В первом элементе массива сохраняется вся сопоставленная строка, во втором — содержимое первой группы захвата, в третьем — содержимое второй группы захвата.

Метод `exec()` возвращает сведения об одном совпадении за раз, даже если шаблон глобален. Если флаг глобального поиска не указан, многократные вызовы метода `exec()` для одной и той же строки всегда возвращают сведения о первом совпадении. Если же флаг `g` задан, при каждом вызове `exec()` позиция поиска совпадения перемещается вперед по строке, например:

Листинг RegExpExecExample02.htm

```
var text = "cat, bat, sat, fat";
var pattern1 = /.at/;
var matches = pattern1.exec(text);

alert(matches.index);         // 0
alert(matches[0]);            // cat
alert(pattern1.lastIndex);    // 0

matches = pattern1.exec(text);
alert(matches.index);         // 0
alert(matches[0]);            // cat
alert(pattern1.lastIndex);    // 0

var pattern2 = /.at/g;

var matches = pattern2.exec(text);
alert(matches.index);         // 0
alert(matches[0]);            // cat
alert(pattern2.lastIndex);    // 0

matches = pattern2.exec(text);
alert(matches.index);         // 5
alert(matches[0]);            // bat
alert(pattern2.lastIndex);    // 8
```

Первый шаблон в этом примере, `pattern1`, не является глобальным, поэтому каждый вызов `exec()` возвращает только первое совпадение (`"cat"`). Второй шаблон, `pattern2`, глобален, так что при каждом вызове `exec()` возвращается следующее совпадение в строке, пока она не заканчивается. Выбор режима также влияет на свойство `lastIndex` шаблона. В режиме глобального сопоставления значение `lastIndex` увеличивается после каждого вызова `exec()`, а в обычном режиме оно остается неизменным.

В Internet Explorer значение lastIndex обновляется даже в неглобальном режиме.

Если требуется выяснить, соответствует ли строка шаблону, но текст совпадения не нужен, можно использовать метод `test()`. Он принимает строковый аргумент и возвращает `true`, если шаблон соответствует аргументу, или `false` в противном случае. Метод `test()` часто используется в инструкциях `if`, например:

```
var text = "000-00-0000";
var pattern = /\d{3}-\d{2}-\d{4}/;

if (pattern.test(text)){
    alert("The pattern was matched.");  // Обнаружено соответствие шаблону
}
```

В этом примере регулярное выражение сопоставляется с последовательностью цифр. Если входной текст соответствует шаблону, выводится сообщение. Подобный код часто применяется для проверки введенных пользователем данных, когда вас интересует только то, допустимы эти данные или нет.

Унаследованные методы `toLocaleString()` и `toString()` возвращают литерал регулярного выражения независимо от того, как оно было создано, например:

Листинг RegExpToStringExample01.htm


```
var pattern = new RegExp("\\[bc\\]at", "gi");
alert(pattern.toString());          // /\[bc\]at/gi
alert(pattern.toLocaleString());    // /\[bc\]at/gi
```

Хотя шаблон в этом примере создается с помощью конструктора `RegExp`, методы `toLocaleString()` и `toString()` возвращают его, как если бы он был задан в формате литерала.

Метод valueOf() типа RegExp возвращает само регулярное выражение.

Свойства конструктора RegExp

У функции конструктора `RegExp` есть несколько свойств, указанных в приведенной далее таблице (в других языках они были бы статическими). Они применяются ко всем регулярным выражениям в области видимости и изменяются согласно

последней операции с регулярным выражением. Эти свойства уникальны еще тем, что есть два способа доступа к ним: по полному и по сокращенному имени (Opera не поддерживает сокращенные имена).

Полное имя	Сокращенное имя	Описание
input	$_	Последняя строка, для которой выполнялось сопоставление. Это свойство не реализовано в Opera
lastMatch	$&	Последний совпавший текст. Это свойство не реализовано в Opera
lastParen	$+	Последняя совпавшая группа захвата. Это свойство не реализовано в Opera
leftContext	$`	Текст в строке input перед lastMatch
multiline	$*	Логическое значение, указывающее, должен ли использоваться многострочный режим со всеми выражениями. Это свойство не реализовано в IE и Opera
rightContext	$'	Текст в строке input после lastMatch

Эти свойства можно использовать для извлечения сведений об операции, выполненной методом `exec()` или `test()`, например:

Листинг RegExpConstructorPropertiesExample01.htm


```
var text = "this has been a short summer";
var pattern = /(.)hort/g;

/*
 * Примечание: Opera не поддерживает свойства input,
 * lastMatch, lastParen и multiline.
 * Internet Explorer не поддерживает свойство multiline.
 */
if (pattern.test(text)){
    alert(RegExp.input);          // this has been a short summer
    alert(RegExp.leftContext);    // this has been a
    alert(RegExp.rightContext);   // summer
    alert(RegExp.lastMatch);      // short
    alert(RegExp.lastParen);      // s
    alert(RegExp.multiline);      // false
}
```

В этом коде создается шаблон, который ищет любой знак, предшествующий строке `"hort"`, и определяет группу захвата для первой буквы. Со свойствами при этом происходит следующее:

- свойство `input` содержит исходную строку;
- свойство `leftContext` содержит символы строки до слова `"short"`, а свойство `rightContext` — после слова `"short"`;

- свойство `lastMatch` содержит последнюю строку, которая соответствует всему регулярному выражению, или `"short"`;
- свойство `lastParen` содержит последнюю совпавшую группу захвата, или `"s"` в данном случае.

Полные имена свойств можно заменить сокращенными, но для доступа к ним нужно использовать квадратные скобки, потому что большинство из них не является допустимыми ECMAScript-идентификаторами:

Листинг RegExpConstructorPropertiesExample02.htm


```
var text = "this has been a short summer";
var pattern = /(.)hort/g;

/*
 * Примечание: Opera не поддерживает сокращенные имена свойств.
 * Internet Explorer не поддерживает свойство multiline.
 */
if (pattern.test(text)){
    alert(RegExp.$_);         // this has been a short summer
    alert(RegExp["$`"]);      // this has been a
    alert(RegExp["$'"]);      // summer
    alert(RegExp["$&"]);      // short
    alert(RegExp["$+"]);      // s
    alert(RegExp["$*"]);      // false
}
```

У конструктора также есть свойства, хранящие до девяти совпадений с группами захвата. Они имеют имена с `RegExp.$1` (первое совпадение) по `RegExp.$9` (девятое совпадение). Эти свойства заполняются при вызове метода `exec()` или `test()`, что позволяет написать такой код:

Листинг RegExpConstructorPropertiesExample03.htm


```
var text = "this has been a short summer";
var pattern = /(..)or(.)/g;

if (pattern.test(text)){
    alert(RegExp.$1);     // sh
    alert(RegExp.$2);     // t
}
```

В этом примере создается шаблон с двумя группами сопоставления, который затем применяется к строке. Хотя метод `test()` просто возвращает логическое значение, при этом также заполняются свойства `$1` и `$2` конструктора `RegExp`.

Ограничения шаблонов

В целом поддержка регулярных выражений в языке ECMAScript очень хороша, но в нем нет некоторых нетривиальных возможностей, доступных в таких языках,

как Perl. Например, следующие функциональные возможности в ECMAScript не поддерживаются (дополнительные сведения см. на сайте www.regularexpressions.info):

- якоря `\A` и `\Z` (начало и конец строки соответственно);
- обратный просмотр;
- классы объединения и пересечения;
- атомарные группировки;
- поддержка Юникода (исключая сопоставление с одним символом за раз);
- именованные группы захвата;
- режимы сопоставления `s` (однострочный) и `x` (с пробельными символами);
- условия;
- комментарии в регулярных выражениях.

Несмотря на эти ограничения, имеющихся в ECMAScript возможностей достаточно для решения большинства задач сопоставления с шаблонами.

Тип Function

Функции входят в число самых интересных ECMAScript-элементов, в основном потому, что на самом деле это объекты. Каждая функция представляет собой экземпляр типа `Function`, у которого есть свойства и методы, как и у любого другого ссылочного типа. Поскольку функции являются объектами, их имена — это просто указатели на объекты функций и поэтому они могут быть не связаны с самими функциями. Для определения функций обычно используется синтаксис объявления функции, например:

```
function sum (num1, num2) {
    return num1 + num2;
}
```

Этот синтаксис почти не отличается от выражения функции, например:

```
var sum = function(num1, num2){
    return num1 + num2;
};
```

Здесь переменная `sum` определяется и инициализируется как функция. После ключевого слова `function` имя не требуется: на функцию можно ссылаться через переменную `sum`. Обратите также внимание, что код заканчивается точкой с запятой, как и при инициализации переменной.

Последний способ определения функции — это вызов конструктора `Function`, который может принимать любое количество аргументов. Его последний аргумент всегда является телом функции, а предыдущие — ее аргументами, например:

```
// не рекомендуется
var sum = new Function("num1", "num2", "return num1 + num2");
```

Использовать этот синтаксис не рекомендуется, поскольку он снижает быстродействие из-за двойной интерпретации кода (в первый раз — обычного ECMAScript-кода, во второй — строк, которые передаются в конструктор). Тем не менее он полезен тем, что помогает думать о функциях и их именах как об объектах и указателях.

Поскольку имена функций — это просто указатели на них, они работают, как и любые другие переменные, содержащие указатели на объекты. Это означает, что у функции может быть несколько имен, например:

Листинг FunctionTypeExample01.htm


```
function sum(num1, num2){
    return num1 + num2;
}
alert(sum(10,10));          // 20

var anotherSum = sum;
alert(anotherSum(10,10));   // 20

sum = null;
alert(anotherSum(10,10));   // 20
```

Здесь определяется функция `sum()`, которая складывает два числа. Затем объявляется переменная `anotherSum`, которой назначается функция `sum`. Имя функции при этом указывается без скобок, поэтому оно интерпретируется как указатель на функцию, а не как ее вызов. В этот момент `anotherSum` и `sum` указывают на одну и ту же функцию, то есть можно вызвать функцию `anotherSum()` и получить тот же результат. Когда переменной `sum` присваивается значение `null`, ее связь с функцией разрывается, а вызовы `anotherSum()` по-прежнему работают.

Никакой перегрузки (новый взгляд)

Отождествление имен функций и указателей также помогает понять, почему в ECMAScript невозможна перегрузка функций. Давайте вернемся к примеру из главы 3:

```
function addSomeNumber(num) {
    return num + 100;
}

function addSomeNumber(num) {
    return num + 200;
}

var result = addSomeNumber(100);  // 300
```

Как видно, при объявлении двух функций с одним и тем же именем последняя из них перезаписывает первую. Этот код почти эквивалентен следующему:

```
var addSomeNumber = function (num) {
    return num + 100;
};

addSomeNumber = function (num) {
    return num + 200;
};

var result = addSomeNumber(100);  // 300
```

Новый код гораздо нагляднее показывает, что происходит: при создании второй функции переменная `addSomeNumber` просто перезаписывается.

Объявления функций и функции-выражения

В этом разделе понятия «объявление функции» и «функции-выражение» используются почти как синонимы, но между ними есть одно важное различие, связанное с тем, как интерпретатор JavaScript загружает данные в контекст выполнения. Объявления функций читаются и становятся доступны в контексте выполнения до интерпретации любого кода, а функции-выражения обрабатываются только по их достижении, например:

Листинг FunctionDeclarationExample01.htm


```
alert(sum(10,10));
function sum(num1, num2){
    return num1 + num2;
}
```

Этот пример работает без ошибок потому, что объявления функций заблаговременно добавляются в контекст выполнения в ходе так называемого *подъема объявлений функций* (function declaration hoisting). При первой обработке кода интерпретатор JavaScript находит объявления функций и выводит их на вершину иерархии. Даже если в исходном коде объявление функции расположено после ее вызова, интерпретатор *поднимает* объявление. Если изменить объявление функции на эквивалентную функцию-выражение, во время выполнения кода возникнет ошибка:

Листинг FunctionInitializationExample01.htm


```
alert(sum(10,10));
var sum = function(num1, num2){
    return num1 + num2;
};
```

Ошибка возникает потому, что на этот раз функция не объявляется, а входит в инструкцию инициализации. Это означает, что она недоступна через переменную `sum`, пока не будет выполнена выделенная строка кода, чего так и не происходит, потому что первая строка вызывает ошибку `unexpected identifier` (неожиданный идентификатор).

В остальном два варианта синтаксиса эквивалентны.

Существуют также именованные функции-выражения, выглядящие как объявления, например var sum = function sum() {}. Более подробно функции-выражения обсуждаются в главе 7.

Функции как значения

Поскольку имена функций в ECMAScript — это обычные переменные, функции можно использовать везде, где можно задействовать любые другие значения. Это означает, что функции могут быть аргументами и возвращаемыми значениями других функций, например:

```
function callSomeFunction(someFunction, someArgument){
    return someFunction(someArgument);
}
```

Эта функция принимает два аргумента. Первым из них является функция, а вторым — передаваемое ей значение. Использовать эту функцию можно следующим образом:

Листинг FunctionAsAnArgumentExample01.htm

```
function add10(num){
    return num + 10;
}

var result1 = callSomeFunction(add10, 10);
alert(result1);    // 20

function getGreeting(name){
    return "Hello, " + name;
}

var result2 = callSomeFunction(getGreeting, "Nicholas");
alert(result2);    // "Hello, Nicholas"
```

Функция `callSomeFunction()` универсальна, поэтому какая бы функция ни была передана ей в качестве аргумента, результат всегда возвращается из этой функции-аргумента. Если помните, имя функции без скобок представляет собой указатель на функцию, а не ее вызов, так что в `callSomeFunction()` передаются переменные `add10` и `getGreeting`, а не результаты вызова этих функций.

Возвращение функций из функций — еще одна очень полезная возможность. Предположим, например, что нам нужно отсортировать массив объектов по их произвольному свойству. Функция сравнения, передаваемая в метод `sort()`, принимает только два аргумента, сравниваемые значения, но мы должны как-то указать, по какому свойству требуется отсортировать массив. Для этого можно определить функцию, создающую функцию сравнения на основе имени свойства.

Листинг FunctionReturningFunctionExample01.htm


```
function createComparisonFunction(propertyName) {

    return function(object1, object2){
        var value1 = object1[propertyName];
        var value2 = object2[propertyName];

        if (value1 < value2){
            return -1;
        } else if (value1 > value2){
            return 1;
        } else {
            return 0;
        }
    };
}
```

Синтаксис этого примера может показаться сложным, но по сути это просто функция внутри функции с оператором `return` в начале. Аргумент `propertyName` используется во внутренней функции со скобочной нотацией для получения значений указанного свойства. Как только значения получены, они сравниваются. Эту функцию можно использовать следующим образом:

```
var data = [{name: "Zachary", age: 28}, {name: "Nicholas", age: 29}];

data.sort(createComparisonFunction("name"));
alert(data[0].name);  // Nicholas

data.sort(createComparisonFunction("age"));
alert(data[0].name);  // Zachary
```

В этом коде создается массив `data` с двумя объектами, у каждого из которых есть свойства `name` и `age`. По умолчанию метод `sort()` для определения порядка сортировки вызвал бы для каждого объекта метод `toString()`, что в данном случае дало бы нелогичный результат. Вызов `createComparisonFunction("name")` создает функцию сравнения, которая выполняет сортировку по свойству `name`, а это означает, что первый элемент будет иметь имя `"Nicholas"` и возраст 29 лет. При вызове функции `createComparisonFunction("age")` она создает функцию сравнения, которая выполняет сортировку по свойству `age`, при этом первым становится пользователь с именем `"Zachary"` и возрастом 28 лет.

Внутри функций

Внутри функции доступны два специальных объекта: `arguments` и `this`. Как отмечено в главе 3, объект `arguments` похож на массив и содержит все аргументы, переданные в функцию. Хотя он используется преимущественно для доступа к аргументам функции, у него есть свойство `callee`, представляющее собой указатель на функцию,

которой принадлежит этот объект arguments. Рассмотрим следующую классическую функцию вычисления факториала:

```
function factorial(num){
    if (num <= 1) {
        return 1;
    } else {
        return num * factorial(num-1)
    }
}
```

Функции вычисления факториала обычно рекурсивны, как в этом примере, который нормально работает, если имя функции не изменяется. Однако правильная работа этой функции зависит от имени "factorial". Эту зависимость можно разорвать, используя свойство arguments.callee:

Листинг FunctionTypeArgumentsExample01.htm


```
function factorial(num){
    if (num <= 1) {
        return 1;
    } else {
        return num * arguments.callee(num-1)
    }
}
```

В новой версии больше нет ссылки на имя "factorial" в теле функции, что гарантирует правильную работу рекурсивного алгоритма независимо от того, по какому имени вызывается функция. Рассмотрим следующий пример:

```
var trueFactorial = factorial;

factorial = function(){
    return 0;
};

alert(trueFactorial(5));  // 120
alert(factorial(5));      // 0
```

Здесь переменной trueFactorial присваивается значение factorial, при этом указатель на функцию дублируется. Затем переменной factorial назначается другая функция, которая просто возвращает 0. Если бы в теле оригинальной функции factorial() не использовалось свойство arguments.callee, вызов trueFactorial() возвратил бы 0. Однако благодаря отсутствию зависимости между функцией и ее именем функция trueFactorial() вычисляет факториал правильно, а 0 возвращает только функция factorial().

Другой специальный объект, this, похож на объект this в Java и C#, но имеет некоторые отличия. Он ссылается на объект контекста, в котором выполняется

функция (если функция вызвана в глобальной области видимости веб-страницы, объект this указывает на window). Рассмотрим пример:

Листинг FunctionTypeThisExample01.htm


```
window.color = "red";
var o = { color: "blue" };

function sayColor(){
    alert(this.color);
}

sayColor();    // "red"

o.sayColor = sayColor;
o.sayColor();  // "blue"
```

Функция sayColor() здесь определена глобально, но ссылается на объект this. Значение this не определяется вплоть до вызова функции, поэтому при выполнении кода оно может изменяться. Когда функция sayColor() вызывается в глобальной области видимости, она выводит строку "red", потому что this указывает на объект window, то есть значение this.color эквивалентно window.color. Когда после назначения функции объекту o вызывается функция o.sayColor(), объект this указывает на o и свойство this.color интерпретируется как o.color, в результате выводится строка "blue".

Помните, что имена функций — это просто переменные, содержащие указатели, поэтому вызов sayColor() в глобальной области видимости и вызов o.sayColor() соответствуют одной и той же функции, хотя и выполняются в разных контекстах.

В ECMAScript 5 определено еще одно свойство объекта функции, а именно caller. Хотя этого свойства нет в спецификации ECMAScript 3, его поддерживают все браузеры, кроме ранних версий Opera. Оно содержит ссылку на функцию, которая вызвала текущую функцию, или имеет значение null, если функция вызвана из глобальной области видимости, например:

Листинг FunctionTypeArgumentsCallerExample01.htm

```
function outer(){
    inner();
}

function inner(){
    alert(inner.caller);
}

outer();
```

При выполнении этого фрагмента выводится оповещение с исходным кодом функции outer(). Поскольку она вызывает функцию inner(), свойство inner.caller

указывает на `outer()`. Чтобы ослабить связь между функциями, можно сделать то же самое с помощью свойства `arguments.callee.caller`:

Листинг FunctionTypeArgumentsCallerExample02.htm


```
function outer(){
    inner();
}

function inner(){
    alert(arguments.callee.caller);
}

outer();
```

Свойство `caller` поддерживается во всех версиях Internet Explorer, Firefox, Chrome и Safari, а также в Opera 9.6.

При выполнении кода в строгом режиме попытка доступа к свойству `arguments.callee` приводит к ошибке. Свойство `arguments.caller` из ECMAScript 5 также вызывает ошибку в строгом режиме и всегда не определено в других режимах. Это сделано для того, чтобы нельзя было спутать свойство `arguments.caller` со свойством `caller` функции. Эти изменения внесены в язык для обеспечения безопасности, чтобы сторонний код не мог инспектировать другой код, выполняемый в том же контексте.

Кроме того, в строгом режиме нельзя присвоить значение свойству `caller` функции. Попытка сделать это приведет к ошибке.

Свойства и методы функций

Поскольку в ECMAScript функции являются объектами, у них есть свойства и методы. У каждой функции имеются свойства `length` и `prototype`. Свойство `length` указывает количество именованных аргументов функции, например:

Листинг FunctionTypeLengthPropertyExample01.htm

```
function sayName(name){
    alert(name);
}

function sum(num1, num2){
    return num1 + num2;
}

function sayHi(){
    alert("Hello");
}

alert(sayName.length);  // 1
alert(sum.length);      // 2
alert(sayHi.length);    // 0
```

В этом фрагменте определены три функции, принимающие разное количество именованных аргументов. Функция `sayName()` принимает один аргумент, поэтому ее свойство `length` имеет значение 1. У функции `sum()` аргументов два, поэтому у нее значение свойства `length` равно двум, а у функции `sayHi()` именованных аргументов нет и значение `length` равно нулю.

Свойство `prototype`, пожалуй, наиболее интересная часть ECMAScript-ядра. В нем располагаются все методы экземпляров ссылочных типов, то есть методы вроде `toString()` и `valueOf()` на самом деле определены в свойстве `prototype`, а экземпляры объектов получают доступ к ним. Это свойство очень важно при определении собственных ссылочных типов и при наследовании (эти темы рассматриваются в главе 6). В ECMAScript 5 свойство `prototype` неперечислимо, а значит, не обнаруживается с помощью `for-in`.

У функций также есть методы `apply()` и `call()`. Оба они вызывают функцию с конкретным аргументом `this`, задавая значение объекта `this` в теле функции. Метод `apply()` принимает два аргумента: значение `this` внутри функции и массив аргументов, который может быть экземпляром `Array` или объектом `arguments`, например:

Листинг FunctionTypeApplyMethodExample01.htm


```
function sum(num1, num2){
    return num1 + num2;
}

function callSum1(num1, num2){
    return sum.apply(this, arguments);     // передача объекта arguments
}

function callSum2(num1, num2){
    return sum.apply(this, [num1, num2]);  // передача массива
}

alert(callSum1(10,10));    // 20
alert(callSum2(10,10));    // 20
```

В этом примере метод `callSum1()` выполняет `sum()`, передавая в него объект `this` (который эквивалентен объекту `window`, потому что метод вызывается в глобальной области видимости) и объект `arguments`. Метод `callSum2()` также вызывает `sum()`, но передает в него вместо объекта `arguments` массив аргументов. Оба метода возвращают правильный результат.

В строгом режиме тип значения this не приводится к window, если функция вызвана без объекта контекста. Вместо этого this получает значение undefined, если не задать его явно, связав функцию с объектом или вызвав метод apply() или call().

Метод `call()` делает то же самое, что и `apply()`, но аргументы передаются в него иначе. Первым аргументом также является значение `this`, но остальные аргументы передаются непосредственно в функцию. При вызове `call()` аргументы необходимо указывать явно, например:

Листинг FunctionTypeCallMethodExample01.htm


```
function sum(num1, num2){
    return num1 + num2;
}

function callSum(num1, num2){
    return sum.call(this, num1, num2);
}

alert(callSum(10,10));    // 20
```

Вызывая метод `call()` в методе `callSum()`, нужно передать каждый аргумент явно, но в результате получается то же, что и при вызове `apply()`. Выбор метода `apply()` или `call()` зависит исключительно от того, как проще передать аргументы в функцию. Если вы собираетесь передать объект `arguments` или у вас уже есть массив аргументов, лучше выбрать метод `apply()`, в противном случае `call()` может оказаться предпочтительнее (если аргументы не передаются, эти методы идентичны).

Истинная мощь методов `apply()` и `call()` заключается не в передаче аргументов, а в возможности изменять значение `this` внутри функции, например:

Листинг FunctionTypeCallExample01.htm

```
window.color = "red";
var o = { color: "blue" };

function sayColor(){
    alert(this.color);
}

sayColor();             // red

sayColor.call(this);    // red
sayColor.call(window);  // red
sayColor.call(o);       // blue
```

Это измененная версия примера, который использовался для демонстрации объекта `this`. Функция `sayColor()` здесь также определена как глобальная, и когда она вызывается в глобальной области видимости, выводится строка `"red"`, потому что значение `this.color` интерпретируется как `window.color`. Затем она явно вызывается в глобальной области видимости как `sayColor.call(this)` и `sayColor.call(window)` с таким же результатом. Вызов `sayColor.call(o)` изменяет контекст функции на `o`, в результате выводится строка `"blue"`.

Преимущество применения метода `call()` (или `apply()`) с целью изменения области видимости состоит в том, что объекту не нужно ничего знать о методе. В первой версии этого примера функция `sayColor()` была добавлена к объекту `o` перед вызовом, а в обновленном примере это больше не требуется.

В ECMAScript 5 определен также дополнительный метод `bind()`. Он создает новый экземпляр функции, значение `this` которого *привязано* к значению, переданному в `bind()`, например:

Листинг FunctionTypeBindMethodExample01.htm


```
window.color = "red";
var o = { color: "blue" };

function sayColor(){
    alert(this.color);
}
var objectSayColor = sayColor.bind(o);
objectSayColor();    // blue
```

Здесь метод `bind()`, которому передается объект `o`, создает из `sayColor()` функцию `objectSayColor()`. У нее значение `this` указывает на `o`, поэтому даже когда она вызывается в глобальной области видимости, выводится строка `"blue"`. Преимущества этого приема обсуждаются в главе 22.

Метод `bind()` поддерживается в Internet Explorer 9+, Firefox 4+, Safari 5.1+, Opera 12+ и Chrome.

Для функций унаследованные методы `toLocaleString()` и `toString()` всегда возвращают код функции. Его точный формат зависит от браузера: одни браузеры возвращают исходный код без каких-либо изменений, а другие — внутреннее представление кода с удаленными комментариями и, возможно, с некоторыми изменениями, внесенными интерпретатором. Из-за этого полагаться на возвращенный код при реализации важного функционала не следует, хотя эта информация может пригодиться при отладке. Что касается метода `valueOf()`, то он просто возвращает саму функцию.

Оболочки примитивных типов

Специальные ссылочные типы `Boolean`, `Number` и `String` упрощают работу с соответствующими примитивными значениями. Они поддерживают стандартные возможности ссылочных типов, но имеют также специальные формы поведения, связанные с их примитивными аналогами. Каждый раз при чтении примитивного значения неявно создается соответствующий объект оболочки примитивного типа, обеспечивающий доступ к методам манипулирования данными. Рассмотрим следующий пример:

```
var s1 = "some text";  // некоторый текст
var s2 = s1.substring(2);
```

Здесь переменной `s1` присваивается примитивное строковое значение, а затем для нее вызывается метод `substring()` и результат сохраняется в `s2`. Примитивные значения не являются объектами, поэтому, по идее, у них не должно быть методов, но этот код работает как предполагалось. Чтобы это стало возможным, запускаются некоторые скрытые процессы. Доступ к `s1` во второй строке осуществляется в режиме чтения, то есть это значение читается из памяти. Всякий раз, когда строковое значение используется в режиме чтения, происходит трехэтапная процедура:

1. Создание экземпляра типа `String`.
2. Вызов указанного метода для экземпляра.
3. Уничтожение экземпляра.

Эти действия можно представить следующими тремя строками ECMAScript-кода:

```
var s1 = new String("some text");
var s2 = s1.substring(2);
s1 = null;
```

Благодаря такому поведению примитивное строковое значение может работать как объект. Те же три этапа выполняются для логических и числовых значений с типами `Boolean` и `Number` соответственно.

Главное различие между ссылочными типами и оболочками примитивных типов — время жизни объекта. Экземпляр ссылочного типа, созданный с помощью оператора `new`, остается в памяти до тех пор, пока не выходит из области видимости, тогда как автоматически создаваемые оболочки примитивных типов существуют только одну строку кода, после чего уничтожаются. Это означает, что к ним невозможно добавить свойства и методы во время выполнения. Взгляните на этот пример:

```
var s1 = "some text;
s1.color = "red";
alert(s1.color);    // undefined
```

Здесь во второй строке предпринимается попытка добавить к строке `s1` свойство `color`, но уже в следующей строке оно недоступно. Это происходит потому, что объект `String`, созданный во второй строке, уничтожается к моменту выполнения третьей строки. В третьей строке создается другой объект `String`, у которого нет свойства `color`.

Оболочки примитивных типов можно создавать явно с помощью конструкторов `Boolean`, `Number` и `String`, но делать это без необходимости не следует, потому что иначе многим разработчикам будет не совсем понятно, с каким значением они имеют дело, примитивным или ссылочным. Все оболочки примитивных типов преобразуются в логическое значение `true`, а вызов оператора `typeof` для них возвращает `"object"`.

Конструктор `Object` может работать как фабричный метод, возвращая экземпляр оболочки примитивного типа на основе типа полученного значения, например:

```
var obj = new Object("some text");
alert(obj instanceof String);  // true
```

При передаче строки в конструктор `Object` создается экземпляр `String`, для числового аргумента возвращается экземпляр `Number`, а для логического — экземпляр `Boolean`.

Имейте в виду, что вызов конструктора оболочки примитивного типа с помощью оператора `new` — это не то же самое, что вызов функции приведения типов с тем же именем:

```
var value = "25";
var number = Number(value);     // функция приведения типов
alert(typeof number);           // "number"

var obj = new Number(value);    // конструктор
alert(typeof obj);              // "object"
```

В этом примере в переменной `number` сохраняется примитивное числовое значение 25, а переменной `obj` присваивается экземпляр `Number`. Функции приведения типов обсуждаются в главе 3.

Хотя явно создавать оболочки примитивных типов не рекомендуется, при работе с примитивными значениями они играют важную роль. У оболочки каждого типа есть методы, которые упрощают работу с данными.

Тип Boolean

`Boolean` — это ссылочный тип, соответствующий булевым значениям. Чтобы создать объект `Boolean`, вызовите конструктор `Boolean`, передав ему значение `true` или `false`, например:

```
var booleanObject = new Boolean(true);
```

Экземпляры `Boolean` переопределяют метод `valueOf()`, чтобы он возвращал примитивное значение `true` или `false`. Метод `toString()` также переопределяется и возвращает строку `"true"` или `"false"`. К сожалению, объекты `Boolean` не только практически бесполезны в ECMAScript, они даже затрудняют понимание кода. Проблемы обычно возникают при использовании объектов `Boolean` в логических выражениях, например:

Листинг BooleanTypeExample01.htm


```
var falseObject = new Boolean(false);
var result = falseObject && true;
alert(result);    // true

var falseValue = false;
result = falseValue && true;
alert(result);    // false
```

В этом коде создается объект `Boolean` со значением `false`, а затем для него и примитивного значения `true` выполняется операция И. В булевой математике результатом операции `false` И `true` является `false`, однако в этой строке кода оценивается объект `falseObject`, а не его значение (`false`). Как уже отмечалось, в логических выражениях все объекты автоматически преобразуются в `true`, так что вместо `falseObject` на самом деле используется значение `true`. Применение оператора И к двум значениям `true` дает в результате `true`.

Есть и другие различия между примитивным и ссылочным логическими типами. Так, оператор `typeof` возвращает `"boolean"` для примитивного типа, но `"object"` для

ссылочного. Кроме того, объект `Boolean` является экземпляром типа `Boolean`, поэтому оператор `instanceof` возвращает для него `true`, а для примитивного значения — `false`:

```
alert(typeof falseObject);              // object
alert(typeof falseValue);               // boolean
alert(falseObject instanceof Boolean);  // true
alert(falseValue instanceof Boolean);   // false
```

Важно понимать различия между примитивным логическим значением и объектом `Boolean` и по возможности не использовать последний.

Тип Number

Тип `Number` — это ссылочный тип для числовых значений. Чтобы создать объект `Number`, вызовите конструктор `Number`, передав в него любое число, например:

Листинг NumberTypeExample01.htm

```
var numberObject = new Number(10);
```

Как и `Boolean`, тип `Number` переопределяет методы `valueOf()`, `toLocaleString()` и `toString()`. Метод `valueOf()` возвращает примитивное числовое значение, представленное объектом, а другие два метода возвращают число как строку. Как отмечено в главе 3, метод `toString()` может принимать аргумент, указывающий основание системы счисления:

Листинг NumberTypeExample01.htm

```
var num = 10;
alert(num.toString());     // "10"
alert(num.toString(2));    // "1010"
alert(num.toString(8));    // "12"
alert(num.toString(10));   // "10"
alert(num.toString(16));   // "a"
```

Кроме унаследованных методов у типа `Number` есть несколько дополнительных методов, позволяющих форматировать числа как строки.

Метод `toFixed()` возвращает число как строку с указанным количеством знаков после точки:

Листинг NumberTypeExample01.htm

```
var num = 10;
alert(num.toFixed(2));    // "10.00"
```

Здесь методу `toFixed()` передается аргумент 2, поэтому он возвращает строку `"10.00"`, заменяя отсутствующую дробную часть двумя нулями. Если число содержит после точки больше знаков, чем указывает аргумент, результат округляется до ближайшего разряда, например:

```
var num = 10.005;
alert(num.toFixed(2));    // "10.01"
```

Округление с помощью метода `toFixed()` может быть полезно в приложениях, работающих с денежными суммами, но важно отметить, что оно выполняется в браузерах по-разному. В Internet Explorer до версии 8 включительно числа из интервалов (–0.94,–0.5] и [0.5,0.94) округляются неправильно, если в метод `toFixed()` передается 0. В этих случаях Internet Explorer возвращает 0, хотя должен возвращать –1 или 1 в зависимости от знака. В Internet Explorer 9 эта ошибка устранена, а другие браузеры работают правильно.

Метод toFixed() может представлять числа, содержащие от 0 до 20 разрядов после точки. Этот диапазон типичен для разных реализаций, хотя в некоторых браузерах он может быть шире.

С форматированием чисел связан также метод `toExponential()`, который возвращает строку с числом в экспоненциальной записи. Как и предыдущий метод, он принимает один аргумент — количество выводимых знаков после точки:

```
var num = 10;
alert(num.toExponential(1));  // "1.0e+1"
```

Этот код выводит строку `"1.0e+1"`, хотя обычно для столь малых чисел экспоненциальная запись не используется. Если вам нужна более уместная форма записи, следует задействовать метод `toPrecision()`.

Метод `toPrecision()` возвращает строковое представление числа с фиксированным количеством знаков или в экспоненциальной записи в зависимости от того, в чем больше смысла. В качестве единственного аргумента он принимает общее количество разрядов итогового числа (не включая степень), например:

Листинг NumberTypeExample01.htm


```
var num = 99;
alert(num.toPrecision(1));  // "1e+2"
alert(num.toPrecision(2));  // "99"
alert(num.toPrecision(3));  // "99.0"
```

В этом примере в первой строке число 99 выводится с одним разрядом как `"1e+2"`, или 100. Поскольку 99 не может быть точно представлено с помощью одного разряда, оно округляется до 100. С двумя разрядами число 99 выводится как `"99"`, а с тремя — как `"99.0"`. По сути, метод `toPrecision()` на основе полученного числового значения выбирает и вызывает метод `toFixed()` или `toExponential()`; все три метода округляют значение вверх или вниз для точного представления числа с правильным количеством разрядов.

Метод toPrecision() может представлять числа, содержащие от 1 до 21 десятичных разрядов. Этот диапазон типичен для разных реализаций, хотя в некоторых браузерах он может быть шире.

Подобно Boolean, объекты Number обеспечивают важный функционал для работы с числовыми значениями, но их не следует создавать непосредственно из-за тех же потенциальных проблем. Операторы typeof и instanceof работают с примитивными и ссылочными числами по-разному:

```
var numberObject = new Number(10);
var numberValue = 10;
alert(typeof numberObject);             // "object"
alert(typeof numberValue);              // "number"
alert(numberObject instanceof Number);  // true
alert(numberValue instanceof Number);   // false
```

Для примитивных чисел оператор typeof всегда возвращает "number", а для объектов Number — "object". Соответственно, объект Number является экземпляром типа Number, а примитивное число — нет.

Тип String

Тип String — это ссылочный аналог строк. Объекты этого типа создаются с помощью конструктора String:

Листинг StringTypeExample01.htm

```
var stringObject = new String("hello world");
```

Методы объекта String доступны для всех строковых примитивов. Все три унаследованных метода — valueOf(), toLocaleString() и toString() — возвращают примитивное строковое значение объекта.

Каждый экземпляр String содержит единственное свойство length, в котором хранится количество знаков в строке, например:

```
var stringValue = "hello world";
alert(stringValue.length);    // "11"
```

В этом примере выводится строка "11", отражающая количество знаков в строке "hello world". Имейте в виду, что даже если строка содержит двухбайтовые символы, а не только однобайтовые ASCII-символы, каждый символ все равно считается за один.

Тип String содержит целый ряд методов, помогающих выполнять самые разные действия со строками.

Методы для работы с символами

Для доступа к конкретным символам строки используются методы charAt() и charCodeAt(), которые принимают один аргумент: позицию символа с отсчетом от нуля. Метод charAt() просто возвращает символ в конкретной позиции как строку из одного символа (знакового типа в ECMAScript нет):

```
var stringValue = "hello world";
alert(stringValue.charAt(1));  // "e"
```

Все правильно: в позиции 1 строки `"hello world"` находится символ `"e"`. Если вам нужен не сам символ, а его код, используйте метод `charCodeAt()`:

```
var stringValue = "hello world";
alert(stringValue.charCodeAt(1));  // "101"
```

В этом примере выводится строка `"101"`, отражающая код символа `"e"` строки.

В ECMAScript 5 определен также другой способ доступа к отдельному символу: с помощью числового индекса в квадратных скобках:

```
var stringValue = "hello world";
alert(stringValue[1]);    // "e"
```

Доступ к отдельным символам посредством скобочной нотации поддерживается в Internet Explorer 8 и во всех текущих версиях Firefox, Safari, Chrome и Opera. Если этот синтаксис используется в Internet Explorer 7 или более ранней версии, результат не определен.

Методы манипулирования строками

Для работы со строками можно использовать несколько методов. Первый из них, `concat()`, присоединяет одну или несколько строк к другой и возвращает объединенную строку:

```
var stringValue = "hello ";
var result = stringValue.concat("world");
alert(result);          // "hello world"
alert(stringValue);     // "hello"
```

В этом примере вызов метода `concat()` для `stringValue` возвращает строку `"hello world"`, а значение `stringValue` остается без изменений. Метод `concat()` принимает любое количество аргументов, что позволяет составить строку из любого количества других строк:

```
var stringValue = "hello ";
var result = stringValue.concat("world", "!");
alert(result);          // "hello world!"
alert(stringValue);     // "hello"
```

В этом измененном примере к строке `"hello"` присоединяются строки `"world"` и `"!"`. Хотя для конкатенации строк предоставляется метод `concat()`, оператор сложения (`+`) используется чаще и в большинстве случаев работает быстрее, даже если строк несколько.

Для создания строковых значений из подстрок в ECMAScript применяются методы `slice()`, `substr()` и `substring()`. Все они возвращают подстроку строки, для которой

были вызваны, и принимают один или два аргумента. Первым является позиция, с которой начинается захват подстроки, а второй, если он используется, указывает, когда нужно остановиться. У методов `slice()` и `substring()` второй аргумент определяет позицию, перед которой операция завершается, а у метода `substr()` — количество возвращаемых символов. Если второй аргумент опущен в каком-либо методе, операция выполняется до конца строки. Как и метод `concat()`, методы `slice()`, `substr()` и `substring()` не изменяют саму строку — они просто возвращают примитивное строковое значение, оставляя оригинал неизменным. Рассмотрим следующий пример:

Листинг StringTypeManipulationMethodsExample01.htm


```
var stringValue = "hello world";
alert(stringValue.slice(3));         // "lo world"
alert(stringValue.substring(3));     // "lo world"
alert(stringValue.substr(3));        // "lo world"
alert(stringValue.slice(3, 7));      // "lo w"
alert(stringValue.substring(3,7));   // "lo w"
alert(stringValue.substr(3, 7));     // "lo worl"
```

В этом примере методы `slice()`, `substr()` и `substring()` используются аналогичным образом и в основном возвращают одинаковые значения. При единственном аргументе `3` все они возвращают строку `"lo world"`, потому что в позиции 3 находится вторая буква `"l"` в слове `"hello"`. При двух аргументах `3` и `7` методы `slice()` и `substring()` возвращают `"lo w"` (буква `"o"` в слове `"world"` находится в позиции 7 и не включается в результат), а метод `substr()` — `"lo worl"`, потому что в его случае второй аргумент указывает количество возвращаемых символов.

Если аргументом является отрицательное число, эти методы работают иначе. Метод `slice()` вычитает его из длины строки. Метод `substr()` вычитает первый отрицательный аргумент из длины строки, а второй преобразует в 0. Метод `substring()` оба отрицательных аргумента преобразует в 0, например:

Листинг StringTypeManipulationMethodsExample01.htm

```
var stringValue = "hello world";
alert(stringValue.slice(-3));         // "rld"
alert(stringValue.substring(-3));     // "hello world"
alert(stringValue.substr(-3));        // "rld"
alert(stringValue.slice(3, -4));      // "lo w"
alert(stringValue.substring(3, -4));  // "hel"
alert(stringValue.substr(3, -4));     // "" (пустая строка)
```

Этот пример ясно демонстрирует различия между тремя методами. Когда методы `slice()` и `substr()` вызываются с единственным отрицательным аргументом, они работают одинаково, потому что –3 преобразуется в 7 (длина строки плюс аргумент). По сути, это преобразует вызовы в `slice(7)` и `substr(7)`. Что касается метода `substring()`, то он возвращает всю строку, потому что –3 преобразуется в 0.

В Internet Explorer до версии 9 при передаче отрицательного числа в метод substr() возвращается исходная строка. В Internet Explorer 9 эта ошибка устранена.

Если отрицателен второй аргумент, эти три метода работают по-разному. Метод `slice()` преобразует второй аргумент в 7, по сути, изменяя вызов на `slice(3, 7)`, и возвращает `"lo w"`. Для метода `substring()` второй аргумент преобразуется в 0, превращая вызов метода в `substring(3, 0)`, который на самом деле эквивалентен `substring(0, 3)`, потому что этот метод интерпретирует меньшее число как начальную позицию, а большее — как конечную. Для метода `substr()` второй аргумент также преобразуется в 0; это означает, что в возвращенной строке должно быть 0 знаков, поэтому возвращается пустая строка.

Методы поиска строк

Есть два метода поиска подстрок в других строках: `indexOf()` и `lastIndexOf()`. Оба метода ищут в строке конкретную подстроку и возвращают ее позицию (или –1, если найти ее не удается). Разница между ними в том, что метод `indexOf()` начинает искать подстроку с начала строки, а `lastIndexOf()` — с конца. Рассмотрим пример:

Листинг StringTypeLocationMethodsExample01.htm


```
var stringValue = "hello world";
alert(stringValue.indexOf("o"));      // 4
alert(stringValue.lastIndexOf("o"));  // 7
```

Здесь первая подстрока `"o"` встречается в позиции 4, это буква `"o"` в слове `"hello"`. Последнее вхождение подстроки `"o"` в строку имеет место в позиции 7 в слове `"world"`. Если бы в строке содержалась только одна буква `"o"`, методы `indexOf()` и `lastIndexOf()` вернули бы одно и то же значение.

Оба эти метода могут принимать необязательный второй аргумент, указывающий начальную позицию поиска в строке. Иначе говоря, метод `indexOf()` выполняет поиск с этой позиции до конца строки, пропуская все символы перед начальной позицией, а метод `lastIndexOf()` начинает поиск с указанной позиции и продвигается к началу строки, пропуская символы между указанной позицией и концом строки:

```
var stringValue = "hello world";
alert(stringValue.indexOf("o", 6));       // 7
alert(stringValue.lastIndexOf("o", 6));   // 4
```

Как видите, если в каждый метод передать второй аргумент 6, возвращаются результаты, противоположные предыдущим. На этот раз метод `indexOf()` возвращает 7, потому что он начинает поиск подстроки с позиции 6 (буква `"w"`) и обнаруживает букву `"o"` в позиции 7. Метод `lastIndexOf()` возвращает 4, потому что он начинает поиск с позиции 6 и продвигается к началу строки, пока не встречает букву `"o"` в слове `"hello"`. С помощью второго аргумента можно найти все экземпляры подстроки в строке, циклически вызывая метод `indexOf()` или `lastIndexOf()`:

Листинг StringTypeLocationMethodsExample02.htm


```
var stringValue = "Lorem ipsum dolor sit amet, consectetur adipisicing...";
var positions = new Array();
var pos = stringValue.indexOf("e");

while(pos > -1){
    positions.push(pos);
    pos = stringValue.indexOf("e", pos + 1);
}

alert(positions);    // "3,24,32,35"
```

Этот фрагмент обрабатывает строку, постоянно увеличивая позицию, в которой метод `indexOf()` начинает поиск. Сначала определяется позиция первой подстроки `"e"` в строке, а затем запускается цикл, в котором в метод `indexOf()` каждый раз передается позиция последней обнаруженной буквы `"e"`, увеличенная на 1. Благодаря этому поиск продолжается после обнаружения каждой подстроки. Позиции подстрок сохраняются в массиве `positions`, чтобы эти данные можно было использовать позже.

Метод trim()

В ECMAScript 5 введен метод `trim()`, который создает копию строки, удаляет все начальные и конечные пробельные символы, а затем возвращает результат, например:

```
var stringValue = " hello world ";
var trimmedStringValue = stringValue.trim();
alert(stringValue);         // " hello world "
alert(trimmedStringValue);  // "hello world"
```

Заметьте, что поскольку метод `trim()` возвращает копию строки, в исходной строке начальный и конечный пробелы сохраняются. Этот метод реализован в Internet Explorer 9+, Firefox 3.5+, Safari 5+, Opera 10.5+ и Chrome. Браузеры Firefox 3.5+, Safari 5+ и Chrome 8+ поддерживают также методы `trimLeft()` и `trimRight()`, которые удаляют только начальные и только конечные пробельные символы соответственно.

Методы изменения регистра символов

Для изменения регистра символов можно использовать методы `toLowerCase()`, `toLocaleLowerCase()`, `toUpperCase()` и `toLocaleUpperCase()`. Методы `toLowerCase()` и `toUpperCase()` созданы по образцу аналогичных Java-методов (`java.lang.String`), а методы `LocaleLowerCase()` и `toLocaleUpperCase()`, по идее, должны быть реализованы на основе конкретного регионального стандарта. Во многих региональных стандартах эти методы не отличаются от универсальных, но в некоторых языках, например в турецком, действуют специальные правила преобразования регистра символов Юникода, что требует использования специфичных методов. Вот некоторые примеры:

Листинг StringTypeCaseMethodExample01.htm


```
var stringValue = "hello world";
alert(stringValue.toLocaleUpperCase());  // "HELLO WORLD"
alert(stringValue.toUpperCase());        // "HELLO WORLD"
alert(stringValue.toLocaleLowerCase());  // "hello world"
alert(stringValue.toLowerCase());        // "hello world"
```

Здесь методы `toLocaleUpperCase()` и `toUpperCase()` выводят строку `"HELLO WORLD"`, а методы `toLocaleLowerCase()` и `toLowerCase()` — `"hello world"`. Если вы не знаете, в какой языковой среде будет выполняться код, безопаснее использовать методы, специфичные для регионального стандарта.

Методы сопоставления строк с шаблонами

Тип `String` содержит несколько методов для сопоставления строк с шаблонами. Первый из них, `match()`, аналогичен методу `exec()` объекта `RegExp`. В качестве единственного аргумента он принимает или строку регулярного выражения, или объект `RegExp`, например:

Листинг StringTypePatternMatchingExample01.htm

```
var text = "cat, bat, sat, fat";
var pattern = /.at/;

// то же, что и pattern.exec(text)
var matches = text.match(pattern);
alert(matches.index);      // 0
alert(matches[0]);         // "cat"
alert(pattern.lastIndex);  // 0
```

Метод `match()` возвращает такой же массив, что и метод `exec()` объекта `RegExp`, когда в него передается строка. Первым элементом массива является строка, которая соответствует всему шаблону, а все остальные элементы (если они есть) представляют группы захвата в выражении.

Другой метод поиска шаблонов называется `search()` и принимает такой же аргумент, что и метод `match()`, а именно регулярное выражение в форме строки или объекта `RegExp`. Метод `search()` возвращает индекс первого вхождения шаблона в строку или –1, если найти его не удается. Поиск шаблона ведется с начала строки. Вот пример:

Листинг StringTypePatternMatchingExample01.htm


```
var text = "cat, bat, sat, fat";
var pos = text.search(/at/);
alert(pos);    // 1
```

Здесь вызов `search(/at/)` возвращает 1 — позицию первого вхождения подстроки `"at"` в строку.

Чтобы упростить замену подстрок, ECMAScript предоставляет метод `replace()`, который принимает два аргумента. Первым может быть объект `RegExp` или строка

(она не преобразуется в регулярное выражение), вторым — строка или функция. Если первым аргументом является строка, заменяется только первое вхождение подстроки в строку. Чтобы заменить все экземпляры подстроки, необходимо передать в метод регулярное выражение с глобальным флагом, например:

Листинг StringTypePatternMatchingExample01.htm

```
var text = "cat, bat, sat, fat";
var result = text.replace("at", "ond");
alert(result);    // "cond, bat, sat, fat"

result = text.replace(/at/g, "ond");
alert(result);    // "cond, bond, sond, fond"
```

В этом примере строка `"at"` сначала передается в метод `replace()` с текстом для замены `"ond"`. В результате слово `"cat"` изменяется на `"cond"`, но остальная часть строки остается неизменной. После замены первого аргумента регулярным выражением с глобальным флагом каждое вхождение `"at"` заменяется строкой `"ond"`.

Если второй аргумент является строкой, для вставки значений можно использовать несколько специальных последовательностей символов. Доступные в ECMA-262 последовательности представлены в таблице.

Последовательность	Текст для замены
$$	$
$&	Подстрока, совпадающая со всем шаблоном. То же, что RegExp.lastMatch
$'	Часть строки перед совпавшей подстрокой. То же, что RegExp.rightContext
$`	Часть строки после совпавшей подстроки. То же, что RegExp.leftContext
$n	n-ная группа захвата, где n — значение от 0 до 9. Например, $1 — это первая группа захвата, $2 — вторая, и т. д. Если захвата нет, используется пустая строка
$nn	nn-ная группа захвата, где nn — значение от 01 до 99. Например, $01 — это первая группа захвата, $02 — вторая, и т. д. Если захвата нет, используется пустая строка

С помощью этих специальных последовательностей можно заменять подстроки, используя сведения о последнем совпадении, например:

Листинг StringTypePatternMatchingExample01.htm

```
var text = "cat, bat, sat, fat";
result = text.replace(/(.at)/g, "word ($1)");
alert(result);    // word (cat), word (bat), word (sat), word (fat)
```

Здесь с помощью последовательности `$1` каждое слово, оканчивающееся на `"at"`, заменяется словом `"word"`, за которым в скобках следует исходное слово.

Второй аргумент метода `replace()` может быть функцией. При наличии одного совпадения в нее передаются три аргумента: совпадение, позиция совпадения в строке и вся строка. Если групп захвата несколько, каждая совпавшая строка передается в функцию как аргумент, при этом двумя последними аргументами являются позиция совпадения с шаблоном в строке и оригинальная строка. Функция должна возвращать строку, указывающую, чем следует заменить совпадение. Использование функции в качестве второго аргумента обеспечивает более детальный контроль над текстом для замены, например:

Листинг StringTypePatternMatchingExample01.htm

```
function htmlEscape(text){
    return text.replace(/[<>"&]/g, function(match, pos, originalText){
        switch(match){
            case "<":
                return "&lt;";
            case ">":
                return "&gt;";
            case "&":
                return "&amp;";
            case "\"":
                return "&quot;";
        }
    });
}
alert(htmlEscape("<p class=\"greeting\">Hello world!</p>"));
// "&lt;p class=&quot;greeting&quot;&gt;Hello world!&lt;/p&gt";
```

Здесь определяется функция `htmlEscape()`, которая обрабатывает знаки «меньше», «больше», амперсанды и двойные кавычки, подготавливая их к вставке в HTML-код. Для этого выполняется поиск этих знаков с помощью регулярного выражения, а затем определяется функция, которая возвращает специфические HTML-сущности для каждого совпавшего знака.

Последний метод для работы с шаблонами, `split()`, разбивает строку на массив подстрок по разделителю, которым может быть строка или объект `RegExp` (в этом методе строка не считается регулярным выражением). Необязательный второй аргумент, ограничение массива, гарантирует, что возвращенный массив не будет превышать определенный размер. Рассмотрим пример:

Листинг StringTypePatternMatchingExample01.htm


```
var colorText = "red,blue,green,yellow";

// ["red", "blue", "green", "yellow"]
var colors1 = colorText.split(",");

// ["red", "blue"]
var colors2 = colorText.split(",", 2);
```

```
// ["", ",", ",", ",", ""]
var colors3 = colorText.split(/[^\,]+/);
```

В этом примере строка `colorText` содержит список цветов, разделенных запятыми. Вызов `split(",")` возвращает массив этих цветов, разделяя строку по запятым. Далее метод вызывается со вторым аргументом, равным двум, при этом результат ограничивается двумя элементами. Наконец, с помощью регулярного выражения можно получить массив запятых. Обратите внимание, что в последнем вызове `split()` возвращенный массив содержит пустую строку перед запятыми и после. Это происходит потому, что разделитель, указанный с помощью регулярного выражения, имеется и в начале строки (подстрока `"red"`), и в конце (подстрока `"yellow"`).

Браузеры различаются по поддержке регулярных выражений в методе `split()`. Хотя простые шаблоны обычно работают одинаково, шаблоны без совпадений и шаблоны с группами захвата могут обрабатываться в браузерах совершенно по-разному. Вот некоторые примечательные различия:

- Internet Explorer до версии 8 включительно игнорирует группы захвата, хотя в ECMA-262 указано, что они должны сохраняться в массиве результатов. Internet Explorer 9 добавляет группы захвата в результаты.
- Firefox до версии 3.6 включительно добавляет пустые строки в массив результатов, если у группы захвата нет совпадения. В ECMA-262 указано, что группы захвата без совпадения должны представляться в массиве результатов как `undefined`.

Есть и другие различия, связанные с применением групп захвата в регулярных выражениях. Используя их, не забывайте тщательно тестировать код в разных браузерах.

Более подробное обсуждение проблем с методом split() и группами захвата в контексте совместимости браузеров см. в посте «JavaScript split bugs: Fixed!» в блоге Стивена Левитана (Steven Levithan) по адресу http://blog.stevenlevithan.com/archives/cross-browser-split.

Метод localeCompare()

Метод `localeCompare()` сравнивает одну строку с другой и возвращает одно из трех значений:

- Если строка должна располагаться по алфавиту перед строковым аргументом, возвращается отрицательное число (обычно −1, но вообще значение может зависеть от реализации).
- Если строка равна строковому аргументу, возвращается 0.
- Если строка должна располагаться по алфавиту после строкового аргумента, возвращается положительное число (обычно 1, но значение может зависеть от реализации).

Рассмотрим пример:

Листинг StringTypeLocaleCompareExample01.htm


```
var stringValue = "yellow";
alert(stringValue.localeCompare("brick"));      // 1
alert(stringValue.localeCompare("yellow"));     // 0
alert(stringValue.localeCompare("zoo"));        // -1
```

В этом коде строка "yellow" сравнивается со значениями "brick", "yellow" и "zoo". Поскольку строка "brick" предшествует по алфавиту строке "yellow", метод localeCompare() возвращает 1. Строка "yellow" равна самой себе, поэтому во втором случае метод localeCompare() возвращает 0. Наконец, строка "zoo" должна располагаться после строки "yellow", поэтому в третий раз возвращается значение –1. Так как значения зависят от реализации, лучше использовать метод localeCompare() следующим образом:

Листинг StringTypeLocaleCompareExample01.htm

```
function determineOrder(value) {
    var result = stringValue.localeCompare(value);
    if (result < 0){
      alert("The string 'yellow' comes before the string '" + value + "'.");
    } else if (result > 0) {
      alert("The string 'yellow' comes after the string '" + value + "'.");
    } else {
      alert("The string 'yellow' is equal to the string '" + value + "'.");
    }
}

determineOrder("brick");
determineOrder("yellow");
determineOrder("zoo");
```

Используя такую конструкцию, вы можете быть уверены, что код будет работать правильно во всех реализациях.

Уникальность метода localeCompare() заключается в том, что региональный стандарт реализации (страна и язык) точно указывает, как метод должен работать. В США, где стандартным языком ECMAScript-реализаций является английский, метод localeCompare() чувствителен к регистру, то есть прописные буквы предшествуют по алфавиту строчным, но в других региональных стандартах это может быть не так.

Метод fromCharCode()

У конструктора String есть метод fromCharCode(), который принимает один или несколько кодов символов и преобразует их в строку. По сути, эта операция обратна вызову метода экземпляра charCodeAt(). Взгляните на пример:

Листинг StringTypeFromCharCodeExample01.htm


```
alert(String.fromCharCode(104, 101, 108, 108, 111));  // "hello"
```

Здесь метод `fromCharCode()` составляет слово `"hello"` из соответствующих кодов символов.

Методы для форматирования HTML-кода

Производители веб-браузеров быстро осознали потенциал динамического форматирования HTML-кода средствами JavaScript и добавили в спецификацию несколько предназначенных для этого методов, которые указаны в приведенной таблице. Имейте, однако, в виду, что эти методы используются редко, потому что они генерируют несемантическую разметку.

Метод	Вывод
anchor(имя)	строка
big()	<big>строка</big>
bold()	строка
fixed()	<tt>строка</tt>
fontcolor(цвет)	строка
fontsize(размер)	строка
italics()	строка
link(url)	строка
small()	строка
strike()	<strike>строка</strike>
sub()	_{строка}
sup()	^{строка}

Встроенные одиночные объекты

ECMA-262 определяет встроенный объект как «любой объект, предоставляемый ECMAScript-реализацией, не зависящий от среды выполнения и присутствующий в начале выполнения ECMAScript-программы». Это означает, что разработчикам не нужно явно создавать встроенные объекты, они уже созданы. Вы уже знаете большинство встроенных объектов, таких как `Object`, `Array` и `String`. В ECMA-262 также определены два встроенных одиночных объекта: `Global` и `Math`.

Объект Global

Объект `Global` является самым необычным объектом в ECMAScript, потому что он недоступен явно. В ECMA-262 он описывается как некое подобие склада для свойств и методов, у которых без `Global` не было бы объекта-владельца. В действительности глобальных переменных или глобальных функций не существует — все

переменные и функции, определенные глобально, становятся свойствами объекта `Global`. Функции, описанные ранее в этой книге, такие как `isNaN()`, `isFinite()`, `parseInt()` и `parseFloat()`, на самом деле являются методами объекта `Global`. Кроме них у объекта `Global` есть несколько других методов.

Методы кодирования URI

Методы `encodeURI()` и `encodeURIComponent()` используются для кодирования универсальных идентификаторов ресурса (Uniform Resource Identifier, URI), передаваемых браузеру. Допустимые URI не могут содержать определенные знаки, например пробелы. Чтобы браузер все же мог принимать и понимать их, методы кодирования URI заменяют все недопустимые знаки специальными кодами в кодировке UTF-8.

Метод `encodeURI()` работает с целыми URI (такими, как `www.wrox.com/illegal value.htm`), а метод `encodeURIComponent()` — исключительно с их сегментами (например, `illegal value.htm` из предыдущего URI). Основное различие между двумя методами заключается в том, что `encodeURI()` не кодирует специальные знаки, входящие в URI, такие как точка, косая черта, вопросительный знак и знак решетки, а `encodeURIComponent()` кодирует любой нестандартный знак, который обнаруживает, например:

Листинг GlobalObjectURIEncodingExample01.htm


```
var uri = "http://www.wrox.com/illegal value.htm#start";

// "http://www.wrox.com/illegal%20value.htm#start"
alert(encodeURI(uri));

// "http%3A%2F%2Fwww.wrox.com%2Fillegal%20value.htm%23start"
alert(encodeURIComponent(uri));
```

Метод `encodeURI()` оставил значение почти нетронутым, заменив только пробел кодом `%20`, в то время как метод `encodeURIComponent()` заменил все не алфавитно-цифровые символы их закодированными эквивалентами. По этой причине метод `encodeURI()` можно использовать с полными URI, а `encodeURIComponent()` — только со строками, которые добавляются в конец существующих URI.

Метод encodeURIComponent() используется гораздо чаще, чем encodeURI(), потому что аргументы строк запросов обычно кодируются отдельно от базового URI.

У этих методов есть обратные методы `decodeURI()` и `decodeURIComponent()`. Метод `decodeURI()` декодирует только те коды, которые генерирует метод `encodeURI()`. Например, код `%20` заменяется пробелом, а код `%23` не заменяется, потому что он представляет знак решетки (#), который метод `encodeURI()` пропускает. Соответственно, метод `decodeURIComponent()` декодирует все коды, создаваемые методом `encodeURIComponent()`; по сути, это означает, что он декодирует все специальные значения. Рассмотрим пример:

Листинг GlobalObjectURIDecodingExample01.htm


```
var uri = "http%3A%2F%2Fwww.wrox.com%2Fillegal%20value.htm%23start";

// http%3A%2F%2Fwww.wrox.com%2Fillegal value.htm%23start
alert(decodeURI(uri));

// http://www.wrox.com/illegal value.htm#start
alert(decodeURIComponent(uri));
```

Переменная `uri` содержит строку, закодированную с помощью метода `encodeURIComponent()`. Первое закомментированное значение является результатом вызова метода `decodeURI()`, который лишь заменил код `%20` пробелом. Второе значение возвращено методом `decodeURIComponent()`, который заменяет все специальные коды соответствующими знаками (эта строка не является допустимым URI).

Методы encodeURI(), encodeURIComponent(), decodeURI() и decodeURIComponent() заменяют методы escape() и unescape(), которые признаны устаревшими в третьей редакции ECMA-262. Новые методы предпочтительнее во всех ситуациях, потому что они кодируют все символы Юникода, а старые правильно кодируют только ASCII-символы. Не используйте методы escape() и unescape() в окончательной версии кода.

Метод eval()

Метод `eval()` является, наверное, самым мощным во всем языке ECMAScript. Он подобен целому интерпретатору ECMAScript и принимает один аргумент, строку ECMAScript-кода (или JavaScript-кода), которую нужно выполнить, например:

```
eval("alert('Hello')");
```

Этот вызов функционально эквивалентен следующему:

```
alert("Hello");
```

Когда интерпретатор обнаруживает вызов `eval()`, он преобразует его аргумент в фактические инструкции ECMAScript и заменяет ими аргумент. Код, выполненный с помощью `eval()`, считается частью контекста выполнения, в котором был вызван метод, и имеет ту же цепочку областей видимости, что и этот контекст. Это означает, что на переменные, определенные во внешнем контексте, можно ссылаться при вызове `eval()`, например:

```
var msg = "Hello world!";
eval("alert(msg)");    // "Hello world!"
```

Переменная `msg` определена вне контекста вызова `eval()`, но при вызове `alert()` все же выводится текст `"Hello world!"`, потому что вторая строка заменяется реальной строкой кода. Аналогично можно определить функцию или переменные при вызове `eval()` и ссылаться на них во внешнем коде:

```
eval("function sayHi() { alert('Hello'); }");
sayHi();
```

Здесь функция `sayHi()` определена внутри вызова `eval()`, но поскольку он заменяется фактической функцией, можно вызвать `sayHi()` в следующей строке. Для переменных этот механизма работает так же:

```
eval("var msg = 'Hello world!';");
alert(msg);    // "Hello world!"
```

Любые переменные или функции, созданные внутри `eval()`, не поднимаются, потому что при синтаксическом анализе кода содержатся в строке. Они создаются только во время выполнения метода `eval()`.

В строгом режиме переменные и функции, созданные внутри `eval()`, недоступны снаружи, так что в этих двух примерах возникли бы ошибки. В строгом режиме попытка присвоить значение переменной `eval` также вызывает ошибку:

```
"use strict";
eval = "Hello";  // ошибка
```

Возможность интерпретировать строки кода очень эффективна, но и очень опасна. Будьте крайне осторожны с методом eval(), особенно при передаче в него данных, введенных пользователем. Злоумышленник может попытаться ввести значения, нарушающие безопасность сайта или приложения (это называется инъекцией кода).

Свойства объекта Global

С некоторыми свойствами объекта `Global` мы уже встречались. Например, в их число входят специальные значения `undefined`, `NaN` и `Infinity`. Конструкторы встроенных ссылочных типов, таких как `Object` и `Function`, также являются свойствами объекта `Global`. Все его свойства указаны в таблице.

Свойство	Описание
undefined	Специальное значение undefined
NaN	Специальное значение NaN
Infinity	Специальное значение Infinity
Object	Конструктор Object
Array	Конструктор Array
Function	Конструктор Function
Boolean	Конструктор Boolean
String	Конструктор String
Number	Конструктор Number

Свойство	Описание
Date	Конструктор Date
RegExp	Конструктор RegExp
Error	Конструктор Error
EvalError	Конструктор EvalError
RangeError	Конструктор RangeError
ReferenceError	Конструктор ReferenceError
SyntaxError	Конструктор SyntaxError
TypeError	Конструктор TypeError
URIError	Конструктор URIError

В ECMAScript 5 запрещено явно присваивать значения свойствам `undefined`, `NaN` и `Infinity`. Попытка сделать это приведет к ошибке даже в нестрогом режиме.

Объект Window

Хотя в ECMA-262 не указан способ непосредственного доступа к объекту `Global`, в веб-браузерах он реализуется с помощью делегата — объекта `window`. Это означает, что все переменные и функции, объявленные в глобальной области видимости, становятся свойствами `window`. Рассмотрим такой пример:

Листинг GlobalObjectWindowExample01.htm


```
var color = "red";

function sayColor(){
    alert(window.color);
}

window.sayColor();  // "red"
```

Здесь определяются глобальная переменная `color` и глобальная функция `sayColor()`, внутри которой переменная используется как `window.color`, чтобы показать, что она стала свойством `window`. Затем функция вызывается непосредственно для объекта `window` как `window.sayColor()`, в результате чего выводится оповещение.

Объект window в JavaScript используется при решении самых разных задач, а не только как реализация ECMAScript-объекта Global. Подробно объект window и объектная модель браузера обсуждаются в главе 8.

Другой способ получить объект `Global` — использовать следующий код:

```
var global = function(){
    return this;
}();
```

В этом коде создается и сразу же вызывается функция-выражение, которая возвращает значение `this`. Как уже отмечалось, если при вызове функции значение `this` не указано явно (то есть функция вызывается не как метод объекта и не с помощью методов `call()`/`apply()`), оно эквивалентно объекту `Global`, Таким образом, вызов функции, которая просто возвращает `this`, обеспечивает согласованный способ получения объекта `Global` в любой среде выполнения. Функции-выражения подробнее обсуждаются в главе 7.

Объект Math

В ECMAScript-объекте `Math` реализованы математические формулы и константы. Вычисления с ним выполняются быстрее, чем аналогичный JavaScript-код. Далее описаны свойства и методы объекта `Math`.

Свойства объекта Math

Объект `Math` имеет несколько свойств, которые в основном представляют специальные математические значения. Они приведены в таблице.

Свойство	Описание
Math.E	Значение e, основание натурального логарифма
Math.LN10	Натуральный логарифм 10
Math.LN2	Натуральный логарифм 2
Math.LOG2E	Двоичный логарифм e
Math.LOG10E	Десятичный логарифм e
Math.PI	Число π
Math.SQRT1_2	Квадратный корень из ½
Math.SQRT2	Квадратный корень из 2

Смысл и способы использования этих значений мы рассматривать не будем, но вам следует знать, что они доступны.

Методы min() и max()

Объект `Math` содержит множество методов, предназначенных для выполнения математических вычислений разной сложности.

Методы `min()` и `max()` определяют наименьшее и наибольшее числа в группе чисел. Они принимают любое количество параметров, например:

Листинг MathObjectMinMaxExample01.htm

```
var max = Math.max(3, 54, 32, 16);
alert(max);    // 54
```

```
var min = Math.min(3, 54, 32, 16);
alert(min);    // 3
```

Для группы чисел 3, 54, 32 и 16 методы `Math.max()` и `Math.min()` возвращают 54 и 3 соответственно. С их помощью можно определять максимальные и минимальные значения в группах чисел без циклов и условных инструкций.

Для нахождения максимального или минимального значения в массиве можно использовать метод `apply()`:

```
var values = [1, 2, 3, 4, 5, 6, 7, 8];
var max = Math.max.apply(Math, values);
```

Главное при этом — передать объект `Math` в метод `apply()` как первый аргумент, чтобы задать должным образом значение `this`. После этого можно передать массив как второй аргумент.

Методы округления

В эту группу входят методы `Math.ceil()`, `Math.floor()` и `Math.round()`, которые округляют дробные значения до целых. Все они выполняют округление по-разному:

- метод `Math.ceil()` всегда округляет числа вверх до ближайшего целого;
- метод `Math.floor()` всегда округляет числа вниз до ближайшего целого;
- метод `Math.round()` выполняет стандартное округление (вверх, если дробная часть равна 0,5 или больше, и вниз в противном случае). Это обычный школьный способ округления.

Следующий пример поясняет работу этих методов:

Листинг MathObjectRoundingExample01.htm


```
alert(Math.ceil(25.9));    // 26
alert(Math.ceil(25.5));    // 26
alert(Math.ceil(25.1));    // 26

alert(Math.round(25.9));   // 26
alert(Math.round(25.5));   // 26
alert(Math.round(25.1));   // 25

alert(Math.floor(25.9));   // 25
alert(Math.floor(25.5));   // 25
alert(Math.floor(25.1));   // 25
```

Для всех значений между 25 (не включая) и 26 метод `Math.ceil()` возвращает 26, потому что он округляет число вверх. Метод `Math.round()` возвращает 26, только если число равно 25,5 или больше, в противном случае возвращается 25. Наконец, метод `Math.floor()` возвращает 25 для всех чисел между 25 и 26 (не включая).

Метод random()

Метод `Math.random()` возвращает случайное число от 0 до 1, не включая 0 и 1. Это позволяет, например, выводить случайные цитаты или факты, когда пользователь заходит на веб-сайт. С помощью метода `Math.random()` можно получать случайные числа из определенного целочисленного диапазона по следующей формуле:

```
number = Math.floor(Math.random() * количество_вариантов +
 первое_возможное_значение)
```

Метод `Math.floor()` используется здесь потому, что `Math.random()` всегда возвращает дробное значение, а значит, после умножения этого значения на число и добавления другого числа также получается дробное значение. Если нужно получить число от 1 до 10, можно использовать такой код:

Листинг MathObjectRandomExample01.htm

```
var num = Math.floor(Math.random() * 10 + 1);
```

Обратите внимание, что как первое возможное значение указана единица. Если требуется число от 2 до 10, код будет таким:

Листинг MathObjectRandomExample02.htm

```
var num = Math.floor(Math.random() * 9 + 2);
```

В интервале от 2 до 10 только 9 целых чисел, поэтому общее количество вариантов равно 9, а первым возможным значением является 2. При желании также можно использовать следующую функцию, которая сама определяет общее количество вариантов и первое возможное значение, например:

Листинг MathObjectRandomExample03.htm

```
function selectFrom(lowerValue, upperValue) {
    var choices = upperValue - lowerValue + 1;
    return Math.floor(Math.random() * choices + lowerValue);
}

var num = selectFrom(2,10);
alert(num);    // число от 2 до 10 включительно
```

Функция `selectFrom()` принимает два аргумента: наименьшее и наибольшее значения, которые могут быть возвращены. Для определения количества вариантов функция вычисляет разность этих значений и увеличивает ее на единицу, после чего это значение используется в формуле. Вызов `selectFrom(2,10)` возвращает случайное число от 2 до 10 (включительно). Используя эту функцию, можно легко выбрать случайный элемент массива:

Листинг MathObjectRandomExample03.htm

```
var colors = ["red", "green", "blue", "yellow", "black", "purple", "brown"];
var color = colors[selectFrom(0, colors.length-1)];
```

В этом примере вторым аргументом `selectFrom()` является длина массива за вычетом 1, что соответствует последней позиции в массиве.

Другие методы

Многие методы объекта `Math` служат для выполнения стандартных математических операций. Мы не будем обсуждать их достоинства и недостатки или сценарии их применения, а ограничимся таблицей с краткими описаниями.

Метод	Описание
Math.abs(число)	Возвращает абсолютное значение числа
Math.exp(число)	Возвращает Math.E в степени, заданной числом
Math.log(число)	Возвращает натуральный логарифм числа
Math.pow(число, степень)	Возвращает число в указанной степени
Math.sqrt(число)	Возвращает квадратный корень из числа
Math.acos(x)	Возвращает арккосинус x
Math.asin(x)	Возвращает арксинус x
Math.atan(x)	Возвращает арктангенс x
Math.atan2(y, x)	Возвращает арктангенс y/x
Math.cos(x)	Возвращает косинус x
Math.sin(x)	Возвращает синус x
Math.tan(x)	Возвращает тангенс x

Хотя эти методы определены в ECMA-262, точность вычисления синусов, косинусов и тангенсов зависит от реализации, потому что вычислить их можно многими разными способами.

Резюме

Объекты в JavaScript являются ссылочными значениями. Для создания специфических объектов можно использовать встроенные ссылочные типы.

Ссылочные типы похожи на классы в традиционном объектно-ориентированном программировании, но реализованы иначе.

- Тип `Object` является родительским для всех других ссылочных типов, которые наследуют от него базовые формы поведения.
- Тип `Array` представляет упорядоченный список значений, позволяя выполнять различные операции над ними.
- Тип `Date` служит для работы с датами и значениями времени (в том числе с текущими датой и временем).

- Тип `RegExp` обеспечивает поддержку регулярных выражений, реализуя в основном базовую и отчасти нетривиальную функциональность.

Один из уникальных аспектов JavaScript заключается в том, что функции на самом деле являются экземплярами типа `Function`, то есть объектами. Соответственно, у функций есть методы, которые можно использовать для расширения их поведения.

С помощью оболочек примитивных типов можно работать с примитивными типами в JavaScript так, как если бы они были объектами. Есть три оболочки примитивных типов: `Boolean`, `Number` и `String`. Все они имеют ряд общих характеристик:

- Каждый из типов-оболочек соответствует примитивному типу с тем же именем.
- При чтении примитивного значения создается экземпляр его оболочки, используемый для работы со значением.
- После выполнения инструкции с примитивным значением объект-оболочка уничтожается.

В начале выполнения кода уже существуют встроенные объекты `Global` и `Math`. Объект `Global` недоступен в большинстве ECMAScript-реализаций, но в веб-браузерах он представлен объектом `window`. Все глобальные переменные и функции являются свойствами объекта `Global`. Объект `Math` содержит свойства и методы, помогающие выполнять сложные математические вычисления.

6 Объектно-ориентированное программирование

- Свойства объектов
- Создание объектов
- Наследование

В объектно-ориентированных языках для создания объектов с одинаковыми свойствами и методами обычно используются классы. В ECMAScript классы не поддерживаются, поэтому объекты в нем не такие, как в языках с классами.

В ECMA-262 объект определяется как «неупорядоченная коллекция свойств, каждое из которых содержит примитивное значение, объект или функцию». Строго говоря, это означает, что объект является массивом значений без конкретного порядка. Каждое свойство или метод объекта определяется именем, которое сопоставлено со значением. По этой и ряду других причин, которые мы обсудим позже, полезно рассматривать ECMAScript-объекты как хэш-таблицы — группы пар имен и значений, в которых значениями могут быть данные или функции.

Каждый объект создается на основе ссылочного типа: либо одного из встроенных типов, описанных в предыдущей главе, либо типа, определенного разработчиком.

Общие сведения об объектах

Как было показано в предыдущей главе, самый простой способ определить объект — это создать экземпляр типа `Object` и добавить к нему свойства и методы, например:

Листинг CreatingObjectsExample01.htm


```
var person = new Object();
person.name = "Nicholas";
person.age = 29;
person.job = "Software Engineer";

person.sayName = function(){
    alert(this.name);
};
```

В этом примере создается объект `person` со свойствами `name`, `age` и `job` и методом `sayName()`. Метод выводит значение `this.name`, которое разрешается в `person.name`. Когда-то это был самый популярный способ создания объектов, но теперь для этого обычно используют литералы объектов. С нотацией литералов объектов предыдущий пример можно переписать следующим образом:

```
var person = {
    name: "Nicholas",
    age: 29,
    job: "Software Engineer",

    sayName: function(){
        alert(this.name);
    }
};
```

Этот код эквивалентен предыдущему примеру. Все свойства объектов создаются с определенными характеристиками, от которых зависит их поведение в JavaScript.

Типы свойств

В пятой редакции ECMA-262 характеристики свойств описываются с помощью внутренних атрибутов, которые подлежат реализации в интерпретаторах JavaScript и недоступны непосредственно в JavaScript. То, что атрибут является внутренним, указывают двойные квадратные скобки, например `[[Enumerable]]`. Хотя в третьей редакции ECMA-262 есть некоторые отличия, в этой книге рассматриваются характеристики свойств только из пятой редакции.

Свойства делятся на два типа: свойства с данными и свойства с функциями доступа.

Свойства с данными

Свойства с данными — это места для хранения значений, которые можно читать и записывать. Поведение свойств с данными описывают четыре атрибута:

- `[[Configurable]]` — указывает, можно ли удалить свойство с помощью оператора `delete`, изменить атрибуты свойства или преобразовать его в свойство

с функциями доступа. По умолчанию этот атрибут имеет значение `true` у всех свойств, определенных непосредственно для объекта, как в предыдущем примере.

- `[[Enumerable]]` — указывает, будет ли свойство возвращаться в циклах `for-in`. По умолчанию этот атрибут имеет значение `true` у всех свойств, определенных непосредственно для объекта, как в предыдущем примере.
- `[[Writable]]` — указывает, можно ли изменить значение свойства. По умолчанию этот атрибут имеет значение `true` у всех свойств, определенных непосредственно для объекта, как в предыдущем примере.
- `[[Value]]` — содержит фактические данные. Это место, откуда читается значение свойства и куда сохраняются новые значения. По умолчанию этот атрибут имеет значение `undefined`.

При явном добавлении свойства к объекту (как в предыдущем примере) атрибутам `[[Configurable]]`, `[[Enumerable]]` и `[[Writable]]` присваивается значение `true`, а атрибуту `[[Value]]` — указанное значение, например:

```
var person = {
    name: "Nicholas"
};
```

Здесь создается свойство `name` со значением `"Nicholas"`, то есть атрибут `[[Value]]` свойства становится равен `"Nicholas"` и любые изменения значения сохраняются в этом месте.

Чтобы изменить для свойства предлагаемое по умолчанию значение какого-либо атрибута, в ECMAScript 5 нужно использовать метод `Object.defineProperty()`. Он принимает три аргумента: объект, для которого требуется добавить или изменить свойство, имя свойства и объект-дескриптор. Свойства дескриптора соответствуют именам атрибутов: `configurable`, `enumerable`, `writable` и `value`. Задав какие-либо или все свойства, можно изменить значения соответствующих атрибутов, например:

Листинг DataPropertiesExample01.htm


```
var person = {};
Object.defineProperty(person, "name", {
    writable: false,
    value: "Nicholas"
});

alert(person.name);        // "Nicholas"
person.name = "Greg";
alert(person.name);        // "Nicholas"
```

В этом примере создается доступное только для чтения свойство `name` со значением `"Nicholas"`. Значение такого свойства изменить нельзя. Любые попытки сделать это в нестрогом режиме игнорируются, а в строгом — приводят к ошибке.

Аналогичные правила действуют и при создании неконфигурируемых свойств:

Листинг DataPropertiesExample02.htm

```
var person = {};
Object.defineProperty(person, "name", {
    configurable: false,
    value: "Nicholas"
});

alert(person.name);        // "Nicholas"
delete person.name;
alert(person.name);        // "Nicholas"
```

Присвоение значения `false` атрибуту `configurable` означает, что удалить свойство из объекта невозможно. Выполнение оператора `delete` для этого свойства будет проигнорировано в нестрогом режиме и вызовет ошибку в строгом. Кроме того, свойство, определенное как неконфигурируемое, нельзя снова сделать конфигурируемым. Попытка вызвать метод `Object.defineProperty()` и изменить любой атрибут, кроме `writable`, приведет к ошибке:

Листинг DataPropertiesExample03.htm


```
var person = {};
Object.defineProperty(person, "name", {
    configurable: false,
    value: "Nicholas"
});

// возникает ошибка
Object.defineProperty(person, "name", {
    configurable: true,
    value: "Nicholas"
});
```

Таким образом, метод `Object.defineProperty()` можно вызывать для свойства много раз, но присвоение значения `false` атрибуту `configurable` налагает на это ограничения.

При вызове метода `Object.defineProperty()` атрибуты `configurable`, `enumerable` и `writable` получают по умолчанию значение `false`, если не указано иное. В большинстве случаев мощные возможности метода `Object.defineProperty()` не требуются, но чтобы хорошо разбираться в JavaScript-объектах, важно знать, как он работает.

Метод Object.defineProperty() появился в Internet Explorer 8, но, к сожалению, с очень ограниченными возможностями. Он работает только с DOM-объектами, и создавать с его помощью можно только свойства с функциями доступа. Из-за неполной реализации использовать этот метод в Internet Explorer 8 не рекомендуется.

Свойства с функциями доступа

Свойства с функциями доступа не содержат данные. Вместо этого они содержат функции чтения и записи, хотя обе они не обязательны. При чтении такого свойства

вызывается функция чтения, которая отвечает за возвращение правильного значения. При записи свойства вызывается функция записи с новым значением, которая решает, что делать с данными. Свойства с функциями доступа имеют четыре атрибута:

- `[[Configurable]]` — указывает, можно ли удалить свойство с помощью оператора `delete`, изменить атрибуты свойства или преобразовать его в свойство с данными. По умолчанию этот атрибут имеет значение `true` у всех свойств, определенных непосредственно для объекта.
- `[[Enumerable]]` — указывает, будет ли свойство возвращаться в циклах `for-in`. По умолчанию этот атрибут имеет значение `true` у всех свойств, определенных непосредственно для объекта.
- `[[Get]]` — функция, вызываемая при чтении свойства. По умолчанию имеет значение `undefined`.
- `[[Set]]` — функция, вызываемая при записи свойства. По умолчанию имеет значение `undefined`.

Явно определить свойство с функциями доступа невозможно. Для этого нужно использовать метод `Object.defineProperty()`, например:

Листинг AccessorPropertiesExample01.htm


```
var book = {
    _year: 2004,
    edition: 1
};

Object.defineProperty(book, "year", {
    get: function(){
        return this._year;
    },
    set: function(newValue){

        if (newValue > 2004) {
            this._year = newValue;
            this.edition += newValue - 2004;
        }
    }
});

book.year = 2005;
alert(book.edition);    // 2
```

В этом фрагменте создается объект `book` с двумя свойствами, предлагаемыми по умолчанию: `_year` и `edition`. Знак подчеркивания в имени `_year` — это популярная нотация, указывающая, что значение не предполагается использовать вне методов объекта. Далее определяется свойство `year` с функциями доступа, у которого функция чтения просто возвращает значение `_year`, а функция записи определяет редакцию книги, выполняя некоторые вычисления. Присвоение значения 2005

свойству year изменяет _year на 2005, а edition на 2. Это типичный сценарий применения свойств с функциями доступа, когда задание свойства влечет за собой другие изменения.

Определять обе функции доступа не требуется. Если свойству назначена только функция чтения, это означает, что оно не поддерживает запись. В нестрогом режиме любые попытки записи такого свойства игнорируются, а в строгом приводят к ошибке. Аналогичным образом при чтении свойства, у которого есть только функция записи, в нестрогом режиме возвращается значение undefined, а в строгом возникает ошибка.

Создание свойств с функциями доступа в стиле ECMAScript 5 поддерживается в Internet Explorer 9+ (в Internet Explorer 8 этот способ реализован частично), Firefox 4+, Safari 5+, Opera 12+ и Chrome, но раньше для этого использовались нестандартные методы __defineGetter__() и __defineSetter__(). Они были представлены в Firefox и позднее скопированы в Safari 3, Chrome 1 и Opera 9.5. С помощью этих унаследованных методов предыдущий пример можно переписать так:

Листинг AccessorPropertiesExample02.htm


```
var book = {
    _year: 2004,
    edition: 1
};

// унаследованные функции доступа
book.__defineGetter__("year", function(){
    return this._year;
});

book.__defineSetter__("year", function(newValue){
    if (newValue > 2004) {
        this._year = newValue;
        this.edition += newValue - 2004;
    }
});

book.year = 2005;
alert(book.edition);    // 2
```

В браузерах, которые не поддерживают метод Object.defineProperty(), изменить атрибуты [[Configurable]] и [[Enumerable]] невозможно.

Определение нескольких свойств

Чтобы для объекта можно было определить сразу несколько свойств, ECMAScript 5 предоставляет метод Object.defineProperties(), принимающий два аргумента: объект, для которого нужно добавить или изменить свойства, и объект с новыми свойствами. Рассмотрим пример:

Листинг MultiplePropertiesExample01.htm

```
var book = {};

Object.defineProperties(book, {
    _year: {
        value: 2004
    },

    edition: {
        value: 1
    },

    year: {
        get: function(){
            return this._year;
        },

        set: function(newValue){
            if (newValue > 2004) {
                this._year = newValue;
                this.edition += newValue - 2004;
            }
        }
    }
});
```

В этом коде для объекта `book` определяются свойства с данными `_year` и `edition` и свойство с функциями доступа `year`. Итоговый объект идентичен объекту из предыдущего раздела. Единственное отличие этого примера в том, что в нем создаются сразу все свойства.

Метод `Object.defineProperties()` поддерживается в Internet Explorer 9+, Firefox 4+, Safari 5+, Opera 12+ и Chrome.

Чтение атрибутов свойств

Получить дескриптор конкретного свойства можно с помощью метода `Object.getOwnPropertyDescriptor()` из ECMAScript 5. Он принимает два аргумента: объект со свойством, дескриптор которого нужно получить, и имя этого свойства. Метод возвращает объект со свойствами `configurable`, `enumerable`, `get` и `set` в случае свойств с функциями доступа или `configurable`, `enumerable`, `writable` и `value` в случае свойств с данными, например:

Листинг GetPropertyDescriptorExample01.htm


```
var book = {};

Object.defineProperties(book, {
    _year: {
        value: 2004
    },
```

```
    edition: {
        value: 1
    },

    year: {
        get: function(){
            return this._year;
        },

        set: function(newValue){
            if (newValue > 2004) {
                this._year = newValue;
                this.edition += newValue - 2004;
            }
        }
    }
});
var descriptor = Object.getOwnPropertyDescriptor(book, "_year");
alert(descriptor.value);            // 2004
alert(descriptor.configurable);     // false
alert(typeof descriptor.get);       // "undefined"

var descriptor = Object.getOwnPropertyDescriptor(book, "year");
alert(descriptor.value);            // undefined
alert(descriptor.enumerable);       // false
alert(typeof descriptor.get);       // "function"
```

Для свойства с данными `_year` возвращается дескриптор, у которого свойство `value` имеет первоначальное значение, `configurable` — значение `false`, а `get` — значение `undefined`. У дескриптора свойства с функциями доступа `year` свойство `value` имеет значение `undefined`, `enumerable` — значение `false`, а свойство `get` является указателем на функцию чтения.

Метод `Object.getOwnPropertyDescriptor()` можно использовать в JavaScript с любыми объектами, включая DOM- и BOM-объекты. Он поддерживается в Internet Explorer 9+, Firefox 4+, Safari 5+, Opera 12+ и Chrome.

Создание объектов

Создать один объект можно с помощью конструктора `Object` или литерала объекта, но для создания многих объектов с одинаковым интерфейсом эти способы не подходят из-за повторения кода. Для решения этой проблемы разработчики используют разновидность паттерна Фабрика.

Паттерн Фабрика

Паттерн Фабрика (factory pattern) — это широко известный паттерн проектирования, который используется при разработке ПО для абстрагирования

процесса создания специфических объектов (другие паттерны проектирования и их JavaScript-реализации мы обсудим чуть позже). Из-за того что определять классы в ECMAScript невозможно, разработчики стали инкапсулировать создание объектов со специфическими интерфейсами в функциях, например:

Листинг FactoryPatternExample01.htm


```
function createPerson(name, age, job){
    var o = new Object();
    o.name = name;
    o.age = age;
    o.job = job;
    o.sayName = function(){
        alert(this.name);
    };
    return o;
}

var person1 = createPerson("Nicholas", 29, "Software Engineer");
var person2 = createPerson("Greg", 27, "Doctor");
```

Здесь функция `createPerson()` принимает в качестве аргументов все сведения, необходимые для создания объекта `Person`. Функцию можно вызывать любое количество раз с разными аргументами, и каждый раз она будет возвращать объект с тремя свойствами и одним методом. Это решает проблему создания многих похожих объектов, однако паттерн Фабрика не позволяет узнать тип объекта, поэтому по мере развития JavaScript появился новый паттерн.

Паттерн Конструктор

Как уже отмечалось, конструкторы в ECMAScript используются для создания объектов специфических типов. Конструкторы встроенных типов, таких как `Object` и `Array`, автоматически становятся доступны в среде во время выполнения. Также можно определять конструкторы со свойствами и методами для собственных типов. Так, предыдущий пример с использованием *паттерна Конструктор* (constructor pattern) можно переписать следующим образом:

Листинг ConstructorPatternExample01.htm


```
function Person(name, age, job){
    this.name = name;
    this.age = age;
    this.job = job;
    this.sayName = function(){
        alert(this.name);
    };
}

var person1 = new Person("Nicholas", 29, "Software Engineer");
var person2 = new Person("Greg", 27, "Doctor");
```

В этом примере функция `Person()` заменяет фабричную функцию `createPerson()`. Новая функция отличается от прежней следующими аспектами:

- объект явно не создается;
- свойства и метод назначаются непосредственно объекту `this`;
- инструкция `return` отсутствует.

Обратите также внимание на то, что имя функции `Person` начинается с прописной буквы. Имена конструкторов всегда начинаются с прописной буквы, а имена обычных функций — со строчной. Эта соглашение позаимствовано в других объектно-ориентированных языках, чтобы было проще различать ECMAScript-функции по способам их применения, так как конструкторы — это просто функции, которые создают объекты.

Для создания экземпляров `Person` используется оператор `new`. В результате выполняются четыре действия:

1. Создание объекта.
2. Назначение нового объекта переменной `this` конструктора (после чего `this` указывает на новый объект).
3. Выполнение кода внутри конструктора (добавление свойств к новому объекту).
4. Возвращение нового объекта.

В конце предыдущего примера переменные `person1` и `person2` создаются как разные экземпляры `Person`. У каждого из этих объектов есть свойство `constructor`, указывающее на `Person`:

```
alert(person1.constructor == Person);    // true
alert(person2.constructor == Person);    // true
```

Изначально свойство `constructor` предназначалось для идентификации типа объектов, однако считается, что для этого безопаснее использовать оператор `instanceof`. Как показывает следующий код, оба объекта в примере являются экземплярами типов `Object` и `Person`:

```
alert(person1 instanceof Object);    // true
alert(person1 instanceof Person);    // true
alert(person2 instanceof Object);    // true
alert(person2 instanceof Person);    // true
```

Определение собственных конструкторов позволяет позднее узнавать типы объектов, созданных с их помощью, что является серьезным преимуществом по сравнению с паттерном Фабрика. В последнем примере объекты `person1` и `person2` считаются экземплярами `Object`, потому что все пользовательские объекты наследуют от типа `Object` (детали см. далее).

Конструкторы, определяемые таким образом, связываются с объектом Global (window в веб-браузерах). Объектная модель браузера (BOM) обсуждается в главе 8.

Конструкторы как функции

Единственным отличием конструкторов от других функций является способ их вызова, потому что определяются все функции одинаково. Любая функция, вызванная с помощью оператора `new`, работает как конструктор, а функция, вызванная без `new`, ведет себя обычным образом. Например, функцию `Person()` из предыдущего примера можно вызвать следующими способами:

Листинг ConstructorPatternExample02.htm


```
// вызов в качестве конструктора
var person = new Person("Nicholas", 29, "Software Engineer");
person.sayName();    // "Nicholas"

// вызов в качестве обычной функции
Person("Greg", 27, "Doctor");        // функция добавляется к window
window.sayName();    // "Greg"

// вызов в области видимости другого объекта
var o = new Object();
Person.call(o, "Kristen", 25, "Nurse");
o.sayName();        // "Kristen"
```

В первой части примера показано типичное применение конструктора, а именно создание объекта с помощью оператора `new`. Во второй части функция `Person()` вызывается без оператора `new`, в результате свойства и методы добавляются к объекту `window`. Запомните, что если функция вызывается без явно заданного значения `this` (то есть не как метод объекта и без метода `call()` или `apply()`), значение `this` указывает на объект `Global` (`window` в веб-браузерах). Таким образом, вызов метода `sayName()` для объекта `window` возвращает значение `"Greg"`. Функцию `Person()` также можно вызвать в области видимости конкретного объекта, используя метод `call()` или `apply()`. В третьей части в качестве `this` указан объект `o`, которому и назначаются все свойства и метод `sayName()`.

Проблемы конструкторов

Хотя концепция конструктора полезна, она не лишена недостатков. Главный из них заключается в том, что методы создаются для каждого экземпляра. Так, в предыдущем примере и у `person1`, и у `person2` есть метод с именем `sayName()`, но эти методы — не один и тот же экземпляр `Function`. Функции в ECMAScript являются объектами, так что при каждом определении функции создается объект. Логически конструктор на самом деле выглядит так:

```
function Person(name, age, job){
    this.name = name;
    this.age = age;
```

```
    this.job = job;
    this.sayName = new Function("alert(this.name)"); //логический эквивалент
}
```

При таком взгляде на конструктор очевидно, что каждый экземпляр `Person` получает свой экземпляр функции, выводящей на экран свойство `name`. Строго говоря, такое создание функции уникально в плане цепочек областей видимости и разрешения идентификаторов, но технически новый экземпляр `Function` создается обычным образом. В общем, одноименные функции в разных экземплярах не эквивалентны, что подтверждает следующий код:

```
alert(person1.sayName == person2.sayName);        // false
```

Не имеет смысла создавать два экземпляра `Function`, делающих одно и то же, особенно если учесть, что благодаря объекту `this` можно отложить привязку функций к конкретным объектам до выполнения кода. Чтобы обойти это ограничение, можно вынести определение функции за пределы конструктора:

Листинг ConstructorPatternExample03.htm


```
function Person(name, age, job){
    this.name = name;
    this.age = age;
    this.job = job;
    this.sayName = sayName;
}

function sayName(){
    alert(this.name);
}

var person1 = new Person("Nicholas", 29, "Software Engineer");
var person2 = new Person("Greg", 27, "Doctor");
```

В этом примере функция `sayName()` определена вне конструктора в глобальной области видимости и назначается внутри конструктора свойству `sayName`. Поскольку свойство `sayName` теперь содержит лишь указатель на функцию, она становится общей для объектов `person1` и `person2`. Это решает проблему повторяющихся функций, но при этом засоряет глобальную область видимости функцией, которая на самом деле используется только в связи с объектами. Если у объекта будет много методов, нам придется создать много глобальных функций и определение нашего ссылочного типа перестанет быть аккуратной группой связанных инструкций. Эту проблему устраняет *паттерн Прототип* (prototype pattern).

Паттерн Прототип

Каждая функция создается со свойством `prototype` — объектом, содержащим свойства и методы, которые должны быть доступны в экземплярах конкретного ссылочного типа. Этот объект в буквальном смысле является прототипом для объекта,

создаваемого при вызове конструктора. Преимущество использования прототипа в том, что все его свойства и методы общие для объектов. Вместо того чтобы назначать атрибуты объекту в конструкторе, их можно назначить непосредственно прототипу, например:

Листинг PrototypePatternExample01.htm

```
function Person(){
}

Person.prototype.name = "Nicholas";
Person.prototype.age = 29;
Person.prototype.job = "Software Engineer";
Person.prototype.sayName = function(){
    alert(this.name);
};

var person1 = new Person();
person1.sayName();        // "Nicholas"

var person2 = new Person();
person2.sayName();        // "Nicholas"

alert(person1.sayName == person2.sayName);        // true
```

Здесь свойства и метод `sayName()` добавляются непосредственно к свойству `prototype` объекта `Person`, а конструктор остается пустым. Тем не менее конструктор можно вызывать для создания объектов, и у них будут доступны все свойства и методы. В отличие от паттерна Конструктор, свойства и методы Прототипа являются общими для экземпляров, так что объекты `person1` и `person2` имеют один набор свойств и одну функцию `sayName()` на двоих. Чтобы понять, как все это работает, нужно подробнее обсудить ECMAScript-прототипы.

Подробности работы прототипов

Когда создается функция, по определенным правилам создается также ее свойство `prototype`. По умолчанию во все прототипы автоматически добавляется свойство `constructor`, указывающее на функцию, к которой оно относится (так, в предыдущем примере свойство `Person.prototype.constructor` указывает на функцию `Person`). Затем в зависимости от конструктора в прототип могут быть добавлены другие свойства и методы.

При определении пользовательского конструктора в прототип добавляется по умолчанию только свойство `constructor`, а все остальные методы наследуются от типа `Object`. Когда с помощью конструктора создается новый экземпляр типа, в экземпляре определяется внутренний указатель на прототип конструктора. В пятой редакции ECMA-262 этот указатель называется `[[Prototype]]`. Стандартного способа доступа к нему из сценариев нет, но в Firefox, Safari и Chrome у каждого объекта есть свойство `__proto__`, которое в других браузерах полностью скрыто от JavaScript-сценариев. Важно отметить наличие непосредственной связи между экземпляром и прототипом конструктора, но не между экземпляром и конструктором.

Связи между объектами в примере с конструктором `Person` и свойством `Person.prototype` показаны на рис. 6.1.

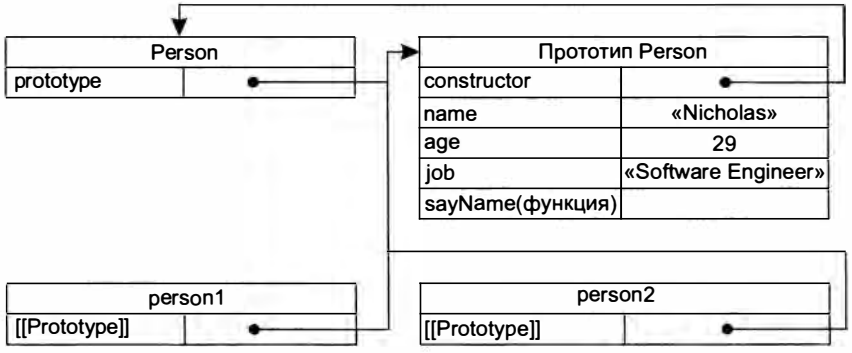

Рис. 6.1

На рисунке изображены конструктор `Person`, прототип `Person` и два экземпляра типа `Person`. Обратите внимание на то, что свойство `Person.prototype` указывает на объект прототипа, а `Person.prototype.constructor` — на функцию `Person`. Прототип содержит свойство `constructor` и другие свойства, которые были добавлены. У каждого экземпляра `Person` (`person1` и `person2`) есть внутренние свойства, указывающие только на свойство `Person.prototype`, но непосредственной связи с конструктором у экземпляров нет. Отметим также, что вызов `person1.sayName()` выполняется нормально, хотя у самих экземпляров нет ни свойств, ни методов. Это возможно благодаря процедуре поиска свойств объекта.

Несмотря на то что указатель `[[Prototype]]` доступен не во всех браузерах, можно проверить наличие связи между объектом и прототипом с помощью метода `isPrototypeOf()`. Он возвращает `true`, если указатель `[[Prototype]]` экземпляра указывает на прототип, для которого вызван метод:

```
alert(Person.prototype.isPrototypeOf(person1));    // true
alert(Person.prototype.isPrototypeOf(person2));    // true
```

В этом примере метод `isPrototypeOf()` прототипа вызывается для экземпляров `person1` и `person2`. Поскольку у обоих есть ссылка на `Person.prototype`, метод возвращает `true`.

В ECMAScript 5 добавлен новый метод `Object.getPrototypeOf()`, который во всех поддерживающих его браузерах возвращает значение `[[Prototype]]`, например:

```
alert(Object.getPrototypeOf(person1) == Person.prototype);  // true
alert(Object.getPrototypeOf(person1).name);                 // "Nicholas"
```

Первая строка этого фрагмента просто подтверждает, что объект, возвращенный методом `Object.getPrototypeOf()`, на самом деле является прототипом объекта.

Вторая инструкция читает свойство `name` прототипа, возвращая строку `"Nicholas"`. С помощью метода `Object.getPrototypeOf()` можно легко получить прототип объекта, что важно для реализации наследования на основе прототипов (см. следующую главу). Этот метод поддерживается в Internet Explorer 9+, Firefox 3.5+, Safari 5+, Opera 12+ и Chrome.

При чтении свойства объекта начинается его поиск. Сначала свойство с указанным именем ищется в самом экземпляре объекта. Если оно обнаруживается у экземпляра, возвращается значение свойства; если его у экземпляра нет, поиск продолжается в прототипе. В случае обнаружения свойства у прототипа возвращается его значение. Так, при вызове `person1.sayName()` интерпретатор JavaScript сначала выясняет, есть ли свойство с именем `sayName` у экземпляра `person1`. Его нет, поэтому интерпретатор ищет свойство `sayName` в прототипе `person1`. В этот раз свойство доступно, поэтому интерпретатор вызывает функцию `sayName()`, связанную с прототипом. При вызове `person2.sayName()` выполняется аналогичная процедура, которая завершается таким же результатом. Так прототипы обеспечивают совместное использование свойств и методов несколькими экземплярами объектов.

Упомянутое свойство constructor есть только у прототипа и доступно из экземпляров объектов.

Хотя в экземплярах объектов можно читать значения из прототипа, перезаписать их нельзя. Если добавить к экземпляру свойство с именем, как у свойства прототипа, последнее будет скрыто, например:

Листинг PrototypePatternExample02.htm


```
function Person(){
}

Person.prototype.name = "Nicholas";
Person.prototype.age = 29;
Person.prototype.job = "Software Engineer";
Person.prototype.sayName = function(){
    alert(this.name);
};

var person1 = new Person();
var person2 = new Person();

person1.name = "Greg";
alert(person1.name);    // "Greg" - из экземпляра
alert(person2.name);    // "Nicholas" - из прототипа
```

В этом примере свойство `name` экземпляра `person1` затеняется новым значением. Оба свойства — и `person1.name`, и `person2.name` — работают нормально, возвращая `"Greg"` (из экземпляра объекта) и `"Nicholas"` (из прототипа) соответственно. При чтении значения `person1.name` в методе `alert()` начинается поиск свойства с именем `name` у экземпляра. Поскольку такое свойство есть у экземпляра, оно используется,

а поиск в прототипе не выполняется. У экземпляра person2 свойства name нет, поэтому поиск продолжается в прототипе и используется свойство name прототипа.

Как только свойство добавлено к экземпляру объекта, оно *затеняет* (shadows) все одноименные свойства прототипа, то есть блокирует их, не изменяя прототип. Даже если присвоить свойству экземпляра значение null, ссылка на прототип не восстанавливается. Оператор delete полностью удаляет свойство экземпляра, делая свойство прототипа доступным:

Листинг PrototypePatternExample03.htm

```
function Person(){
}

Person.prototype.name = "Nicholas";
Person.prototype.age = 29;
Person.prototype.job = "Software Engineer";
Person.prototype.sayName = function(){
    alert(this.name);
};

var person1 = new Person();
var person2 = new Person();

person1.name = "Greg";
alert(person1.name);    // "Greg" - из экземпляра
alert(person2.name);    // "Nicholas" - из прототипа

delete person1.name;
alert(person1.name);    // "Nicholas" - из прототипа
```

Здесь оператор delete вызывается для свойства person1.name, которое ранее было затенено значением "Greg". Это восстанавливает ссылку на свойство name прототипа, так что при следующем доступе к person1.name возвращается значение свойства прототипа.

Метод hasOwnProperty(), унаследованный от типа Object, позволяет выяснить, принадлежит свойство экземпляру или прототипу. Он возвращает true, только если свойство с указанным именем есть у экземпляра, например:

```
function Person(){
}

Person.prototype.name = "Nicholas";
Person.prototype.age = 29;
Person.prototype.job = "Software Engineer";
Person.prototype.sayName = function(){
    alert(this.name);
};

var person1 = new Person();
var person2 = new Person();
```

```
alert(person1.hasOwnProperty("name"));        // false

person1.name = "Greg";
alert(person1.name);    // "Greg" - из экземпляра
alert(person1.hasOwnProperty("name"));        // true

alert(person2.name);    // "Nicholas" - из прототипа
alert(person2.hasOwnProperty("name"));        // false

delete person1.name;
alert(person1.name);    // "Nicholas" - из прототипа
alert(person1.hasOwnProperty("name"));        // false
```

Метод `hasOwnProperty()` в этом примере ясно показывает, когда используется свойство экземпляра, а когда — свойство прототипа. Вызов `person1.hasOwnProperty("name")` возвращает `true` только после перезаписи свойства `name` для экземпляра `person1` (это указывает, что используется свойство экземпляра, а не прототипа). Рисунок 6.2 поясняет выполненные в примере действия (ради простоты связь с конструктором `Person` опущена).

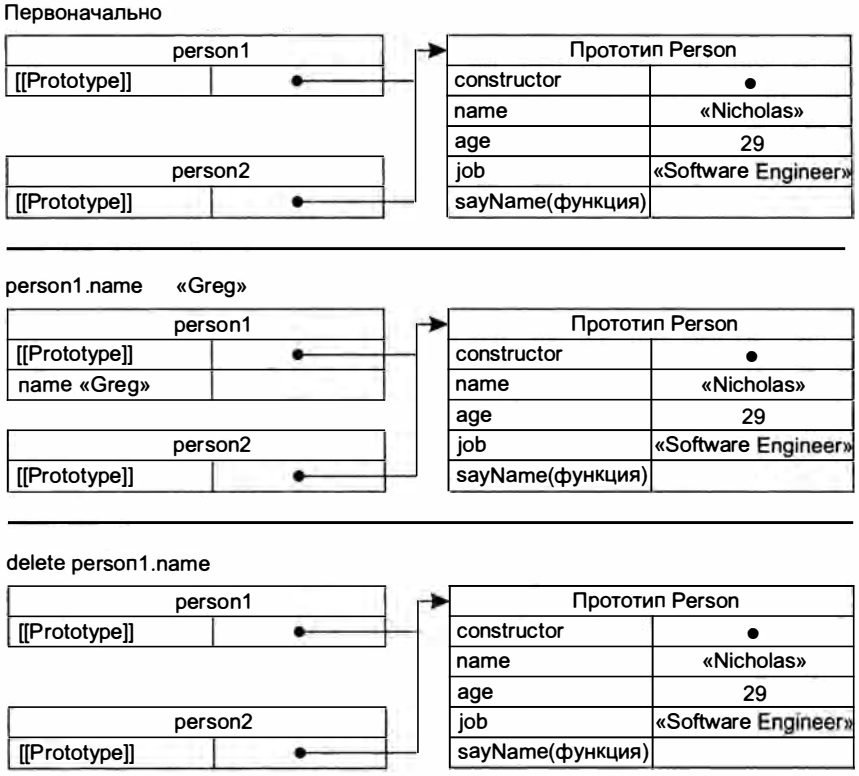

Рис. 6.2

Метод Object.getOwnPropertyDescriptor() из ECMAScript 5 работает только со свойствами экземпляра; чтобы получить дескриптор свойства прототипа, необходимо вызвать метод Object.getOwnPropertyDescriptor() непосредственно для объекта прототипа.

Прототипы и оператор in

Оператор `in` можно использовать отдельно или в цикле `for-in`. В первом случае он возвращает `true`, если свойство с указанным именем имеется у экземпляра или у прототипа объекта. Рассмотрим пример:

Листинг PrototypePatternExample04.htm


```
function Person(){
}

Person.prototype.name = "Nicholas";
Person.prototype.age = 29;
Person.prototype.job = "Software Engineer";
Person.prototype.sayName = function(){
    alert(this.name);
};

var person1 = new Person();
var person2 = new Person();

alert(person1.hasOwnProperty("name"));        // false
alert("name" in person1);  // true

person1.name = "Greg";
alert(person1.name);       // "Greg" – из экземпляра
alert(person1.hasOwnProperty("name"));        // true
alert("name" in person1);  // true

alert(person2.name);       // "Nicholas" – из прототипа
alert(person2.hasOwnProperty("name"));        // false
alert("name" in person2);   // true

delete person1.name;
alert(person1.name);        // "Nicholas" – из прототипа
alert(person1.hasOwnProperty("name"));        // false
alert("name" in person1);   // true
```

Во всем этом фрагменте свойство `name` доступно у каждого объекта непосредственно или в прототипе. Таким образом, выражение `"name" in person1` всегда возвращает `true` независимо от того, принадлежит ли свойство именно экземпляру. Объединив вызов метода `hasOwnProperty()` с оператором `in`, можно выяснить, принадлежит ли свойство прототипу:

```
function hasPrototypeProperty(object, name){
    return !object.hasOwnProperty(name) && (name in object);
}
```

Поскольку оператор `in` возвращает `true`, если свойство так или иначе доступно через объект, а метод `hasOwnProperty()` возвращает `true`, только если свойство имеется у экземпляра, свойство прототипа можно узнать по тому, что оператор `in` возвращает `true`, а метод `hasOwnProperty()` — `false`, например:

Листинг PrototypePatternExample05.htm


```
function Person(){
}

Person.prototype.name = "Nicholas";
Person.prototype.age = 29;
Person.prototype.job = "Software Engineer";
Person.prototype.sayName = function(){
    alert(this.name);
};

var person = new Person();
alert(hasPrototypeProperty(person, "name"));    // true

person.name = "Greg";
alert(hasPrototypeProperty(person, "name"));    // false
```

В этом коде свойство `name` сначала определено у прототипа, поэтому метод `hasPrototypeProperty()` возвращает `true`. После перезаписи свойства оно относится к экземпляру, и метод `hasPrototypeProperty()` возвращает `false`. Да, свойство `name` есть и у прототипа, но оно больше не используется, потому что затенено свойством экземпляра.

В цикле `for-in` возвращаются все свойства экземпляра и прототипа, которые доступны через объект и могут быть перечислены. Свойства экземпляра, затеняющие неперечислимые свойства прототипа (неперечислимые свойства — это свойства, у которых атрибут `[[Enumerable]]` имеет значение `false`), также возвращаются в цикле `for-in`, потому что все свойства, определенные разработчиком, перечислимы во всех браузерах, за исключением Internet Explorer 8 и более ранних версий.

В старых реализациях Internet Explorer имеется дефект, из-за которого свойства, затеняющие неперечислимые свойства, не возвращаются в цикле `for-in`, например:

Листинг PrototypePatternExample06.htm


```
var o = {
    toString : function(){
        return "My Object";
    }
};

for (var prop in o){
    if (prop == "toString"){
        alert("Found toString");    // не выводится в IE
    }
}
```

У объекта o есть свойство экземпляра `toString()`, которое затеняет неперечислимый метод `toString()` прототипа, поэтому при выполнении этого кода должно появиться оповещение о том, что обнаружен метод `toString()`. Однако в Internet Explorer это оповещение не появляется, потому что браузер пропускает свойство, учитывая значение атрибута `[[Enumerable]]` метода `toString()` прототипа. Этот дефект влияет на все свойства и методы, которые неперечислимы по умолчанию, а именно `hasOwnProperty()`, `propertyIsEnumerable()`, `toLocaleString()`, `toString()` и `valueOf()`. В ECMAScript 5 атрибут `[[Enumerable]]` устанавливается в `false` также для свойств `constructor` и `prototype`, но это зависит от реализации.

Получить список всех перечислимых свойств экземпляра в ECMAScript 5 можно с помощью метода `Object.keys()`, который принимает объект и возвращает массив с именами соответствующих свойств, например:

Листинг ObjectKeysExample01.htm


```
function Person(){
}

Person.prototype.name = "Nicholas";
Person.prototype.age = 29;
Person.prototype.job = "Software Engineer";
Person.prototype.sayName = function(){
    alert(this.name);
};

var keys = Object.keys(Person.prototype);
alert(keys);        // "name,age,job,sayName"

var p1 = new Person();
p1.name = "Rob";
p1.age = 31;
var p1keys = Object.keys(p1);
alert(p1keys);    // "name,age"
```

Здесь переменной `keys` назначается массив со строками `"name"`, `"age"`, `"job"` и `"sayName"`. В таком порядке они отображались бы и в цикле `for-in`. При вызове для экземпляра `Person` метод `Object.keys()` возвращает массив со строками `name` и `age`, двумя свойствами экземпляра.

Если вам нужен список всех свойств экземпляра, перечислимых и неперечислимых, можно аналогичным образом использовать метод `Object.getOwnPropertyNames()`:

Листинг ObjectPropertyNamesExample01.htm

```
var keys = Object.getOwnPropertyNames(Person.prototype);
alert(keys);        // "constructor,name,age,job,sayName"
```

Обратите внимание на то, что список результатов содержит неперечислимое свойство `constructor`. Методы `Object.keys()` и `Object.getOwnPropertyNames()` могут быть полезны как альтернатива циклу `for-in`. Они поддерживаются в Internet Explorer 9+, Firefox 4+, Safari 5+, Opera 12+ и Chrome.

Альтернативный синтаксис прототипов

В предыдущем примере имя `Person.prototype` нужно было вводить при добавлении каждого свойства и метода. Чтобы писать меньше кода и нагляднее группировать элементы прототипов, чаще просто перезаписывают прототип литералом объекта, содержащим все свойства и методы, например:

Листинг PrototypePatternExample07.htm


```
function Person(){
}

Person.prototype = {
    name : "Nicholas",
    age : 29,
    job : "Software Engineer",
    sayName : function () {
        alert(this.name);
    }
};
```

В этом примере свойству `Person.prototype` присваивается новый объект, созданный с помощью литерала. Результат получается таким же за одним исключением: свойство `constructor` больше не указывает на объект `Person`. При создании функции создается ее объект `prototype` и автоматически задается свойство `constructor`. Новый синтаксис полностью перезаписывает объект `prototype`, предлагаемый по умолчанию, после чего свойство `constructor` указывает на совершенно новый объект (конструктор `Object`), а не на саму функцию. Хотя оператор `instanceof` по-прежнему работает, полагаться на то, что свойство `constructor` показывает тип объекта, больше нельзя:

Листинг PrototypePatternExample07.htm

```
var friend = new Person();
alert(friend instanceof Object);        // true
alert(friend instanceof Person);        // true
alert(friend.constructor == Person);    // false
alert(friend.constructor == Object);    // true
```

Здесь оператор `instanceof` все еще возвращает `true` и для `Object`, и для `Person`, однако свойство `constructor` теперь имеет значение `Object`, а не `Person`. Если значение `constructor` важно, можно явно восстановить его:

Листинг PrototypePatternExample08.htm

```
function Person(){
}

Person.prototype = {
    constructor: Person,
    name : "Nicholas",
    age : 29,
```

```
    job : "Software Engineer",
    sayName : function () {
        alert(this.name);
    }
};
```

Добавление свойства `constructor` со значением `Person` в литерал объекта решает проблему.

Помните, что у конструктора, восстановленного таким способом, атрибут `[[Enumerable]]` имеет значение `true`. Свойство `constructor` встроенных объектов по умолчанию неперечислимо, поэтому если ваш интерпретатор JavaScript совместим с ECMAScript 5, возможно, лучше использовать с той же целью метод `Object.defineProperty()`:

```
function Person(){
}

Person.prototype = {
    name : "Nicholas",
    age : 29,
    job : "Software Engineer",
    sayName : function () {
        alert(this.name);
    }
};

// восстановление конструктора (только в ECMAScript 5)
Object.defineProperty(Person.prototype, "constructor", {
    enumerable: false,
    value: Person
});
```

Динамическая природа прототипов

Поскольку при доступе к значению, которого нет в экземпляре, выполняется его поиск в прототипе, изменения прототипа немедленно отражаются на экземплярах, в том числе на тех, которые существовали до изменения, например:

Листинг PrototypePatternExample09.htm


```
var friend = new Person();

Person.prototype.sayHi = function(){
    alert("hi");
};

friend.sayHi(); // "hi" - все работает!
```

В этом коде создается экземпляр `Person`, который сохраняется в переменной `friend`. Затем к свойству `Person.prototype` добавляется метод `sayHi()`. Несмотря на то что экземпляр `friend` был создан до этого изменения, новый метод доступен ему

благодаря связи с прототипом. При вызове `friend.sayHi()` сначала выполняется поиск свойства `sayHi` в экземпляре; когда выясняется, что его в экземпляре нет, поиск продолжается в прототипе. Так как экземпляр лишь связан с прототипом с помощью указателя, а не является его копией, интерпретатор обнаруживает в прототипе новое свойство `sayHi` и возвращает назначенную ему функцию.

Хотя свойства и методы, добавленные в прототип, немедленно становятся доступны во всех экземплярах объектов, при перезаписи всего прототипа наблюдается другое поведение. Указатель `[[Prototype]]` задается при вызове конструктора, поэтому изменение прототипа на другой объект нарушает связь между конструктором и оригинальным прототипом. Помните, у экземпляра есть указатель только на прототип, но не на конструктор. Рассмотрим следующий пример:

Листинг PrototypePatternExample10.htm


```
function Person(){
}

var friend = new Person();

Person.prototype = {
    constructor: Person,
    name : "Nicholas",
    age : 29,
    job : "Software Engineer",
    sayName : function () {
        alert(this.name);
    }
};

friend.sayName();        // ошибка
```

В этом примере создается экземпляр типа `Person`, а затем объект его прототипа перезаписывается. При вызове метода `friend.sayName()` возникает ошибка, потому что у прототипа, на который указывает экземпляр `friend`, нет свойства с таким именем. Рисунок 6.3 поясняет, почему это происходит.

Перезапись прототипа у конструктора приводит к тому, что новые экземпляры ссылаются на новый прототип, а уже существующие — на старый.

Прототипы встроенных объектов

Паттерн Прототип важен не только для определения пользовательских типов, этот паттерн может также служить для реализации всех встроенных ссылочных типов. В каждом из этих типов (включая `Object`, `Array`, `String` и т. д.) методы определены в прототипе конструктора. Например, метод `sort()` принадлежит свойству `Array.prototype`, а `substring()` — свойству `String.prototype`:

```
alert(typeof Array.prototype.sort);           // "function"
alert(typeof String.prototype.substring);     // "function"
```

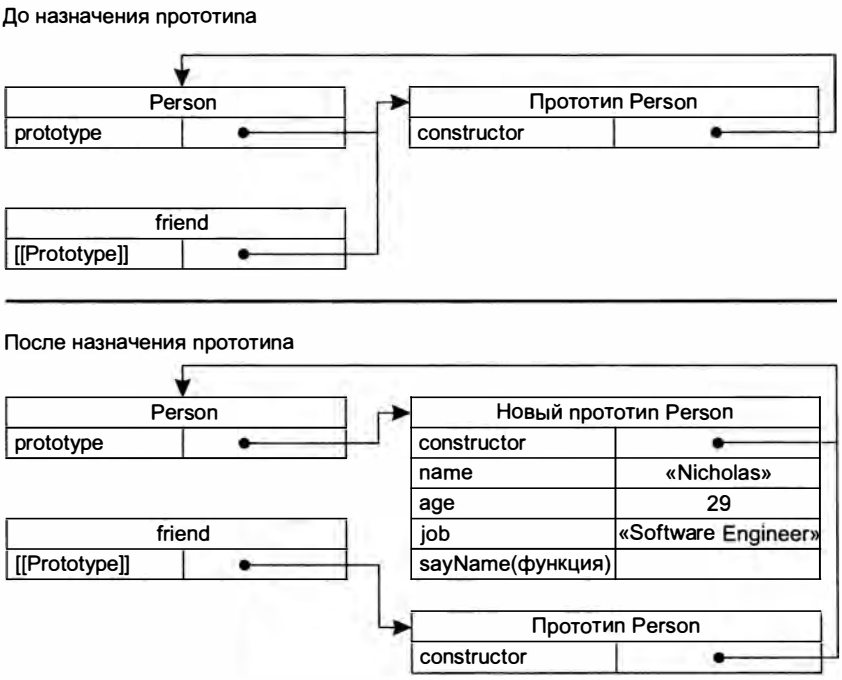

Рис. 6.3

С помощью прототипов встроенных объектов можно получать ссылки на методы, предлагаемые по умолчанию, и определять новые методы. Кроме того, эти прототипы можно изменять, что позволяет добавлять методы к встроенным объектам. Например, следующий код добавляет метод `startsWith()` к оболочке примитивного типа `String`:

Листинг PrototypePatternExample11.htm


```
String.prototype.startsWith = function (text) {
    return this.indexOf(text) == 0;
};

var msg = "Hello world!";
alert(msg.startsWith("Hello"));    // true
```

В этом примере метод `startsWith()` возвращает `true`, если переданный в него текст совпадает с началом строки. Поскольку метод назначен свойству `String.prototype`, его можно использовать со всеми строками в среде. Чтобы метод `startsWith()` стал доступен для строки `msg`, для нее неявно создается оболочка примитивного типа `String`.

Изменять прототипы встроенных объектов в окончательном коде не рекомендуется. Это затрудняет понимание кода и может вызывать конфликты имен, когда метод, не являющийся встроенным в одном браузере, является таковым в другом. Кроме того, так можно случайно перезаписать встроенный метод.

Проблемы прототипов

К сожалению, у паттерна Прототип есть недостатки. Прежде всего, он не позволяет передать в конструктор аргументы инициализации, из-за чего свойства всех экземпляров имеют по умолчанию одинаковые значения. Это неудобно, но главная проблема прототипов связана с тем, что многие экземпляры используют их совместно.

Все свойства прототипа являются общими для всех экземпляров, что идеально для функций. Свойства с примитивными значениями также работают нормально, потому что свойство прототипа можно затенить, создав в экземпляре одноименное свойство. Реальная проблема возникает, если свойство содержит ссылочное значение. Рассмотрим пример:

Листинг PrototypePatternExample12.htm


```
function Person(){
}

Person.prototype = {
    constructor: Person,
    name : "Nicholas",
    age : 29,
    job : "Software Engineer",
    friends : ["Shelby", "Court"],
    sayName : function () {
        alert(this.name);
    }
};

var person1 = new Person();
var person2 = new Person();

person1.friends.push("Van");

alert(person1.friends);        // "Shelby,Court,Van"
alert(person2.friends);        // "Shelby,Court,Van"
alert(person1.friends === person2.friends);        // true
```

Здесь для объекта `Person.prototype` определяется свойство `friends`, содержащее массив строк, после чего создаются два экземпляра `Person` и в массив `person1.friends` добавляется новая строка. Поскольку массив `friends` относится к свойству `Person.prototype`, а не к объекту `person1`, внесенные изменения отражаются в свойстве `person2.friends`, которое указывает на тот же массив. Если нужно, чтобы массив был общим для всех экземпляров, это нормально, однако обычно экземпляры должны иметь собственные копии всех свойств. По этой причине паттерн Прототип редко используется сам по себе.

Объединение паттернов Конструктор и Прототип

Для определения собственных типов чаще всего объединяют паттерны Конструктор и Прототип. С помощью первого из них определяют свойства экземпляра,

а с помощью второго — методы и общие свойства. При таком подходе все экземпляры получают свои копии свойств экземпляра, но совместно используют ссылки на методы, экономя память. Кроме того, это позволяет передавать аргументы в конструктор, объединяя, по сути, лучшие части обоих паттернов. Предыдущий пример можно переписать следующим образом:

Листинг HybridPatternExample01.htm

```
function Person(name, age, job){
    this.name = name;
    this.age = age;
    this.job = job;
    this.friends = ["Shelby", "Court"];
}

Person.prototype = {
    constructor: Person,
    sayName : function () {
        alert(this.name);
    }
};

var person1 = new Person("Nicholas", 29, "Software Engineer");
var person2 = new Person("Greg", 27, "Doctor");

person1.friends.push("Van");

alert(person1.friends);        //"Shelby,Court,Van"
alert(person2.friends);        //"Shelby,Court"
alert(person1.friends === person2.friends);        //false
alert(person1.sayName === person2.sayName);        //true
```

Обратите внимание на то, что теперь все свойства экземпляра определены в конструкторе, а общее свойство `constructor` и метод `sayName()` — в прототипе. Когда в массив `person1.friends` добавляется новая строка, на `person2.friend` это не влияет, потому что у каждого объекта свой массив.

Гибридный паттерн, объединяющий Конструктор и Прототип, в ECMAScript чаще всего используется для определения собственных ссылочных типов. Вообще говоря, при определении ссылочных типов он стал практически общепринятым.

Паттерн Динамический прототип

Разработчикам с опытом программирования на других объектно-ориентированных языках может показаться непривычным очевидное разделение конструктора и прототипа. *Паттерн Динамический прототип* (dynamic prototype pattern) решает эту проблему, не лишая нас преимуществ совместного использования конструктора и прототипа. Он инкапсулирует в конструкторе всю информацию, в том числе инициализацию прототипа, которая выполняется только при необходимости. Чтобы узнать, нуждается ли прототип в инициализации, можно проверить наличие доступного метода, например:

Листинг DynamicPrototypeExample01.htm


```
function Person(name, age, job){

    // свойства
    this.name = name;
    this.age = age;
    this.job = job;

    // методы
    if (typeof this.sayName != "function"){

        Person.prototype.sayName = function(){
            alert(this.name);
        };
    }
}

var friend = new Person("Nicholas", 29, "Software Engineer");
friend.sayName();
```

Выделенный код внутри конструктора добавляет к прототипу метод `sayName()`, если он не существует. Этот фрагмент выполняется только при первом вызове конструктора. Прототип оказывается инициализирован, и изменять его не требуется. Помните, что изменения прототипа немедленно отражаются на всех экземплярах, так что этот подход успешно решает задачу. В инструкции `if` достаточно проверить наличие любого свойства или метода, которые должны быть доступны после инициализации, — проверять все свойства и методы не требуется. Этот паттерн позволяет узнавать типы объектов с помощью оператора `instanceof`.

Если используется паттерн Динамический прототип, перезаписать прототип литералом объекта будет невозможно. Как уже отмечалось, перезапись прототипа при существующем экземпляре разрывает связь между ними.

Паттерн Паразитный конструктор

Паттерн Паразитный конструктор (parasitic constructor pattern) обычно используют как запасной вариант, если другие паттерны не срабатывают. По сути, он просто создает и возвращает другой объект, но при этом выглядит как обычный конструктор, например:

Листинг HybridFactoryPatternExample01.htm


```
function Person(name, age, job){
    var o = new Object();
    o.name = name;
    o.age = age;
    o.job = job;
    o.sayName = function(){
        alert(this.name);
    };
```

```
    return o;
}
var friend = new Person("Nicholas", 29, "Software Engineer");
friend.sayName();    // "Nicholas"
```

В этом примере конструктор `Person` создает новый объект, инициализирует его свойствами и методами, а затем возвращает. Этот код в точности повторяет паттерн Фабрика за исключением того, что функция вызывается как конструктор с помощью оператора `new`. Если конструктор не возвращает значение, по умолчанию он возвращает новый экземпляр объекта. Инструкция `return` в конце конструктора переопределяет значение, возвращаемое при его вызове.

Этот паттерн позволяет создавать конструкторы для объектов, которые невозможно создать иначе. Например, вам может потребоваться специальный массив с дополнительным методом. Поскольку непосредственного доступа к конструктору `Array` у вас нет, можно воспользоваться Паразитным конструктором:

Листинг HybridFactoryPatternExample02.htm


```
function SpecialArray(){

    // создание массива
    var values = new Array();

    // добавление значений
    values.push.apply(values, arguments);

    // назначение метода
    values.toPipedString = function(){
        return this.join("|");
};

    // возвращение массива
    return values;
}

var colors = new SpecialArray("red", "blue", "green");
alert(colors.toPipedString());    // "red|blue|green"
```

В этом примере определяется конструктор `SpecialArray`. Он создает массив и инициализирует его с помощью метода `push()`, которому передаются все аргументы конструктора. Затем к экземпляру добавляется метод `toPipedString()`, который просто выводит на экран значения массива с вертикальной чертой в качестве разделителя. Наконец, массив возвращается как значение функции. При вызове конструктора `SpecialArray` ему передаются первоначальные значения массива, а затем вызывается метод `toPipedString()`.

Говоря об этом паттерне важно помнить, что между возвращенным объектом и конструктором или прототипом конструктора нет никакой связи. Объект работает так, как если бы он был создан вне конструктора, поэтому вы не можете узнать тип

объекта с помощью оператора `instanceof`. По этой причине не следует использовать паттерн Паразитный конструктор, если работают другие паттерны.

Паттерн Защищенный конструктор

Согласно Дугласу Крокфорду (Douglas Crockford), *защищенные объекты* (durable objects) — это JavaScript-объекты, у которых нет открытых свойств, а методы не ссылаются на объект `this`. Защищенные объекты используются в основном в безопасных средах (в которых запрещены ключевые слова `this` и `new`), а также для защиты данных от остальной части приложения (как в гибридных веб-приложениях). *Паттерн Защищенный конструктор* (durable constructor pattern) в целом аналогичен паразитному конструктору, но имеет два отличия: методы созданного экземпляра не ссылаются на `this`, а конструктор никогда не вызывается с помощью оператора `new`. Конструктор `Person` из предыдущего раздела можно превратить в защищенный следующим образом:

```
function Person(name, age, job){

    // создание возвращаемого объекта
    var o = new Object();

    // необязательно: определите здесь закрытые переменные/функции

    // присоединение методов
    o.sayName = function(){
        alert(name);
    };

    // возвращение объекта
    return o;
}
```

Значение `name` в возвращенном объекте доступно только через метод `sayName()`. Защищенный конструктор `Person` используется следующим образом:

```
var friend = Person("Nicholas", 29, "Software Engineer");
friend.sayName();    // "Nicholas"
```

Переменная `friend` является защищенным объектом, то есть получить доступ к каким-либо его данным без вызова метода нельзя. Даже если где-то в другом коде в этот объект будут добавлены методы или данные, оригинальные данные, переданные в конструктор, останутся недоступными. Благодаря этому паттерн Защищенный конструктор может быть полезен при работе в безопасных средах выполнения, таких как ADsafe (www.adsafe.org) и Caja (http://code.google.com/p/google-caja/).

Как и паразитный конструктор, защищенный конструктор не связан с экземпляром объекта, поэтому оператор instanceof работать не будет.

Наследование

В контексте объектно-ориентированного программирования чаще всего вспоминают и обсуждают концепцию наследования. Многие объектно-ориентированные языки поддерживают наследование либо интерфейса, то есть только сигнатур методов, либо реализации, то есть фактических методов. Из-за того что в языке ECMAScript у функций нет сигнатур, наследование интерфейса в нем невозможно, а поддерживается только наследование реализации, которое выполняется преимущественно с помощью цепочек прототипов.

Цепочки прототипов

В ECMA-262 указано, что главный метод наследования в ECMAScript основан на *цепочках прототипов* (prototype chains). Его идея состоит в том, что с помощью прототипов один ссылочный тип получает свойства и методы другого. Если помните, у каждого конструктора есть объект прототипа, который указывает на этот конструктор, а у экземпляров есть внутренний указатель на прототип. А что, если прототипом окажется экземпляр другого типа? В этом случае у самого прототипа будет указатель на очередной прототип, у которого, в свою очередь, будет указатель на следующий конструктор. Если вторым прототипом также будет экземпляр другого типа, последовательность продолжится, формируя цепочку между экземплярами и прототипами. На этой идее и основано наследование с помощью цепочек прототипов.

Реализация цепочки прототипов включает следующий шаблон кода:

Листинг PrototypeChainingExample01.htm


```
function SuperType(){
    this.property = true;
}

SuperType.prototype.getSuperValue = function(){
    return this.property;
};

function SubType(){
    this.subproperty = false;
}

// наследование от SuperType
SubType.prototype = new SuperType();

SubType.prototype.getSubValue = function (){
    return this.subproperty;
};

var instance = new SubType();
alert(instance.getSuperValue());    // true
```

В этом коде определяются типы SuperType и SubType, оба с одним свойством и одним методом. Главное различие между двумя типами в том, что SubType наследуется от SuperType, для чего новый экземпляр SuperType назначается свойству SubType.prototype. В результате оригинальный прототип перезаписывается новым объектом, то есть все свойства и методы экземпляра SuperType появляются у SubType.prototype. После наследования к свойству SubType.prototype добавляется новый метод в дополнение к тем, что были унаследованы от SuperType. Отношения между экземпляром, конструкторами и прототипами показаны на рис. 6 4.

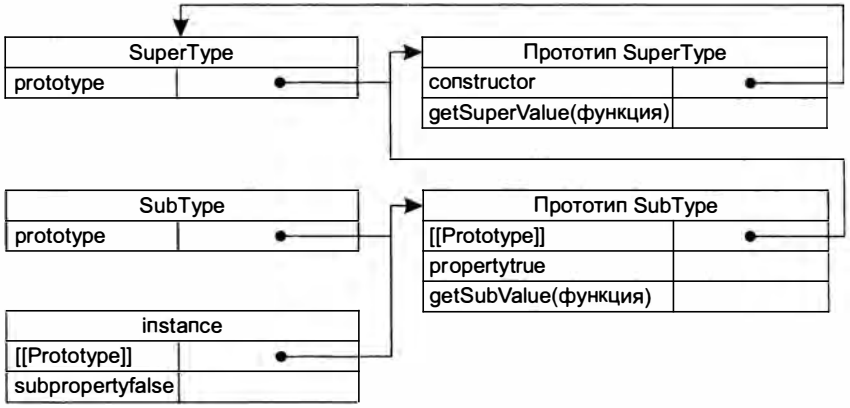

Рис. 6.4

Таким образом, вместо прототипа, предлагаемого по умолчанию, типу SubType назначается новый прототип, который является экземпляром SuperType. Он не только получает свойства и методы экземпляра SuperType, но и указывает на прототип SuperType. В итоге объект instance указывает на свойство SubType.prototype, указывающее на SuperType.prototype. Заметьте, что метод getSuperValue() остается у объекта SuperType.prototype, а свойство property переходит к SubType.prototype. Это объясняется тем, что getSuperValue() — метод прототипа, а property — свойство экземпляра. SubType.prototype теперь является экземпляром SuperType, поэтому и свойство property хранится в нем. Отметим также, что instance.constructor указывает на SuperType, потому что свойство constructor у объекта SubType.prototype было перезаписано.

Цепочки прототипов учитываются механизмом поиска в прототипах. Как вы, наверное, помните, при чтении свойства сначала выполняется его поиск в экземпляре, а если свойство там не обнаруживается, инициируется поиск в прототипе. Если наследование реализовано с помощью цепочки прототипов, поиск может быть продолжен в ней. Так, в предыдущем примере при вызове instance.getSuperValue() выполняется поиск метода в экземпляре, затем — в свойстве SubType.prototype и, наконец, — в свойстве SuperType.prototype, где он и определен. Поиск свойств и методов всегда продолжается до конца цепочки прототипов.

Прототипы, предлагаемые по умолчанию

На самом деле в цепочке прототипов есть еще одно звено. Все ссылочные типы наследуются по умолчанию через цепочку прототипов от типа `Object`, а это означает, что в любой функции внутренний указатель указывает на прототип `Object.prototype`. Именно так пользовательские типы наследуют все предлагаемые по умолчанию методы типа `Object`, например `toString()` и `valueOf()`. Полностью цепочка прототипов с дополнительным уровнем наследования для предыдущего примера показана на рис. 6.5.

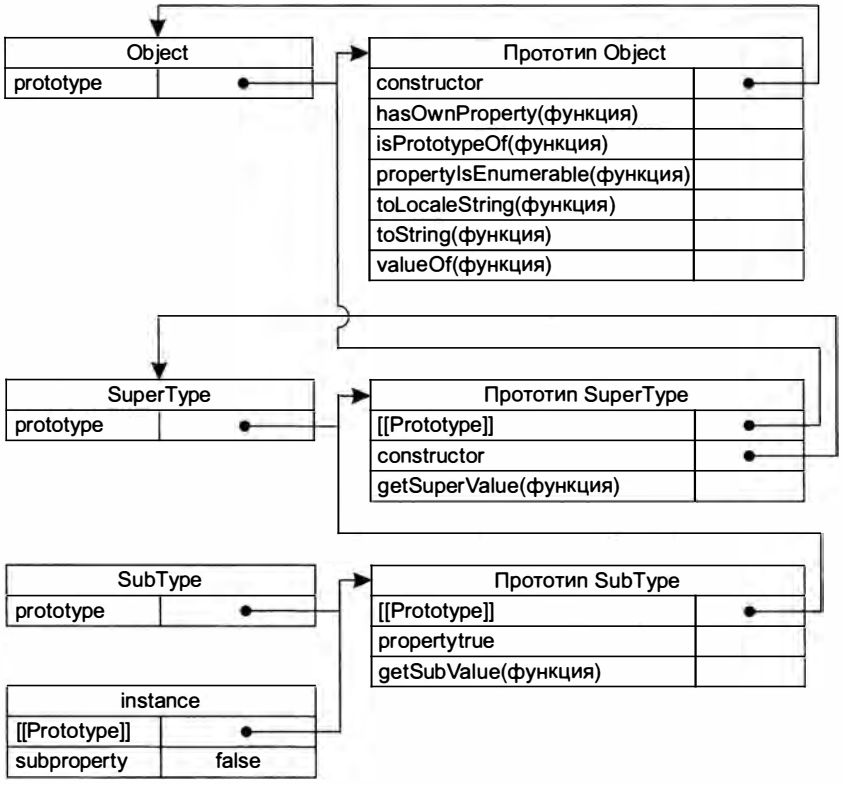

Рис. 6.5

Тип `SubType` наследуется от `SuperType`, а `SuperType` — от `Object`. Если добавить в код вызов `instance.toString()`, будет вызван метод, принадлежащий свойству `Object.prototype`.

Связи между прототипами и экземплярами

Есть два способа выяснить, связаны ли экземпляр и прототип. Первый — использовать оператор `instanceof`, который возвращает `true`, если указанный конструктор имеется в цепочке прототипов экземпляра, например:

Листинг PrototypeChainingExample01.htm


```
alert(instance instanceof Object);       // true
alert(instance instanceof SuperType);    // true
alert(instance instanceof SubType);      // true
```

Объект `instance` благодаря цепочке прототипов является экземпляром типов `Object`, `SuperType` и `SubType`, поэтому оператор `instanceof` возвращает `true` для всех трех конструкторов.

Второй способ основан на имеющемся у каждого прототипа в цепочке методе `isPrototypeOf()`, который возвращает `true`, если переданный ему экземпляр входит в цепочку, например:

Листинг PrototypeChainingExample01.htm

```
alert(Object.prototype.isPrototypeOf(instance));       // true
alert(SuperType.prototype.isPrototypeOf(instance));    // true
alert(SubType.prototype.isPrototypeOf(instance));      // true
```

Работа с методами

В подтипе часто требуется переопределить метод супертипа или реализовать новые методы, отсутствующие в супертипе. Для этого необходимо добавить методы в прототип после его назначения подтипу. Рассмотрим пример:

Листинг PrototypeChainingExample02.htm


```
function SuperType(){
    this.property = true;
}

SuperType.prototype.getSuperValue = function(){
    return this.property;
};

function SubType(){
    this.subproperty = false;
}

// наследование от SuperType
SubType.prototype = new SuperType();

// новый метод
SubType.prototype.getSubValue = function (){
    return this.subproperty;
};

// переопределение существующего метода
SubType.prototype.getSuperValue = function (){
    return false;
};

var instance = new SubType();
alert(instance.getSuperValue());        // false
```

Выделенный фрагмент содержит определения двух методов. Первый, `getSubValue()`, представляет собой новый метод типа `SubType`, а второй, `getSuperValue()`, затеняет одноименный метод, который уже имеется в цепочке прототипов. При вызове `getSuperValue()` для экземпляра `SubType` вызывается новая версия, но для экземпляров `SuperType` по-прежнему вызывается оригинал. Следует отметить, что оба метода определяются после назначения экземпляра `SuperType` прототипом `SubType`.

При использовании цепочки прототипов нельзя создавать методы прототипа с помощью литерала объекта, потому что это перезаписывает цепочку, например:

Листинг PrototypeChainingExample03.htm


```
function SuperType(){
    this.property = true;
}

SuperType.prototype.getSuperValue = function(){
    return this.property;
};

function SubType(){
    this.subproperty = false;
}

// наследование от SuperType
SubType.prototype = new SuperType();

// попытка добавить новые методы аннулирует предыдущую строку
SubType.prototype = {
    getSubValue : function (){
        return this.subproperty;
    },

    someOtherMethod : function (){
        return false;
    }
};

var instance = new SubType();
alert(instance.getSuperValue());    //ошибка!
```

В этом коде экземпляр `SuperType`, сделанный первоначально прототипом `SubType`, заменяется литералом объекта. В результате прототипом становится новый экземпляр `Object`, что разрывает цепочку прототипов и, соответственно, связь между типами `SubType` и `SuperType`.

Проблемы с цепочками прототипов

Хотя цепочка прототипов — эффективное средство наследования, оно не лишено недостатков. Главная проблема связана с прототипами, содержащими ссылочные значения. Напомню, что свойства прототипов с такими значениями являются

общими для всех экземпляров, поэтому свойства обычно определяют в конструкторе, а не в прототипе. Когда наследование реализуется на основе прототипов, в качестве прототипа на самом деле используется экземпляр другого типа, при этом свойства экземпляра становятся свойствами прототипа. Эту проблему поясняет следующий пример:

Листинг PrototypeChainingExample04.htm


```
function SuperType(){
    this.colors = ["red", "blue", "green"];
}

function SubType(){
}

// наследование от SuperType
SubType.prototype = new SuperType();

var instance1 = new SubType();
instance1.colors.push("black");
alert(instance1.colors);        // "red,blue,green,black"

var instance2 = new SubType();
alert(instance2.colors);        // "red,blue,green,black"
```

Здесь в конструкторе `SuperType` определяется свойство `colors`, содержащее массив (ссылочное значение). Каждый экземпляр `SuperType` имеет собственное свойство `colors` с отдельным массивом. При наследовании типа `SubType` от `SuperType` с помощью цепочки прототипов объект `SubType.prototype` становится экземпляром `SuperType` и получает собственное свойство `colors`, что аналогично явному созданию свойства `SubType.prototype.colors`. В результате все экземпляры `SubType` совместно используют одно свойство `colors`. Это подтверждается тем, что изменения свойства `instance1.colors` отражаются на `instance2.colors`.

Вторая проблема с цепочкой прототипов заключается в том, что при создании экземпляра подтипа нельзя передать аргументы в конструктор супертипа. Фактически, нет способа передать аргументы в конструктор супертипа, не затронув все его экземпляры. Из-за этих двух проблем цепочки прототипов редко используются сами по себе.

Кража конструктора

Пытаясь решить проблему наследования ссылочных значений прототипов, разработчики начали использовать паттерн *Кража конструктора* (constructor stealing), который часто называют маскировкой объекта или классическим наследованием. Суть этого приема проста и сводится к вызову конструктора супертипа в конструкторе подтипа. Так как функции — это просто объекты, выполняющие код в конкретном контексте, мы можем с помощью методов `apply()` и `call()` вызывать конструкторы для только что созданных объектов, например:

Листинг ConstructorStealingExample01.htm

```
function SuperType(){
    this.colors = ["red", "blue", "green"];
}

function SubType(){
    // наследование от SuperType
    SuperType.call(this);
}
var instance1 = new SubType();
instance1.colors.push("black");

alert(instance1.colors);        // "red,blue,green,black"

var instance2 = new SubType();
alert(instance2.colors);        // "red,blue,green"
```

Кража конструктора выполняется в выделенной строке. Для этого метод `call()` (который можно заменить методом `apply()`) вызывает конструктор `SuperType` в контексте созданного экземпляра `SubType`, выполняя для него весь содержащийся в конструкторе код инициализации. Благодаря этому каждый экземпляр получает собственную копию свойства `colors`.

Передача аргументов

Одним из преимуществ кражи конструктора над цепочками прототипов является возможность передать в конструкторе подтипа аргументы в конструктор супертипа. Рассмотрим пример:

Листинг ConstructorStealingExample02.htm


```
function SuperType(name){
    this.name = name;
}

function SubType(){
    // наследование от SuperType с передачей аргумента в супертип
    SuperType.call(this, "Nicholas");

    // свойство экземпляра
    this.age = 29;
}

var instance = new SubType();
alert(instance.name);    // "Nicholas";
alert(instance.age);     // 29
```

В этом коде конструктор `SuperType` принимает единственный аргумент `name`, который просто назначается свойству. В конструкторе типа `SubType` можно передать значение конструктору `SuperType`, задав значение свойства `name` для экземпляра `SubType`. Чтобы гарантировать, что другие свойства подтипа не будут перезаписаны в конструкторе `SuperType`, их можно определить после его вызова.

Проблемы с кражами конструктора

Кража конструктора имеет тот же недостаток, что и паттерн Конструктор с пользовательскими типами: необходимость определять методы в конструкторе не позволяет задействовать их многократно. Кроме того, методы, определенные в прототипе супертипа, недоступны в подтипе, вследствие чего все типы могут использовать только паттерн Конструктор. Из-за этих проблем кража конструктора также редко используется сама по себе.

Комбинированное наследование

Комбинированное наследование (combination inheritance), которое иногда называют псевдоклассическим, объединяет преимущества цепочки прототипов и кражи конструктора. Его идея состоит в том, что для наследования свойств и методов прототипа используется цепочка прототипов, а для наследования свойств экземпляра — кража конструктора. Определение методов в прототипе обеспечивает многократное использование функций, но при этом каждый экземпляр может иметь собственные свойства. Рассмотрим следующий пример:

Листинг CombinationInheritanceExample01.htm


```
function SuperType(name){
    this.name = name;
    this.colors = ["red", "blue", "green"];
}

SuperType.prototype.sayName = function(){
    alert(this.name);
};

function SubType(name, age){

    // наследование свойств
    SuperType.call(this, name);

    this.age = age;
}

// наследование методов
SubType.prototype = new SuperType();

SubType.prototype.sayAge = function(){
    alert(this.age);
};

var instance1 = new SubType("Nicholas", 29);
instance1.colors.push("black");
alert(instance1.colors);    // "red,blue,green,black"
instance1.sayName();        // "Nicholas";
instance1.sayAge();         // 29

var instance2 = new SubType("Greg", 27);
alert(instance2.colors);    // "red,blue,green"
instance2.sayName();        // "Greg";
instance2.sayAge();         // 27
```

Здесь в конструкторе `SuperType` определяются свойства `name` и `colors`, а к прототипу `SuperType` добавляется единственный метод `sayName()`. В конструкторе `SubType` мы вызываем конструктор `SuperType`, передавая в него аргумент `name`, и определяем свойство `age`, после чего назначаем экземпляр `SuperType` прототипом `SubType` и определяем в подтипе новый метод `sayAge()`. Завершается этот код созданием двух отдельных экземпляров `SubType` с собственными копиями свойств (включая `colors`), но общими методами.

Комбинированное наследование преодолело недостатки цепочки прототипов и кражи конструктора и стало самым популярным паттерном наследования в JavaScript. Оно позволяет использовать оператор `instanceof` и метод `isPrototypeOf()` для идентификации связей между объектами.

Прототипное наследование

В 2006 году Дуглас Крокфорд (Douglas Crockford) в статье «Prototypal Inheritance in JavaScript» (Прототипное наследование в JavaScript) представил способ наследования без строго определенных конструкторов. Идея заключалась в том, что с помощью прототипов можно создавать новые объекты на основе существующих без определения пользовательских типов. Функция, которую он привел в качестве примера, выглядела так:

```
function object(o){
    function F(){}
    F.prototype = o;
    return new F();
}
```

Функция `object()` создает временный конструктор, назначает полученный объект прототипом конструктора и возвращает новый экземпляр временного типа. По сути, она выполняет поверхностное копирование любого переданного в нее объекта. Рассмотрим такой пример:

Листинг PrototypalInheritanceExample01.htm


```
var person = {
    name: "Nicholas",
    friends: ["Shelby", "Court", "Van"]
};

var anotherPerson = object(person);
anotherPerson.name = "Greg";
anotherPerson.friends.push("Rob");

var yetAnotherPerson = object(person);
yetAnotherPerson.name = "Linda";
yetAnotherPerson.friends.push("Barbie");

alert(person.friends);    // "Shelby,Court,Van,Rob,Barbie"
```

Согласно Крокфорду, если некоторый объект нужно использовать как основу для создания другого объекта, следует передать его в функцию `object()` и внести нужные изменения в возвращенный объект. В приведенном примере объект `person` содержит информацию, которая должна быть доступна в других объектах, поэтому мы передаем его в функцию `object()`, которая возвращает новый объект. Прототипом этого объекта является экземпляр `person` с примитивным и ссылочным свойствами. После двух вызовов функции `object()` ссылочное свойство `person.friends` является общим для трех объектов: `person`, `anotherPerson` и `yetAnotherPerson`. По сути, этот код создает два клона `person`.

В ECMAScript 5 концепция прототипного наследования была формализована в методе `Object.create()`. Он принимает два аргумента: прототип нового объекта и необязательный объект, определяющий для нового объекта дополнительные свойства. Если второй аргумент опущен, метод `Object.create()` эквивалентен методу `object()`:

Листинг PrototypalInheritanceExample02.htm


```
var person = {
    name: "Nicholas",
    friends: ["Shelby", "Court", "Van"]
};

var anotherPerson = Object.create(person);
anotherPerson.name = "Greg";
anotherPerson.friends.push("Rob");

var yetAnotherPerson = Object.create(person);
yetAnotherPerson.name = "Linda";
yetAnotherPerson.friends.push("Barbie");

alert(person.friends);    // "Shelby,Court,Van,Rob,Barbie"
```

Второй аргумент метода `Object.create()` имеет такой же формат, что и в методе `Object.defineProperties()`: каждое дополнительное свойство указывается вместе с его дескриптором. Любые свойства, добавленные таким образом, затеняют одноименные свойства прототипа:

Листинг PrototypalInheritanceExample03.htm

```
var person = {
    name: "Nicholas",
    friends: ["Shelby", "Court", "Van"]
};

var anotherPerson = Object.create(person, {
    name: {
        value: "Greg"
    }
});

alert(anotherPerson.name);    // "Greg"
```

Метод `Object.create()` поддерживается в Internet Explorer 9+, Firefox 4+, Safari 5+, Opera 12+ и Chrome.

Прототипное наследование полезно, если накладные расходы на создание отдельных конструкторов нежелательны, но при этом требуется, чтобы новый объект был похож на уже существующий. Помните, что свойства со ссылочными значениями в этом случае будут общими, как и при использовании паттерна Прототип.

Паразитное наследование

С прототипным наследованием тесно связана концепция *паразитного наследования* (parasitic inheritance), также популяризированная Крокфордом. Паттерн Паразитное наследование похож как на паттерн Паразитный конструктор, так и на паттерн Фабрика. Он включает создание объекта в функции, расширение его возможностей и возвращение расширенного объекта из функции. Базовый паттерн Паразитное наследование выглядит следующим образом:

```
function createAnother(original){
    var clone = object(original); // создание объекта путем вызова функции
    clone.sayHi = function(){     // расширение возможностей объекта
        alert("hi");
    };
    return clone;                 // возвращение объекта
}
```

Функция `createAnother()` принимает в качестве единственного аргумента объект, на основе которого нужно создать производный объект. Полученный аргумент передается в функцию `object()`, а возвращенный из нее результат назначается переменной `clone`. К объекту `clone` добавляется метод `sayHi()`, а затем расширенный объект возвращается. Функцию `createAnother()` можно использовать следующим образом:

```
var person = {
    name: "Nicholas",
    friends: ["Shelby", "Court", "Van"]
};

var anotherPerson = createAnother(person);
anotherPerson.sayHi();    // "hi"
```

В этом фрагменте на основе объекта `person` создается объект `anotherPerson`, который содержит все свойства и методы `person`, а также дополнительный метод `sayHi()`.

Паразитное наследование — это еще один паттерн, который полезен, если вы имеете дело в основном с объектами, а не собственными типами и конструкторами. Метод `object()` для паразитного наследования не требуется — подойдет любая функция, возвращающая новый объект.

Как и в паттерне Конструктор, функции, добавленные к объектам при паразитном наследовании, не используются повторно, что делает код менее эффективным.

Паразитное комбинированное наследование

Комбинированное наследование — наиболее популярный паттерн наследования в JavaScript, но и у него есть слабые стороны. Самое неэффективное в этом паттерне то, что конструктор супертипа всегда вызывается дважды: в первый раз для создания прототипа подтипа, а во второй — внутри конструктора подтипа. По сути, прототип подтипа получает все свойства экземпляра супертипа только для того, чтобы перезаписать их в конструкторе подтипа. Давайте еще раз взглянем на пример с комбинированным наследованием:

```
function SuperType(name){
    this.name = name;
    this.colors = ["red", "blue", "green"];
}
SuperType.prototype.sayName = function(){
    alert(this.name);
};
function SubType(name, age){
    SuperType.call(this, name);              // второй вызов SuperType()
    this.age = age;
}
SubType.prototype = new SuperType();         // первый вызов SuperType()
SubType.prototype.constructor = SubType;
SubType.prototype.sayAge = function(){
    alert(this.age);
};
```

Выделенные строки кода указывают, когда вызывается конструктор `SuperType`. При выполнении этого кода к объекту `SubType.prototype` добавляются свойства `name` и `colors`, которые первоначально являются свойствами экземпляра типа `SuperType`. При последующем вызове конструктора `SubType` в нем вызывается конструктор `SuperType`, который создает свойства экземпляра `name` и `colors` у нового объекта, маскируя свойства прототипа. Этот процесс показан на рис. 6.6.

Как видите, из-за двукратного вызова конструктора `SuperType` свойства `name` и `colors` есть как у экземпляра, так и у прототипа `SubType`. К счастью, это можно исправить.

При *паразитном комбинированном наследовании* (parasitic combination inheritance) мы используем кражу конструктора для наследования свойств и гибридную форму цепочки прототипов для наследования методов. Идея в том, что вместо назначения прототипа подтипу путем вызова конструктора супертипа мы просто используем копию прототипа супертипа. Иначе говоря, выполняется паразитное наследование прототипа супертипа, после чего результат назначается прототипу подтипа. Базовый паттерн таков:

```
function inheritPrototype(subType, superType){
    var prototype = object(superType.prototype); // создание объекта
    prototype.constructor = subType;             // расширение объекта
    subType.prototype = prototype;               // назначение объекта
}
```

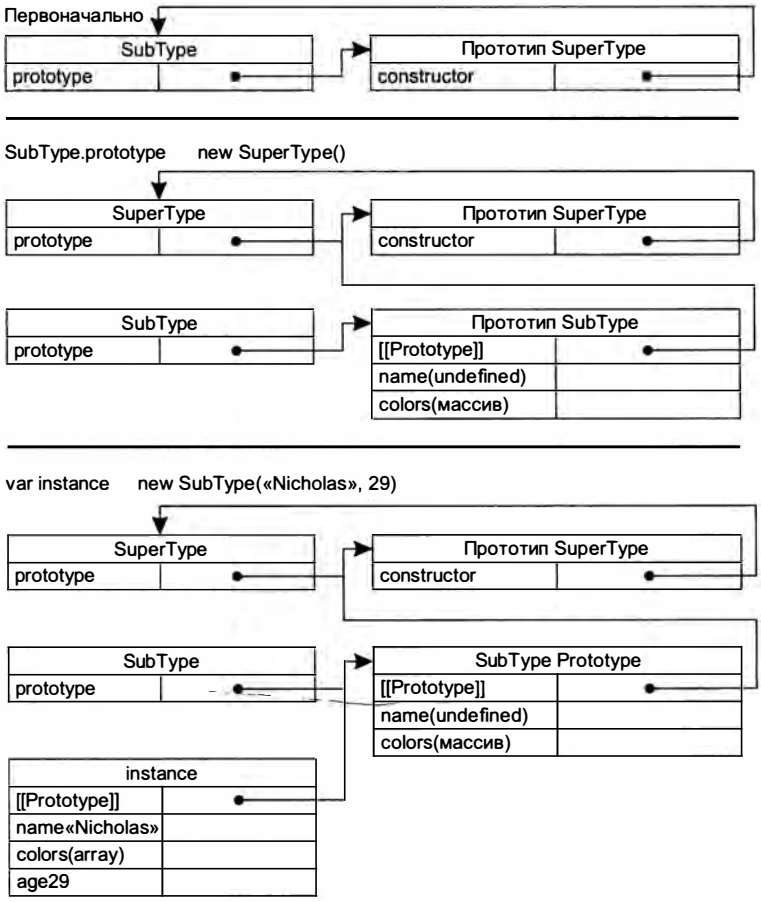

Рис. 6.6

Функция `inheritPrototype()` реализует очень простое паразитное комбинированное наследование. Она принимает два аргумента: конструктор подтипа и конструктор супертипа. Внутри функции первым делом создается клон прототипа супертипа. Затем к прототипу добавляется свойство `constructor`, чтобы компенсировать потерю первоначального свойства конструктора при перезаписи прототипа. Наконец, созданный объект назначается прототипом подтипа. Вызовом функции `inheritPrototype()` можно заменить назначение прототипа подтипа в предыдущем примере:

Листинг ParasiticCombinationInheritanceExample01.htm


```
function SuperType(name){
    this.name = name;
    this.colors = ["red", "blue", "green"];
}
```

```
SuperType.prototype.sayName = function(){
    alert(this.name);
};

function SubType(name, age){
    SuperType.call(this, name);

    this.age = age;
}

inheritPrototype(SubType, SuperType);

SubType.prototype.sayAge = function(){
    alert(this.age);
};
```

Этот код более эффективен, потому что конструктор `SuperType` вызывается только один раз и лишние свойства в прототипе `SubType` не создаются. Кроме того, цепочка прототипов остается нетронутой, что сохраняет работоспособность оператора `instanceof` и метода `isPrototypeOf()`. Паразитное комбинированное наследование считается оптимальным способом наследования ссылочных типов.

Библиотека YUI (Yahoo! User Interface — пользовательский интерфейс Yahoo!) стала первой популярной JavaScript-библиотекой, в которой в методе Y.extend() было реализовано паразитное комбинированное наследование. Подробные сведения о YUI доступны на сайте http://yuilibrary.com/.

Резюме

ECMAScript поддерживает объектно-ориентированное программирование без классов и интерфейсов. Объекты можно создавать и расширять в любой момент выполнения кода, что делает их динамичными, а не строго определенными сущностями. Вместо классов для создания объектов используются паттерны.

- Паттерн Фабрика представляет собой простую функцию, которая создает объект, назначает ему свойства и методы, а затем возвращает его. Этот паттерн вышел из употребления, когда появился паттерн Конструктор.
- Паттерн Конструктор позволяет разработчикам определять собственные ссылочные типы, экземпляры которых можно создавать с помощью оператора `new`, как и экземпляры встроенных типов. К сожалению, никакие члены конструктора, включая функции, не используются повторно, хотя ничто не мешает слабо типизированным ECMAScript-функциям быть общими для нескольких экземпляров объектов.
- В паттерне Прототип эта проблема решается путем назначения общих свойств и методов свойству `prototype` конструктора. В комбинированном паттерне Конструктор+Прототип свойства экземпляра определяются в конструкторе, а общие свойства и методы — в прототипе.

Наследование в JavaScript реализуется преимущественно с помощью цепочек прототипов, для формирования которых свойству `prototype` конструктора подтипа назначается экземпляр другого типа. Как и при наследовании с классами, подтип при этом получает все свойства и методы супертипа. Проблема цепочек прототипов состоит в том, что все унаследованные свойства и методы становятся общими для экземпляров объектов, из-за чего этот паттерн редко используется сам по себе. Паттерн Кража конструктора устраняет эту проблему, вызывая конструктор супертипа в конструкторе подтипа. Это позволяет каждому экземпляру иметь собственные свойства, но вынуждает определять типы только с помощью паттерна Конструктор. Чаще всего применяется комбинированное наследование, в котором общие свойства и методы наследуются с помощью цепочки прототипов, а свойства экземпляра — путем кражи конструктора.

Существуют также альтернативные паттерны наследования.

- Прототипное наследование — это наследование без предопределенных конструкторов, при котором выполняется поверхностное копирование базового объекта. Результат копирования можно расширять.
- С прототипным тесно связано паразитное наследование, которое включает создание объекта на основе другого объекта или некоторой информации, расширение созданного объекта и его возвращение. Этот паттерн можно также использовать с комбинированным наследованием, чтобы не вызывать лишний раз конструктор супертипа.
- Паразитное комбинированное наследование считается наиболее эффективным способом реализации наследования на основе типов.

7 Функции-выражения

- Характеристики функций-выражений
- Рекурсия
- Закрытые переменные и замыкания

Функции-выражения — одна из наиболее мощных возможностей JavaScript, которую многие не понимают. Как отмечено в главе 5, определить функцию можно с помощью объявления или выражения. Объявление функции имеет следующий формат:

```
function functionName(arg0, arg1, arg2) {
    // тело функции
}
```

Идентификатор после ключевого слова `function` определяет имя функции. В Firefox, Safari, Chrome и Opera у функций с именем есть нестандартное свойство `name`, значением которого всегда является этот идентификатор, например:

Листинг FunctionNameExample01.htm


```
// работает только в Firefox, Safari, Chrome и Opera
alert(functionName.name);    // "functionName"
```

Объявления функций примечательны тем, что они *поднимаются* (hoisting), то есть считываются до выполнения кода. Это означает, что вызовы функции могут предшествовать ее объявлению:

Листинг FunctionDeclarationHoisting01.htm


```
sayHi();
function sayHi(){
    alert("Hi!");
}
```

Ошибка в этом примере не возникает как раз благодаря тому, что объявление функции считывается до выполнения кода.

Функцию также можно создать с помощью функции-выражения. Обычно она имеет следующий формат:

```
var functionName = function(arg0, arg1, arg2){
    // тело функции
};
```

Этот паттерн выглядит как обычная инициализация переменной `functionName`, только здесь ей присваивается функция. Созданная функция считается *анонимной* (anonymous function), потому что после ключевого слова `function` нет идентификатора; анонимные функции также иногда называют *лямбда-функциями* (lambda functions). Свойство `name` у таких функций содержит пустую строку.

Функции-выражения можно использовать только после их создания. Так, при выполнении следующего кода возникает ошибка:

```
sayHi();    // ошибка - функция еще не существует
var sayHi = function(){
    alert("Hi!");
};
```

Подъем функций — ключевое различие между объявлениями и функциями-выражениями. Например, результат выполнения следующего кода может вас удивить:

Листинг FunctionDeclarationsErrorExample01.htm

```
// Никогда не делайте этого!
if(condition){
    function sayHi(){
        alert("Hi!");
    }
} else {
    function sayHi(){
        alert("Yo!");
    }
}
```

На первый взгляд, этот код выбирает то или иное объявление функции `sayHi()` в зависимости от условия, но в действительности ECMAScript не поддерживает такой синтаксис и интерпретаторы JavaScript пытаются исправить ошибку. Проблема в том, что они делают это по-разному. Большинство браузеров возвращает второе объявление функции независимо от условия, а Firefox возвращает первое, если условие истинно. Такой код опасен и использовать его не следует, однако функции-выражения в нем вполне допустимы:

```
// все в порядке
var sayHi;

if(condition){
```

```
    sayHi = function(){
        alert("Hi!");
    };
} else {
    sayHi = function(){
        alert("Yo!");
    };
}
```

Этот код работает без сюрпризов, назначая одно из функций-выражений переменной `sayHi` в зависимости от условия.

Возможность назначать функции переменным позволяет также возвращать функции из других функций. Давайте вспомним функцию `createComparisonFunction()` из главы 5:

```
function createComparisonFunction(propertyName) {

    return function(object1, object2){
        var value1 = object1[propertyName];
        var value2 = object2[propertyName];

        if (value1 < value2){
            return -1;
        } else if (value1 > value2){
            return 1;
        } else {
            return 0;
        }
    };
}
```

Функция `createComparisonFunction()` возвращает анонимную функцию. Возможно, она будет назначена переменной или вызвана иным образом, но внутри `createComparisonFunction()` она анонимна. Каждый раз, когда функции используются как значения, мы имеем дело с функциями-выражениями. Позже в этой главе мы рассмотрим другие способы их применения.

Рекурсия

Рекурсивной функцией (recursive function) обычно называют функцию, которая вызывает сама себя, например:

Листинг RecursionExample01.htm


```
function factorial(num){
    if (num <= 1){
        return 1;
    } else {
        return num * factorial(num-1);
    }
}
```

Это классический пример вычисления факториала. Хотя с функцией все в порядке, ее работу легко можно нарушить:

Листинг RecursionExample01.htm

```
var anotherFactorial = factorial;
factorial = null;
alert(anotherFactorial(4));    // ошибка!
```

Здесь функция `factorial()` назначается переменной `anotherFactorial`, а затем переменной `factorial` присваивается значение `null`, в результате чего остается только одна ссылка на оригинальную функцию. При вызове `anotherFactorial()` возникает ошибка, потому что функция `factorial()` больше не доступна. Эту проблему можно устранить с помощью свойства `arguments.callee`.

Если помните, `arguments.callee` — это указатель на выполняемую функцию, с помощью которого можно вызвать ее рекурсивно:

Листинг RecursionExample02.htm

```
function factorial(num){
    if (num <= 1){
        return 1;
    } else {
        return num * arguments.callee(num-1);
    }
}
```

Использование свойства `arguments.callee` в выделенной строке вместо имени функции гарантирует, что функция будет работать независимо от способа доступа к ней. В рекурсивных функциях рекомендуется всегда задействовать свойство `arguments.callee`, а не имя функции.

В строгом режиме значение `arguments.callee` недоступно, а попытка прочитать его приводит к ошибке. К счастью, тот же результат можно получить с помощью *именованных функций-выражений* (named function expressions), например:

```
var factorial = (function f(num){
    if (num <= 1){
        return 1;
    } else {
        return num * f(num-1);
    }
});
```

В этом коде именованная функция-выражение `f()` назначается переменной `factorial`. Имя `f` остается тем же, даже если функция назначается другой переменной, благодаря чему рекурсивный вызов всегда выполняется правильно. Этот подход работает и в нестрогом, и в строгом режиме.

Замыкания

Термины «анонимная функция» и «замыкание» часто употребляются как синонимы, но это ошибка. *Замыкание* (closure) — это функция, которой доступны переменные из области видимости другой функции. Обычно для создания замыкания одну функцию определяют внутри другой, как функцию `createComparisonFunction()` в предыдущем примере:

```
function createComparisonFunction(propertyName) {

    return function(object1, object2){
        var value1 = object1[propertyName];
        var value2 = object2[propertyName];

        if (value1 < value2){
            return -1;
        } else if (value1 > value2){
            return 1;
        } else {
            return 0;
        }
    };
}
```

В выделенных строках внутренней (анонимной) функции осуществляется доступ к переменной `propertyName`, полученной внешней функцией. Интересно то, что эта переменная остается доступной для анонимной функции, даже когда она вызывается в другом коде. Это происходит потому, что цепочка областей видимости внутренней функции включает область видимости функции `createComparisonFunction()`. Чтобы понять, как это возможно, давайте обсудим, что происходит при первом вызове функции.

Чтобы хорошо разбираться в замыканиях, важно в деталях рассмотреть создание и использование цепочек областей видимости, с которыми мы познакомились в главе 4. При вызове функции для нее создаются контекст выполнения и цепочка областей видимости. Объект активации функции инициализируется значениями из объекта `arguments` и имеющимися именованными аргументами. Объект активации внешней функции является вторым в цепочке областей видимости, затем ее продолжают другие оставшиеся функции-контейнеры и завершает глобальный контекст выполнения.

При чтении и записи значений в функции выполняется поиск соответствующих переменных в цепочке областей видимости. Рассмотрим пример:

```
function compare(value1, value2){
    if (value1 < value2){
        return -1;
    } else if (value1 > value2){
        return 1;
```

```
    } else {
        return 0;
    }
}

var result = compare(5, 10);
```

В этом коде определена функция `compare()`, которая затем вызывается в глобальном контексте выполнения. При первом вызове функции создается объект активации, содержащий переменные `arguments`, `value1` и `value2`. Следующим в цепочке областей видимости контекста выполнения `compare()` является объект переменных глобального контекста выполнения, который содержит переменные `this`, `result` и `compare`. Эти связи иллюстрирует рис. 7.1.

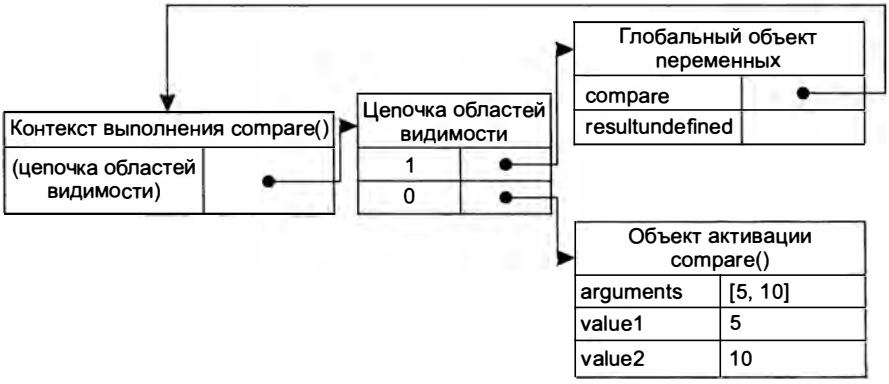

Рис. 7.1

За кулисами переменные в каждом контексте выполнения представляются объектом. Объект переменных глобального контекста существует всегда, тогда как объекты переменных локальных контекстов, например функции `compare()`, создаются только на время выполнения функции. При определении функции `compare()` создается ее цепочка областей видимости, которая инициализируется глобальным объектом переменных и сохраняется во внутреннем свойстве `[[Scope]]`. При вызове функции создается контекст выполнения и составляется его цепочка областей видимости, для чего в нее копируются объекты из свойства `[[Scope]]` функции. После этого система создает объект активации (используемый также как объект переменных) и помещает его в начало цепочки областей видимости контекста. В нашем примере это означает, что цепочка областей видимости контекста выполнения `compare()` содержит два объекта переменных: локальный объект активации и глобальный объект переменных. Имейте в виду, что цепочка областей видимости является, по сути, списком указателей на объекты переменных и не содержит сами объекты.

Когда переменная используется внутри функции, выполняется поиск имени переменной в цепочке областей видимости. По завершении функции локальный объект активации уничтожается, после чего в памяти остается только глобальная область видимости. Однако замыкания работают иначе.

Если одна функция определена внутри другой, в цепочку областей видимости внутренней функции добавляется объект активации внешней. Так, в примере с функцией `createComparisonFunction()` ссылка на нее содержится в цепочке областей видимости анонимной функции. На рис. 7.2 показаны эти связи при выполнении следующего кода:

```
var compare = createComparisonFunction("name");
var result = compare({ name: "Nicholas" }, { name: "Greg" });
```

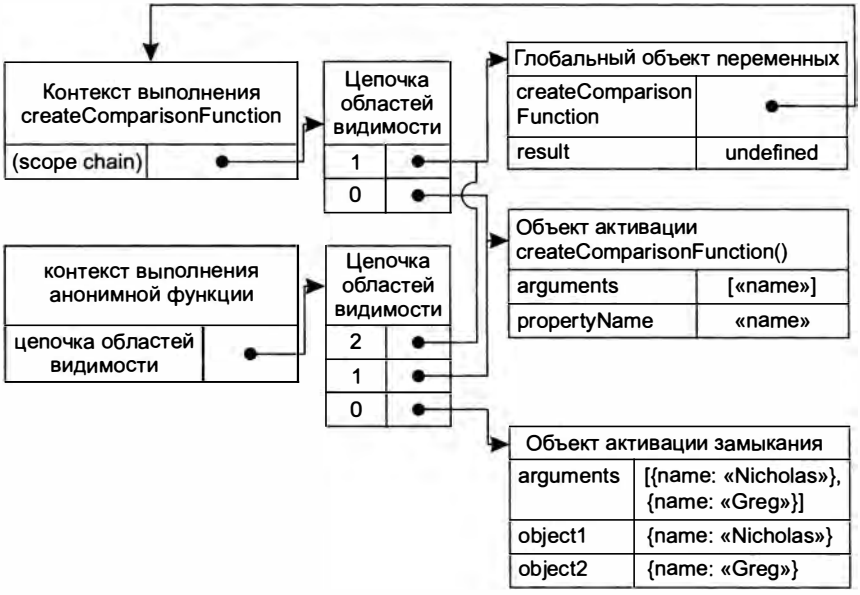

Рис. 7.2

Когда анонимная функция возвращается из `createComparisonFunction()`, ее цепочка областей видимости содержит объект активации `createComparisonFunction()` и глобальный объект переменных, что предоставляет анонимной функции доступ ко всем переменным функции `createComparisonFunction()`. Другой интересный побочный эффект заключается в том, что объект активации функции `createComparisonFunction()` не может быть уничтожен при ее завершении, потому что ссылка на него остается в цепочке областей видимости анонимной функции. Таким образом, при завершении функции `createComparisonFunction()` цепочка областей видимости ее контекста выполнения уничтожается, но ее объект активации остается в памяти вплоть до уничтожения анонимной функции:

```
// создание функции
var compareNames = createComparisonFunction("name");

// вызов функции
```

```
var result = compareNames({ name: "Nicholas" }, { name: "Greg"});

// завершение работы с функцией - теперь память может быть возвращена среде
compareNames = null;
```

Здесь функция сравнения назначается переменной `compareNames`. По завершении работы с функцией переменной `compareNames` присваивается значение `null`, что разрывает ее связь с функцией и позволяет сборщику мусора возвратить память среде. После этого все области видимости (кроме глобальной) можно безопасно уничтожить. Состояния цепочек областей видимости при вызове функции `compareNames()` в этом примере показаны на рис. 7.2.

Поскольку замыкание захватывает область видимости внешней функции, оно занимает больше памяти, чем обычная функция. Чрезмерное использование замыканий заметно увеличивает потребление памяти, поэтому применять их без необходимости не следует. Хотя оптимизирующие интерпретаторы JavaScript, такие как V8, пытаются освобождать память, заблокированную в замыканиях, при работе с замыканиями все равно нужно знать меру.

Замыкания и переменные

У такой конфигурации цепочки областей видимости есть один примечательный побочный эффект. Замыкание всегда получает последнее значение любой переменной из внешней функции. Помните, что замыкание хранит ссылку на весь объект переменных, а не только на конкретную переменную. Рассмотрим следующий пример:

Листинг ClosureExample01.htm


```
function createFunctions(){
    var result = new Array();

    for (var i=0; i < 10; i++){
        result[i] = function(){
            return i;
        };
    }

    return result;
}
```

Эта функция возвращает массив функций. На первый взгляд, каждая функция должна просто возвращать значение ее индекса: функция в нулевой позиции — 0, функция в первой позиции — 1, и т. д. Но на самом деле каждая функция возвращает 10. Поскольку у каждой функции в цепочке областей видимости есть объект активации `createFunctions()`, все функции ссылаются на одну и ту же переменную `i`. Когда функция `createFunctions()` завершается, `i` имеет значение 10, а раз все анонимные функции ссылаются на один и тот же объект переменных, значение `i` внутри каждой функции равно 10. Чтобы получить привычный результат, функцию можно изменить следующим образом:

Листинг ClosureExample02.htm

```
function createFunctions(){
    var result = new Array();

    for (var i=0; i < 10; i++){
        result[i] = function(num){
            return function(){
                return num;
            };
        }(i);
    }

    return result;
}
```

В этой версии `createFunctions()` все анонимные функции возвращают разные числа. Вместо того чтобы назначать замыкание непосредственно массиву, здесь определяется и сразу же вызывается анонимная функция. В качестве аргумента она принимает число, которое должна возвратить итоговая функция. Как аргумент в анонимную функцию передается переменная `i`. Поскольку аргументы функций передаются по значению, текущее значение `i` копируется в аргумент `num`. Внутри анонимной функции создается и возвращается замыкание, которое возвращает переменную `num`. В результате все функции в массиве `result` имеют собственные копии `num` и могут возвращать разные числа.

Объект this

При использовании объекта `this` внутри замыканий могут возникать непростые ситуации. Привязка объекта `this` во время выполнения происходит на основе контекста, в котором выполняется функция: в глобальных функциях `this` имеет значение `window` в нестрогом режиме и `undefined` в строгом, а в методах объектов `this` указывает на объект. Анонимные функции к объектам не привязываются, поэтому в них `this` указывает на `windows` в нестрогом режиме и имеет значение `undefined` в строгом, однако из-за синтаксиса замыканий это не всегда очевидно. Рассмотрим следующий пример:

Листинг ThisObjectExample01.htm


```
var name = "The Window";

var object = {
    name : "My Object",

    getNameFunc : function(){
        return function(){
            return this.name;
        };
    }
};

alert(object.getNameFunc()());    // "The Window" (в нестрогом режиме)
```

В этом коде мы создаем глобальную переменную `name` и объект со свойством `name`. У объекта также есть метод `getNameFunc()`, возвращающий анонимную функцию, которая в свою очередь возвращает `this.name`. Поскольку `getNameFunc()` возвращает функцию, вызов `object.getNameFunc()()` немедленно вызывает ее, в результате чего возвращается строка. Однако в нашем случае это строка `"The Window"`, являющаяся значением глобальной переменной `name`. Почему в анонимной функции не был использован объект `this` внешней функции?

Если помните, каждая вызванная функция автоматически получает специальные переменные `this` и `arguments`. У внутренней функции никогда нет прямого доступа к этим переменным, полученным внешней функцией. Чтобы в замыкании был доступен другой объект `this`, можно сохранить его в переменной, например:

Листинг ThisObjectExample02.htm


```
var name = "The Window";

var object = {
    name : "My Object",

    getNameFunc : function(){
        var that = this;
        return function(){
            return that.name;
        };
    }
};

alert(object.getNameFunc()());    // "My Object"
```

Этот пример отличается от предыдущего двумя выделенными строками. Перед определением анонимной функции объект `this` назначается переменной `that`, которая доступна в замыкании как переменная с уникальным именем, определенная в функции-контейнере. Даже после возвращения функции переменная `that` все еще связана с `object`, поэтому вызов `object.getNameFunc()()` возвращает строку `"My Object"`.

Так работают оба объекта: и this, и arguments. Если в замыкании нужен доступ к объекту arguments, находящемуся в области видимости функции-контейнера, сохраните ссылку на него в другой переменной, доступной замыканию.

В нескольких специальных случаях значение `this` может оказаться не таким, как вы думаете, например:

Листинг ThisObjectExample03.htm

```
var name = "The Window";

var object = {
    name : "My Object",
```

```
    getName: function(){
        return this.name;
    }
};
```

Здесь метод `getName()` просто возвращает значение `this.name`. Вот разные способы его вызова и соответствующие результаты:

```
object.getName();          // "My Object"
(object.getName)();        // "My Object"
(object.getName = object.getName)();    // "The Window" в нестрогом режиме
```

В первой строке метод `object.getName()`, вызванный обычным образом, возвращает строку `"My Object"`, потому что значение `this.name` совпадает с `object.name`. Во второй строке вызов `object.getName` в скобках может показаться ссылкой на функцию, но на самом деле используется то же значение `this`, потому что инструкции `object.getName` и `(object.getName)` эквивалентны. В третьей строке значением выражения присваивания является сама функция, поэтому значение `this` не сохраняется и возвращается строка `"The Window"`.

Едва ли вы будете намеренно использовать приемы, представленные во второй и третьей строках, однако полезно знать, как способно меняться значение `this` даже при таком незначительном изменении синтаксиса.

Утечки памяти

Замыкания вызывают специфические проблемы в браузерах Internet Explorer до версии 9, потому что в них реализованы разные способы сборки мусора для JScript- и COM-объектов (см. главу 4). Из-за сохранения области видимости, в которой находится HTML-элемент, иногда его невозможно уничтожить. Рассмотрим следующий пример:

```
function assignHandler(){
    var element = document.getElementById("someElement");
    element.onclick = function(){
        alert(element.id);
    };
}
```

В этом коде в качестве обработчика событий переменной `element` создается замыкание, в результате формируется циклическая ссылка (события обсуждаются в главе 13). Анонимная функция хранит ссылку на объект активации функции `assignHandler()`, что не позволяет уменьшить счетчик ссылок на `element`. Пока анонимная функция существует, счетчик ссылок на `element` не может быть обнулен, а это означает, что память никогда не будет освобождена. Проблему можно решить, немного изменив код:

```
function assignHandler(){
    var element = document.getElementById("someElement");
    var id = element.id;

    element.onclick = function(){
        alert(id);
    };

    element = null;
}
```

В этой версии в замыкании используется копия идентификатора элемента, что устраняет циклическую ссылку, но этого недостаточно для решения проблемы памяти. Помните: у замыкания есть ссылка на весь объект активации внешней функции, содержащий переменную `element`. Даже если замыкание не ссылается на `element` непосредственно, ссылка все равно хранится в этом объекте активации, поэтому еще нужно присвоить переменной `element` значение `null`. Это аннулирует ссылку на COM-объект и уменьшает его счетчик ссылок, позволяя возвратить память среде.

Имитация блочной области видимости

Как уже отмечалось, в JavaScript нет блочных областей видимости; это означает, что переменные, определенные внутри блока, на самом деле создаются в функции-контейнере. Рассмотрим пример:

Листинг BlockScopeExample01.htm


```
function outputNumbers(count){
    for (var i=0; i < count; i++){
        alert(i);
    }

    alert(i);    // count
}
```

Здесь определен цикл `for`, в котором переменная `i` инициализируется нулевым значением. В Java или C++ она была бы доступна только в блоке цикла `for` и уничтожилась бы при завершении цикла. Однако в JavaScript переменная `i` определяется как часть объекта активации `outputNumbers()` и доступна с этого момента внутри функции. Даже следующее некорректное повторное объявление переменной не уничтожает ее значение:

Листинг BlockScopeExample02.htm

```
function outputNumbers(count){
    for (var i=0; i < count; i++){
        alert(i);
    }

    var i;        // повторное объявление переменной
    alert(i);     // count
}
```

JavaScript никогда не сообщает, что переменная объявлена более одного раза, а просто игнорирует все последующие объявления (но не повторную инициализацию).

Для предотвращения таких проблем можно имитировать блочные области видимости с помощью анонимных функций.

Базовый синтаксис анонимной функции, используемой как блочная область видимости, которую часто называют *закрытой областью видимости* (private scope), таков:

```
(function(){
    // код блока
})();
```

Этот синтаксис определяет анонимную функцию, которая сразу же и вызывается, из-за чего ее иногда называют *функцией немедленного вызова* (immediately invoked function). То, что выглядит как объявление функции, заключается в скобки, чтобы показать, что на самом деле это выражение функции, а затем функция вызывается с помощью второй группы скобок в конце. Если этот синтаксис непонятен, взгляните на следующий пример:

```
var count = 5;
outputNumbers(count);
```

Здесь переменная `count` инициализируется значением 5. Конечно, она необязательна, потому что значение передается непосредственно в функцию. Чтобы сократить код, можно заменить переменную `count` при вызове функции значением 5:

```
outputNumbers(5);
```

Этот код работает так же, как и предыдущий, потому что переменная эквивалентна ее фактическому значению. Рассмотрим теперь такой пример:

```
var someFunction = function(){
    // код блока
};
someFunction();
```

Здесь определяется анонимная функция, которая назначается переменной `someFunction`. Затем функция вызывается, для чего после ее имени указываются скобки: `someFunction()`. В предыдущем примере мы смогли заменить переменную `count` ее фактическим значением, и здесь можно попытаться сделать то же самое. Однако такой вариант не сработает:

```
function(){
    // код блока
}();    // ошибка!
```

Этот код вызывает синтаксическую ошибку, потому что JavaScript интерпретирует ключевое слово `function` как начало объявления функции, а за объявлением не могут следовать скобки. Однако они *могут* следовать за *функцией-выражением*. Чтобы преобразовать объявление функции в функцию-выражение достаточно заключить объявление в скобки:

```
(function(){
    // код блока
})();
```

Такие закрытые области видимости можно использовать везде, где переменные нужны временно, например:

Листинг BlockScopeExample03.htm


```
function outputNumbers(count){
    (function () {
        for (var i=0; i < count; i++){
            alert(i);
        }
    })();

    alert(i);    // ошибка
}
```

В этой версии функции `outputNumbers()` создается закрытая область видимости для цикла `for`. Любые переменные, определенные в анонимной функции, уничтожаются, как только она завершается, так что переменная `i` недоступна вне цикла. Переменную `count` можно использовать внутри закрытой области видимости, потому что анонимной функции как замыканию доступны переменные области видимости функции-контейнера.

Этот прием часто применяется для ограничения количества переменных и функций, добавляемых в глобальную область видимости. Как правило, добавлять переменные и функции в глобальную область видимости нежелательно из-за потенциальных конфликтов имен, особенно если приложение разрабатывается коллективно. Благодаря закрытым областям видимости каждый разработчик может использовать собственные переменные, не засоряя глобальную область видимости. Рассмотрим пример:

```
(function(){

    var now = new Date();
    if (now.getMonth() == 0 && now.getDate() == 1){
        alert("Happy new year!");
    }

})();
```

Этот код определяет, запущен ли он 1 января, и если да — выводит на экран поздравление с Новым годом. Переменная `now` локальна для анонимной функции и не засоряет глобальную область видимости.

Отсутствие ссылок на анонимную функцию снимает проблему использования памяти в таких замыканиях, позволяя уничтожить цепочку областей видимости по завершении функции.

Закрытые переменные

Строго говоря, в JavaScript нет закрытых членов — все свойства объектов являются открытыми. Тем не менее концепция *закрытых переменных* (private variables) поддерживается. Любая переменная, определенная внутри функции, считается закрытой, поскольку она недоступна вне функции. Это относится к аргументам функций, локальным переменным, а также функциям, определенным внутри других функций. Рассмотрим пример:

```
function add(num1, num2){
    var sum = num1 + num2;
    return sum;
}
```

Закрытые переменные `num1`, `num2` и `sum` доступны внутри, но не вне этой функции. Если создать внутри нее замыкание, переменные будут доступны в нем по цепочке областей видимости, что позволяет создавать открытые методы с доступом к закрытым переменным.

Привилегированный метод (privileged method) — это открытый метод, предоставляющий доступ к закрытым переменным и (или) функциям. Есть два способа создания привилегированных методов для объектов. Первый — сделать это в конструкторе:

```
function MyObject(){

    // закрытые переменные и функции
    var privateVariable = 10;

    function privateFunction(){
        return false;
    }

    // привилегированные методы
    this.publicMethod = function (){
        privateVariable++;
        return privateFunction();
    };
}
```

При использовании этого паттерна сначала в конструкторе определяются все закрытые переменные и функции, после чего для доступа к ним создаются привилегированные методы. Этот паттерн работает, потому что при определении в конструкторе привилегированные методы становятся замыканиями, которым доступны все переменные и функции, определенные в области видимости конструктора. В данном примере переменная `privateVariable` и функция `privateFunction()` доступны только методу `publicMethod()`. Как только создан экземпляр `MyObject`, прямой доступ к `privateVariable` и `privateFunction()` невозможен — использовать их можно только через метод `publicMethod()`.

С помощью закрытых и привилегированных членов можно скрывать данные, которые не должны быть доступны напрямую:

Листинг PrivilegedMethodExample01.htm


```
function Person(name){

    this.getName = function(){
        return name;
    };

    this.setName = function (value) {
        name = value;
    };
}

var person = new Person("Nicholas");
alert(person.getName());        // "Nicholas"
person.setName("Greg");
alert(person.getName());        // "Greg"
```

В этом конструкторе определены привилегированные методы `getName()` и `setName()`. Каждый из них доступен вне конструктора и использует закрытую переменную `name`. Вне конструктора `Person` получить доступ к `name` невозможно. Поскольку оба метода определены внутри конструктора, они являются замыканиями и имеют доступ к `name` по цепочке областей видимости. Закрытая переменная `name` уникальна для каждого экземпляра `Person`, потому что методы создаются заново при каждом вызове конструктора. Как отмечено в главе 6, это один из недостатков, присущих паттерну Конструктор. Использование статических закрытых переменных с привилегированными методами решает эту проблему.

Статические закрытые переменные

Привилегированные методы можно также создавать для закрытых переменных или функций, определенных в закрытой области видимости:

```
(function(){

    // закрытые переменные и функции
    var privateVariable = 10;

    function privateFunction(){
        return false;
    }

    // конструктор
    MyObject = function(){
    };

    // открытые и привилегированные методы
    MyObject.prototype.publicMethod = function(){
```

```
        privateVariable++;
        return privateFunction();
    };

})();
```

В этом паттерне закрытая область видимости содержит конструктор и его методы. Первыми определяются закрытые переменные и функции, за ними следуют конструктор и открытые методы. Открытые методы определяются в прототипе, как и в обычном паттерне Прототип. Обратите внимание, что конструктор определяется в функции-выражении, а не в объявлении. При объявлении всегда создается локальная функция, что в данном случае нежелательно. По этой же причине перед `MyObject` не указано ключевое слово `var`. Помните, что при инициализации необъявленной переменной всегда создается глобальная переменная, так что объект `MyObject` становится глобальным и доступным вне закрытой области видимости. Имейте также в виду, что присвоение значения необъявленной переменной в строгом режиме приводит к ошибке.

Основное различие между этим паттерном и предыдущим состоит в том, что в этот раз закрытые переменные и функции являются общими для всех экземпляров. Поскольку привилегированный метод определен в прототипе, все экземпляры используют одну и ту же функцию, а сам привилегированный метод как замыкание всегда содержит ссылку на область видимости функции-контейнера. Рассмотрим пример:

Листинг PrivilegedMethodExample02.htm


```
(function(){

    var name = "";

    Person = function(value){
        name = value;
    };

    Person.prototype.getName = function(){
        return name;
    };

    Person.prototype.setName = function (value){
        name = value;
    };
})();

var person1 = new Person("Nicholas");
alert(person1.getName());    // "Nicholas"
person1.setName("Greg");
alert(person1.getName());    // "Greg"

var person2 = new Person("Michael");
alert(person1.getName());    // "Michael"
alert(person2.getName());    // "Michael"
```

Здесь закрытая переменная `name` доступна конструктору `Person`, а также методам `getName()` и `setName()`, потому что в этом паттерне она становится статической и используется всеми экземплярами. Это означает, что вызов `setName()` для одного экземпляра влияет на все остальные экземпляры. При вызове `setName()` или создании экземпляра `Person` переменная `name` получает новое значение, которое после этого возвращается любым экземпляром.

Создание статических закрытых переменных обеспечивает возможность многократного использования кода с помощью прототипов, но при этом у отдельных экземпляров нет собственных закрытых переменных. В конечном счете, выбор между закрытыми переменными экземпляра и статическими закрытыми переменными зависит от конкретных требований.

Чем дальше в цепочке областей видимости находится переменная, тем сильнее замедляется поиск из-за использования замыканий и закрытых переменных.

Паттерн Модуль

Предыдущие паттерны создают закрытые переменные и привилегированные методы для пользовательских типов. *Паттерн Модуль* (module pattern), описанный Дугласом Крокфордом, делает то же самое для объектов-одиночек. *Одиночка* (singleton) — это объект, который может быть только единственным. Для создания одиночек в JavaScript традиционно используют нотацию литералов объектов, например:

```
var singleton = {
    name : value,
    method : function () {
        // код метода
    }
};
```

Паттерн Модуль расширяет объект-одиночку, позволяя использовать в нем закрытые переменные и привилегированные методы:

```
var singleton = function(){

    // закрытые переменные и функции
    var privateVariable = 10;

    function privateFunction(){
        return false;
    }

    // привилегированные/открытые методы и свойства
    return {

        publicProperty: true,

        publicMethod : function(){
```

```
            privateVariable++;
            return privateFunction();
        }

    };
}();
```

В паттерне Модуль используется анонимная функция, которая возвращает объект. Внутри этой функции сначала определяются закрытые переменные и функции, после чего возвращается литерал объекта, содержащий только открытые свойства и методы. Поскольку объект определен внутри анонимной функции, закрытые переменные и функции доступны всем открытым методам. По сути, литерал объекта определяет открытый интерфейс для объекта-одиночки. Это может быть полезно, если требуется инициализировать одиночку и обеспечить доступ к его закрытым переменным, например:

Листинг ModulePatternExample01.htm


```
var application = function(){

    // закрытые переменные и функции
    var components = new Array();

    // инициализация
    components.push(new BaseComponent());

    // открытый интерфейс
    return {
        getComponentCount : function(){
            return components.length;
        },

        registerComponent : function(component){
            if (typeof component == "object"){
                components.push(component);
            }
        }
    };
}();
```

В веб-приложениях объект-одиночку часто используют для управления данными уровня приложения. Так, объект-одиночка `application` из приведенного примера служит для управления компонентами. При создании объекта строится закрытый массив `components`, в который добавляется новый экземпляр `BaseComponent` (он используется исключительно для демонстрации инициализации, поэтому его код не важен). Доступ к массиву `components` осуществляется с помощью привилегированных методов `getComponentCount()` и `registerComponent()`. Первый из них просто возвращает количество зарегистрированных компонентов, а второй регистрирует новый компонент.

Паттерн Модуль полезен в ситуациях вроде этой, когда нужно создать один объект, инициализировать его некоторыми данными и предоставить открытые методы для

доступа к его закрытым данным. Каждый объект-одиночка, создаваемый таким образом, является экземпляром типа `Object`, поскольку представляется литералом объекта. Это непоследовательно, потому что одиночки обычно используются глобально, а не передаются в функции как аргументы; соответственно, определять их тип с помощью оператора `instanceof` не требуется.

Расширенный паттерн Модуль

Альтернативный способ применения паттерна Модуль подразумевает расширение возможностей объекта перед его возвращением. Этот паттерн полезен, если объект-одиночка должен быть экземпляром конкретного типа, но в него требуется добавить свойства и (или) методы, например:

```
var singleton = function(){

    // закрытые переменные и функции
    var privateVariable = 10;

    function privateFunction(){
        return false;
    }

    // создание объекта
    var object = new CustomType();

    // добавление привилегированных/открытых свойств и методов
    object.publicProperty = true;

    object.publicMethod = function(){
        privateVariable++;
        return privateFunction();

    };

    // возвращение объекта
    return object;
}();
```

Если бы объект `application` в примере с паттерном Модуль был экземпляром `BaseComponent`, можно было бы использовать следующий код:

Листинг ModuleAugmentationPatternExample01.htm


```
var application = function(){

    // закрытые переменные и функции
    var components = new Array();

    // инициализация
    components.push(new BaseComponent());
```

```
    // создание локальной копии application
    var app = new BaseComponent();

    // открытый интерфейс
    app.getComponentCount = function(){
        return components.length;
    };

    app.registerComponent = function(component){
        if (typeof component == "object"){
            components.push(component);
        }
    };

    // возвращение объекта application
    return app;
}();
```

Как и в предыдущем примере, в этой версии объекта-одиночки `application` сначала определяются закрытые переменные. Ее главное отличие — создание локальной переменной `app` типа `BaseComponent`, которая в итоге станет объектом `application`. Затем в объект `app` добавляются открытые методы для доступа к закрытым переменным, после чего он возвращается и назначается переменной `application`.

Резюме

Функции-выражения — это полезные конструкции JavaScript, которые делают возможным по-настоящему динамическое программирование без именования функций. Такие анонимные функции, называемые также лямбда-функциями, лежат в основе ряда эффективных приемов. Давайте вспомним ключевые моменты главы.

- В отличие от объявлений функций, функции-выражения не требуют указания имени функции. Поэтому такие выражения называют анонимными функциями.
- Отсутствие четко определенного способа ссылки на функцию делает рекурсивные функции более сложными.
- В нестрогом режиме для рекурсивного вызова функции можно использовать свойство `arguments.callee` вместо имени функции, которое может измениться.

При определении функции внутри другой функции формируется замыкание, которому доступны все переменные в функции-контейнере. Перечислим некоторые характеристики замыканий.

- Цепочка областей видимости замыкания содержит объекты переменных локальной функции, функции-контейнера и глобального контекста.
- Как правило, при завершении функции ее область видимости и все ее переменные уничтожаются.

- Если функция возвращает замыкание, ее область видимости остается в памяти, пока существует замыкание.

С помощью замыканий можно имитировать в JavaScript блочные области видимости, которые изначально не поддерживаются:

- функция может быть создана и сразу же после создания вызвана с выполнением ее кода и без всякой ссылки на нее;
- в результате такого приема все переменные в функции уничтожаются, если явно не присвоить их переменным во внешней области видимости.

Замыкания можно также использовать для создания закрытых переменных в объектах.

- Хотя JavaScript формально не поддерживает закрытые свойства объектов, с помощью замыканий можно реализовать открытые методы с доступом к переменным, определенным в закрытой области видимости.
- Открытые методы с доступом к закрытым переменным называются привилегированными.
- Привилегированные методы можно создавать для пользовательских типов с помощью паттерна Конструктор или Прототип, а также для объектов-одиночек, используя паттерны Модуль с обычными или расширенными возможностями.

Функции-выражения и замыкания — очень эффективные JavaScript-конструкции, используемые для решения самых разных задач. Однако помните, что замыкания вынуждают хранить в памяти дополнительные области видимости, увеличивая потребление ресурсов.

8 Объектная модель браузера

- Объект `window` — основа BOM
- Управление окнами, фреймами и всплывающими окнами
- Получение сведений о странице с помощью объекта `location`
- Получение сведений о браузере с помощью объекта `navigator`

Объектная модель браузера (Browser Object Model, BOM) описана в ECMAScript как ядро JavaScript, но правильнее было бы охарактеризовать ее как основу для использования JavaScript в Интернете. BOM-объекты обеспечивают доступ к функционалу браузера независимо от контента веб-страницы. Тема BOM интересна и одновременно сложна, потому что из-за длительного отсутствия спецификации производители браузеров свободно расширяли BOM по своему усмотрению. Многие элементы, схожие в разных браузерах, стали стандартами де-факто, которые соблюдаются по сей день из соображений взаимной совместимости. Чтобы стандартизировать эти фундаментальные аспекты JavaScript, консорциум W3C определил основные BOM-элементы в спецификации HTML5.

Объект window

В основе BOM лежит объект `window`, который представляет экземпляр браузера и имеет двойное назначение. С одной стороны, это JavaScript-интерфейс для доступа к окну браузера, а с другой — ECMAScript-объект `Global`. Это означает, что все объекты, переменные и функции, определенные в коде веб-страницы, используют объект `window` как глобальный и могут вызывать его методы, такие как `parseInt()`.

Глобальная область видимости

Поскольку объект `window` дублируется как ECMAScript-объект `Global`, все переменные и функции, объявленные глобально, становятся его свойствами и методами, например:

```
var age = 29;
function sayAge(){
    alert(this.age);
}

alert(window.age);        // 29
sayAge();                 // 29
window.sayAge();          // 29
```

При определении в глобальной области видимости переменная `age` и функция `sayAge()` автоматически добавляются к объекту `window`. Таким образом, переменная `age` доступна также как свойство `window.age`, а функция `sayAge()` — как `window.sayAge()`. Поскольку `sayAge()` существует в глобальной области видимости, вызов `this.age` проецируется на `window.age`.

Несмотря на то что глобальные переменные становятся свойствами объекта `window`, между ними есть небольшое различие. Глобальные переменные нельзя удалить с помощью оператора `delete`, а свойства, определенные непосредственно для объекта `window`, можно:

Листинг DeleteOperatorExample01.htm


```
var age = 29;
window.color = "red";

// ошибка в IE до версии 9, false во всех остальных браузерах
delete window.age;

// ошибка в IE до версии 9, true во всех остальных браузерах
delete window.color;      // true

alert(window.age);        // 29
alert(window.color);      // undefined
```

У свойств, добавленных к объекту `window` с помощью ключевого слова `var`, атрибут `[[Configurable]]` имеет значение `false`, поэтому их нельзя удалить, используя оператор `delete`. В Internet Explorer 8 и более ранних версий попытка сделать это приводит к ошибке независимо от того, как было создано свойство. В Internet Explorer 9 и более поздних версий ошибка не возникает.

Попытка доступа к необъявленной переменной также приводит к ошибке, но ее можно предотвратить, проверив наличие потенциально необъявленной переменной у объекта `window`:

```
// ошибка, потому что переменная oldValue не объявлена
var newValue = oldValue;

// ошибки нет, потому что выполняется обращение к свойству
// newValue получает значение undefined
var newValue = window.oldValue;
```

Многие JavaScript-объекты, которые считаются глобальными, например `location` и `navigator` (оба они обсуждаются в этой главе), на самом деле являются свойствами объекта `window`.

Internet Explorer для Windows Mobile не поддерживает непосредственное создание свойств и методов объекта window с использованием синтаксиса window.свойство = значение, но все переменные и функции, объявленные глобально, становятся членами объекта window.

Отношения окон и фреймов

Если страница содержит фреймы, каждый из них имеет собственный объект `window` и хранится в коллекции `frames`. В ней объекты `window` индексируются по номеру (слева направо, а затем сверху вниз; нумерация начинается с нуля) и по имени фрейма. У каждого объекта `window` есть свойство `name`, содержащее имя фрейма. Рассмотрим пример:

Листинг FramesetExample01.htm

```
<html>
    <head>
        <title>Frameset Example</title>
    </head>
    <frameset rows="160,*">
        <frame src="frame.htm" name="topFrame">
        <frameset cols="50%,50%">
            <frame src="anotherframe.htm" name="leftFrame">
            <frame src="yetanotherframe.htm" name="rightFrame">
        </frameset>
    </frameset>
</html>
```

Этот код создает набор фреймов с одним фреймом вверху и двумя фреймами внизу. Верхний фрейм доступен как `window.frames[0]` или `window.frames["topFrame"]`, но вы, вероятно, использовали бы объект `top` вместо `window` (например, `top.frames[0]`).

Объект `top` всегда соответствует самому верхнему (дальнему от центра) фрейму, которым является само окно браузера. Это обеспечивает правильную исходную позицию для доступа к остальным фреймам. Если во фрейме есть ссылка на объект `window`, она указывает на уникальный экземпляр `window` этого, а не самого верхнего фрейма. На рис. 8.1 показано, как из кода самого верхнего окна можно обращаться к фреймам из предыдущего примера.

Рис. 8.1

Другой объект `window` называется `parent`. Он всегда указывает на непосредственный родительский фрейм текущего фрейма. В некоторых случаях им может быть объект `top` (например, если фреймов нет, при этом объекты `parent` и `top` совпадают с `window`). Рассмотрим пример:

Листинг frameset1.htm


```
<html>
    <head>
        <title>Frameset Example</title>
    </head>
    <frameset rows="100,*">
        <frame src="frame.htm" name="topFrame">
        <frameset cols="50%,50%">
            <frame src="anotherframe.htm" name="leftFrame">
            <frame src="anotherframeset.htm" name="rightFrame">
        </frameset>
    </frameset>
</html>
```

Этот набор фреймов включает фрейм, содержащий другой набор фреймов со следующим кодом:

Листинг anotherframeset.htm

```
<html>
    <head>
        <title>Frameset Example</title>
    </head>
```

```
    <frameset cols="50%,50%">
        <frame src="red.htm" name="redFrame">
        <frame src="blue.htm" name="blueFrame">
    </frameset>
</html>
```

Когда первый набор фреймов загружается в браузер, он загружает другой набор фреймов в `rightFrame`. В коде, который относится к `redFrame` (или `blueFrame`), объект `parent` указывает на `rightFrame`. Если же код относится к фрейму `topFrame`, объекту `parent` соответствует фрейм `top`, потому что родительским для него является внешний фрейм. На рис. 8.2 показаны значения разных объектов `window` в этом примере.

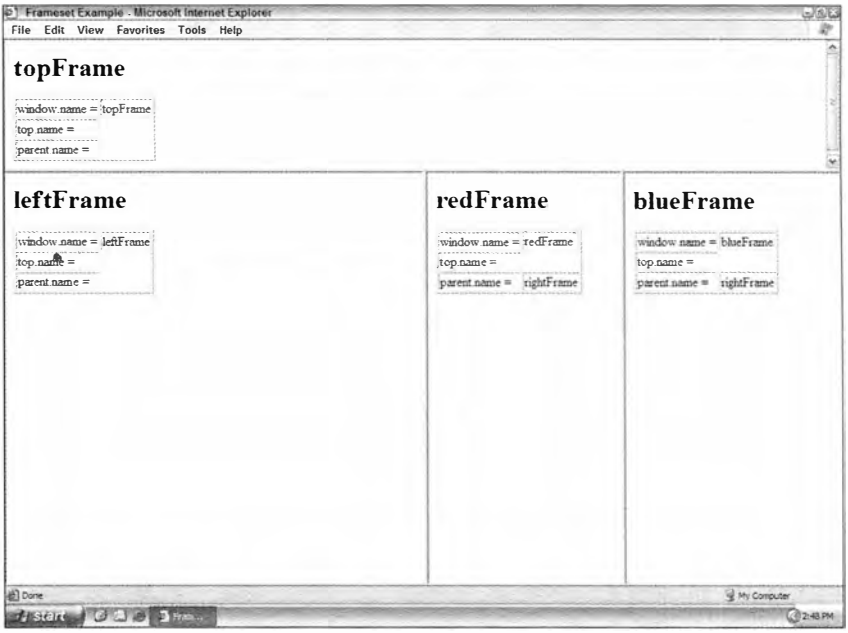

Рис. 8.2

Заметьте, что у самого верхнего объекта `window` свойство `name` никогда не имеет значения, если только окно не было открыто с помощью метода `window.open()`, который мы обсудим позже.

Есть еще один объект `window`, который называется `self` и всегда указывает на `window`. Каждый из упомянутых объектов можно использовать вместо другого. Хотя у `self` нет отдельного значения, он добавлен к объектам `top` и `parent` ради единообразия.

Каждый из этих объектов на самом деле является свойством объекта `window` и доступен как `window.parent`, `window.top` и т. д. Это означает, что объекты `window` можно сцеплять, например `window.parent.parent.frames[0]`.

При использовании фреймов браузер содержит несколько объектов Global. Глобальные переменные определяются в каждом фрейме как свойства его объекта window. Поскольку любой объект window содержит конструкторы встроенных типов, каждый фрейм имеет собственные версии конструкторов, которые различаются. Например, конструктор top.Object не эквивалентен top.frames[0].Object. Это влияет на использование оператора instanceof, когда объекты передаются между фреймами.

Расположение окна

Расположение объекта `window` можно определять и изменять с помощью нескольких свойств и методов. Internet Explorer, Safari, Opera и Chrome предоставляют для этого свойства `screenLeft` и `screenTop`, которые указывают расположение окна относительно левого и верхнего краев экрана соответственно. В Firefox для этого используются свойства `screenX` и `screenY`, которые также поддерживаются в Safari и Chrome. Эти свойства поддерживаются и в Opera, но применять их не следует, потому что они не соответствуют свойствам `screenLeft` и `screenTop`. Следующий код определяет левую и верхнюю позиции окна в любом браузере:

Листинг WindowPositionExample01.htm


```
var leftPos = (typeof window.screenLeft == "number") ?
                  window.screenLeft : window.screenX;
var topPos = (typeof window.screenTop == "number") ?
                  window.screenTop : window.screenY;
```

Здесь мы с помощью тернарного оператора выясняем, существуют ли свойства `screenLeft` и `screenTop`. Если да (как в Internet Explorer, Safari, Opera и Chrome), они назначаются переменным. Если их нет (как в Firefox), используются свойства `screenX` и `screenY`.

Работая с этими значениями, нужно помнить о некоторых нюансах. В Internet Explorer, Opera и Chrome свойства `screenLeft` и `screenTop` определяют расстояние от левого и верхнего краев экрана до области просмотра страницы, представленной объектом `window`. Если объект `window` является самым верхним и окно браузера находится в самом верху экрана (с нулевой координатой `y`), значением `screenTop` будет высота панелей инструментов (в пикселях) над областью просмотра страницы. В Firefox и Safari эти значения рассчитываются относительно всего окна браузера, то есть у окна, примыкающего к верхнему краю экрана, свойство верхней позиции равно нулю.

Хуже того, Firefox, Safari и Chrome всегда возвращают значения `top.screenX` и `top.screenY` для каждого фрейма на странице. Даже если страница смещена на какой-то интервал, при использовании свойств `screenX` и `screenY` эти значения возвращаются относительно объекта `window`. Internet Explorer и Opera предоставляют точные координаты фреймов относительно краев экрана.

Таким образом, хотя мы не можем правильно определить левую и верхнюю координаты окна во всех браузерах, мы можем точно переместить окно в новое положение

с помощью методов moveTo() и moveBy(), которые принимают по два аргумента. Метод moveTo() принимает целевые координаты x и y, а метод moveBy() — смещения в каждом направлении. Рассмотрим пример:

```
// перемещение окна в левый вверхний угол
window.moveTo(0,0);

// перемещение окна вниз на 100 пикселей
window.moveBy(0, 100);

// перемещение окна в позицию (200, 300)
window.moveTo(200, 300);

// перемещение окна влево на 50 пикселей
window.moveBy(-50, 0);
```

Эти методы отключены по умолчанию для главного окна браузера в Internet Explorer 7+, Chrome и Opera. Кроме того, они работают только с самым верхним объектом window, но не с фреймами.

Размеры окна

Определить размеры окна без привязки к определенным браузерам непросто. Internet Explorer 9+, Firefox, Safari, Opera и Chrome предоставляют для этого свойства innerWidth, innerHeight, outerWidth и outerHeight. В Internet Explorer 9+, Safari, Firefox и Chrome свойства outerWidth и outerHeight возвращают размеры самого окна браузера (независимо от того, запрашиваются ли они у самого верхнего окна или у фрейма). В Opera эти значения определяют размеры области просмотра страницы. Свойства innerWidth и innerHeight возвращают размеры области просмотра страницы внутри окна браузера (не учитывая границы и панели инструментов).

В Internet Explorer 8 и более ранних версий нельзя получить текущие размеры окна браузера, но можно запросить сведения об области просмотра страницы из DOM.

Свойства document.documentElement.clientWidth и document.documentElement.clientHeight определяют ширину и высоту области просмотра страницы в Internet Explorer, Firefox, Safari, Opera и Chrome. В Internet Explorer 6 эти свойства доступны только в стандартном режиме; в режиме совместимости эту же информацию можно получить с помощью свойств document.body.clientWidth и document.body.clientHeight. В Chrome в режиме совместимости размеры области просмотра содержатся в свойствах clientWidth и clientHeight объектов document.documentElement и document.body.

Короче говоря, надежно определить размеры самого окна браузера нельзя, но зато можно получить размеры области просмотра страницы:

Листинг WindowSizeExample01.htm

```
var pageWidth = window.innerWidth,
    pageHeight = window.innerHeight;
```

```
if (typeof pageWidth != "number"){
    if (document.compatMode == "CSS1Compat"){
        pageWidth = document.documentElement.clientWidth;
        pageHeight = document.documentElement.clientHeight;
    } else {
        pageWidth = document.body.clientWidth;
        pageHeight = document.body.clientHeight;
    }
}
```

В этом коде переменным `pageWidth` и `pageHeight` присваиваются первоначальные значения `window.innerWidth` и `window.innerHeight` соответственно. Затем мы проверяем, является ли значение `pageWidth` числом; если нет, нужно проверить, работает ли браузер в стандартном режиме, для чего используется свойство `document.compatMode`, которое подробно обсуждается в главе 11. Если включен стандартный режим, используются значения `document.documentElement.clientWidth` и `document.documentElement.clientHeight`, в противном случае — значения `document.body.clientWidth` и `document.body.clientHeight`.

Для мобильных устройств свойства `window.innerWidth` и `window.innerHeight` определяют размеры визуальной области просмотра, то есть области страницы, видимой на экране. Internet Explorer для мобильных устройств не поддерживает эти свойства, но предоставляет ту же информацию в виде свойств `document.documentElement.clientWidth` и `document.documentElement.clientHeight`, значения которых изменяются при изменении масштаба страницы.

В других браузерах для мобильных устройств свойства объекта `document.documentElement` определяют размеры области просмотра макета, то есть фактические размеры визуализированной страницы (в отличие от визуальной области просмотра, которая охватывает только небольшую часть всей страницы). Internet Explorer для мобильных устройств хранит эти значения в свойствах `document.body.clientWidth` и `document.body.clientHeight`, которые остаются постоянными при изменении масштаба.

Из-за этих различий браузеров для мобильных устройств и настольных компьютеров перед выбором нужных свойств следует сначала определить систему, с которой работает пользователь.

Тема областей просмотра на мобильных устройствах непроста и имеет много нюансов и исключений. Петер-Пол Кох (Peter-Paul Koch), консультант по разработке мобильных приложений, раскрыл ее в своей статье, доступной на сайте http://quirksmode.org/mobile/viewports2.html. Ознакомьтесь с ней, если вы разрабатываете приложения для мобильных устройств.

Размеры окна браузера можно изменить с помощью методов `resizeTo()` и `resizeBy()`, которые принимают по два аргумента. Метод `resizeTo()` принимает новые значения ширины и высоты, а `resizeBy()` — изменения каждого размера, например:

```
// задание размеров 100 x 100
window.resizeTo(100, 100);

// задание размеров 200 x 150
window.resizeBy(100, 50);

// задание размеров 300 x 300
window.resizeTo(300, 300);
```

Как и методы перемещения окна, эти методы по умолчанию отключены в Internet Explorer 7+, Chrome и Opera и работают только с самым верхним объектом `window`.

Открытие окон и навигация

Метод `window.open()` позволяет перейти по указанному URL-адресу и открыть новое окно браузера. Он принимает четыре аргумента: URL-адрес страницы, которую нужно загрузить, целевое окно, строку параметров и логическое значение, указывающее, должна ли новая страница заменить текущую в журнале браузера. Обычно используют только три первых аргумента; последний указывают, если не нужно открывать новое окно.

Если вторым аргументом метода `window.open()` является имя уже существующего окна или фрейма, страница по указанному URL-адресу загружается в это окно или фрейм, например:

```
//то же, что и <a href="http://www.wrox.com" target="topFrame"></a>
window.open("http://www.wrox.com/", "topFrame");
```

Выполнение этого кода аналогично щелчку на ссылке, у которой атрибут `href` имеет значение `"http://www.wrox.com"`, а атрибут `target` — `"topFrame"`. При наличии окна или фрейма с именем `"topFrame"` страница загружается в него, в противном случае создается новое окно с именем `"topFrame"`. Вторым аргументом также может быть одно из специальных имен окон: `_self`, `_parent`, `_top` или `_blank`.

Всплывающие окна

Если второй аргумент метода `window.open` не соответствует именам существующих окон и фреймов, метод создает окно или вкладку на основе строки, переданной ему в качестве третьего аргумента. Если этот аргумент отсутствует, метод открывает в браузере новое окно или вкладку (в зависимости от того, как настроен браузер) с параметрами, предлагаемыми по умолчанию. Элементы окна или вкладки, такие как панели инструментов, адресная строка и строка состояния, также отображаются согласно параметрам, предлагаемым по умолчанию. Если указано, что новое окно открывать не следует, третий аргумент игнорируется.

Третьим аргументом является строка параметров отображения нового окна, разделенных запятыми. Допустимые параметры указаны в таблице.

Параметр	Значение	Описание
fullscreen	"yes" или "no"	Указывает, нужно ли создать окно браузера развернутым во весь экран (работает только в Internet Explorer)
height	Число	Начальная высота нового окна. Не может быть меньше 100
left	Число	Начальная левая координата нового окна. Не может быть отрицательным числом
location	"yes" или "no"	Указывает, нужно ли отобразить адресную строку. Значение по умолчанию зависит от браузера. Если задано значение "no", адресная строка может быть либо скрыта, либо отключена в зависимости от браузера
menubar	"yes" или "no"	Указывает, нужно ли отобразить панель меню. По умолчанию "no"
resizable	"yes" или "no"	Указывает, можно ли изменять размеры нового окна, перетаскивая его границы. По умолчанию "no"
scrollbars	"yes" или "no"	Указывает, можно ли прокручивать новое окно, если контент не помещается в области просмотра. По умолчанию "no"
status	"yes" или "no"	Указывает, нужно ли отобразить строку состояния. Значение по умолчанию зависит от браузера
toolbar	"yes" или "no"	Указывает, нужно ли отобразить панель инструментов. По умолчанию "no"
top	Число	Начальная верхняя координата нового окна. Не может быть отрицательным числом
width	Число	Начальная ширина нового окна. Не может быть меньше 100

Любые из этих параметров можно указать как набор разделенных запятыми пар имен и значений. Имя и значение в каждой паре разделяются знаком равенства (пробелы в строке параметров не допускаются). Вот пример:

```
window.open("http://www.wrox.com/","wroxWindow",
            "height=400,width=400,top=10,left=10,resizable=yes");
```

Этот код в 10 пикселях от верхнего и левого краев экрана открывает новое окно с размерами 400 × 400, которые можно изменять.

Метод `window.open()` возвращает ссылку на созданное окно. Это такой же объект `window`, как и любые другие, только обычно лучше контролируемый. Например, браузеры, которые по умолчанию не позволяют изменять размеры главного окна или перемещать его, могут разрешать это для окон, созданных методом `window.open()`.

Используя возвращенный объект, можно управлять новым открытым окном так же, как и любым другим, например:

```
var wroxWin = window.open("http://www.wrox.com/","wroxWindow",
                    "height=400,width=400,top=10,left=10,resizable=yes");

// изменение размеров окна
wroxWin.resizeTo(500, 500);

// перемещение окна
wroxWin.moveTo(100, 100);
```

Закрыть новое окно можно, вызвав метод `close()`:

```
wroxWin.close();
```

Этот метод работает только со всплывающими окнами, созданными методом `window.open()`. Закрыть главное окно браузера без подтверждения пользователя невозможно. Однако всплывающие окна могут закрывать себя сами без подтверждения пользователя, вызывая метод `top.close()`. После закрытия окна ссылка на него остается доступной, но годится только для проверки свойства `closed`:

```
wroxWin.close();
alert(wroxWin.closed);    // true
```

Созданное окно ссылается на окно, которое его открыло, с помощью свойства `opener`. Оно определено только для самого верхнего объекта `window` (то есть `top`) всплывающего окна и представляет собой указатель на окно или фрейм, для которого был вызван метод `window.open()`, например:

```
var wroxWin = window.open("http://www.wrox.com/","wroxWindow",
                    "height=400,width=400,top=10,left=10,resizable=yes");

alert(wroxWin.opener == window);    // true
```

У всплывающего окна есть указатель на исходное окно, но обратное неверно. Окна не следят за тем, какие окна они породили, так что при необходимости вы сами должны их отслеживать.

Некоторые браузеры, например Internet Explorer 8+ и Google Chrome, пытаются запускать отдельный процесс для каждой новой вкладки. Когда одна вкладка открывает другую, объектам `window` нужно взаимодействовать друг с другом, поэтому в такой ситуации вкладки не могут выполняться в разных процессах. В Chrome можно указать, что для вкладки следует создать отдельный процесс, присвоив свойству `opener` значение `null`:

```
var wroxWin = window.open("http://www.wrox.com/","wroxWindow",
                    "height=400,width=400,top=10,left=10,resizable=yes");

wroxWin.opener = null;
```

Присвоение значения `null` свойству `opener` указывает браузеру, что новой вкладке не нужно взаимодействовать с исходной, так что она может работать как отдельный процесс. После разрыва связи между вкладками восстановить ее невозможно.

Ограничения безопасности

В свое время по Интернету прокатилась эпидемия рекламных всплывающих окон, которые часто выдавали за системные диалоговые окна, чтобы пользователи не могли распознать недобросовестную рекламу. В ответ на это производители браузеров начали ограничивать возможности настройки всплывающих окон.

В Internet Explorer 6 операционной системы Windows XP с пакетом обновления 2 были реализованы такие меры безопасности, как запрет на создание всплывающих окон и их перемещение за пределы экрана, а также на отключение строки состояния. Начиная с Internet Explorer 7 по умолчанию запрещено отключать адресную строку, перемещать всплывающие окна и изменять их размеры. В Firefox 1 была отключена возможность блокировать строку состояния, из-за чего она отображалась во всех всплывающих окнах независимо от строки параметров, переданной в метод `window.open()`. В Firefox 3 то же самое было сделано для адресной строки. Opera открывает всплывающие окна только в главном окне браузера, но блокирует их при опасности спутать их с системными диалоговыми окнами.

Кроме того, браузеры разрешают создание всплывающего окна только после действия пользователя. Например, вместо вызова метода `window.open()` во время загрузки страницы может быть выведено сообщение об ошибке, потому что всплывающие окна разрешено открывать только в ответ на щелчок мышью или нажатие клавиши.

В Chrome принят другой подход к обработке всплывающих окон, появление которых не было инициировано пользователем. Вместо того чтобы блокировать их, браузер отображает только строку заголовка всплывающего окна и помещает ее в правом нижнем углу главного окна.

Internet Explorer снимает некоторые ограничения, связанные со всплывающими окнами, при отображении веб-страниц, сохраненных на жестком диске компьютера. Если тот же код запускается с сервера, ограничения снова вступают в силу.

Блокирование всплывающих окон

Большинство браузеров содержат встроенные средства блокирования всплывающих окон, а для тех, которые их не содержат, доступны программы вроде Yahoo! Toolbar. Так или иначе, большинство непредвиденных всплывающих окон блокируются, при этом происходит одно из двух. Если окно блокируется средством, встроенным в браузер, метод `window.open()` обычно возвращает значение `null`, по которому можно узнать, что случилась блокировка, например:

```
var wroxWin = window.open("http://www.wrox.com", "_blank");
if (wroxWin == null){
```

```
    alert("The popup was blocked!");
}
```

Если всплывающее окно блокируется надстройкой браузера или другой программой, метод `window.open()` обычно генерирует ошибку. Следовательно, чтобы правильно определить, что всплывающее окно было заблокировано, необходимо проверить значение, возвращаемое методом `window.open()`, и заключить его вызов в блок `try-catch`:

Листинг PopupBlockerExample01.htm


```
var blocked = false;

try {
    var wroxWin = window.open("http://www.wrox.com", "_blank");
    if (wroxWin == null){
        blocked = true;
    }
} catch (ex){
    blocked = true;
}

if (blocked){
    alert("The popup was blocked!");
}
```

Этот код определяет, что вызов `window.open()` был заблокирован любым из указанных способов. Имейте в виду, что это не препятствует браузеру вывести на экран собственное сообщение о блокировании всплывающего окна.

Интервалы и тайм-ауты

JavaScript работает в браузере в однопоточном режиме, но позволяет планировать выполнение кода в конкретные моменты времени с помощью тайм-аутов и интервалов. Тайм-ауты используются для запуска кода после указанного промежутка времени, а интервалы — для периодического запуска.

Тайм-аут можно задать с помощью метода `window.setTimeout()`, который принимает два аргумента: выполняемый код и интервал (в миллисекундах), по прошествии которого нужно запустить этот код. Первым аргументом может быть либо строка с JavaScript-кодом (как при использовании метода `eval()`), либо функция. Например, оба следующих фрагмента выводят на экран оповещение через 1 секунду:

Листинг TimeoutExample01.htm


```
// не делайте так!
setTimeout("alert('Hello world!')", 1000);

//предпочтительный способ
setTimeout(function() {
    alert("Hello world!");
}, 1000);
```

Хотя оба варианта приемлемы, использовать строку как первый аргумент не рекомендуется, так как это снижает быстродействие.

Второй аргумент, время ожидания в миллисекундах, не определяет точный момент выполнения указанного фрагмента. Поскольку JavaScript-код выполняется в однопоточном режиме, в каждый момент обрабатывается только одна инструкция. Для управления выполнением кода используется очередь задач JavaScript, которые запускаются в том же порядке, в каком были добавлены в очередь. Второй аргумент метода `setTimeout()` указывает интерпретатору JavaScript добавить задачу в очередь через указанное время. Если очередь пуста, этот код выполняется незамедлительно, в противном случае он должен дождаться своей очереди.

Метод `setTimeout()` возвращает числовой идентификатор тайм-аута. Он уникально идентифицирует запланированный код и позволяет отменить тайм-аут. Чтобы отменить тайм-аут, который еще не был обработан, вызовите метод `clearTimeout()`, передав ему идентификатор тайм-аута, например:

Листинг TimeoutExample02.htm

```
// задание тайм-аута
var timeoutId = setTimeout(function() {
    alert("Hello world!");
}, 1000);

// тайм-аут больше не нужен
clearTimeout(timeoutId);
```

Если метод `clearTimeout()` вызывается до истечения указанного времени, тайм-аут полностью отменяется. Вызов `clearTimeout()` после выполнения запланированного кода ни на что не влияет.

Весь запланированный код тайм-аута выполняется в глобальной области видимости, так что объект `this` внутри функции всегда указывает на `window` в нестрогом режиме и имеет значение `undefined` в строгом.

Интервал работает подобно тайм-ауту, только код запускается периодически через указанные промежутки времени до отмены интервала или до выгрузки страницы. Задать интервал можно с помощью метода `setInterval()`, который принимает те же аргументы, что и `setTimeout()`, то есть выполняемый код в виде строки или функции и период его запуска в миллисекундах. Рассмотрим пример:

Листинг IntervalExample01.htm


```
//не делайте так!
setInterval("alert('Hello world!') ", 10000);

// предпочтительный способ
setInterval(function() {
    alert("Hello world!");
}, 10000);
```

Метод `setInterval()` возвращает идентификатор интервала, который можно использовать для отмены интервала с помощью метода `clearInterval()`. Для интервалов это важнее, чем для тайм-аутов, потому что интервал, оставленный без присмотра, будет запускать код вплоть до выгрузки страницы. Вот пример типичного применения интервала:

Листинг IntervalExample02.htm

```
var num = 0;
var max = 10;
var intervalId = null;

function incrementNumber() {
    num++;

    // при достижении значения max интервал отменяется
    if (num == max) {
        clearInterval(intervalId);
        alert("Done");
    }
}

intervalId = setInterval(incrementNumber, 500);
```

Переменная `num` увеличивается здесь каждые полсекунды, пока не достигает максимального значения, после чего интервал отменяется. Этот прием также можно реализовать с помощью тайм-аутов:

Листинг TimeoutExample03.htm


```
var num = 0;
var max = 10;

function incrementNumber() {
    num++;

    //если значение max не достигнуто, задается новый тайм-аут
    if (num < max) {
        setTimeout(incrementNumber, 500);
    } else {
        alert("Done");
    }
}

setTimeout(incrementNumber, 500);
```

Заметьте, что при использовании тайм-аутов не требуется отслеживать идентификатор тайм-аута, потому что выполнение кода прекращается само по себе и продолжается только в случае задания нового тайм-аута. Этот прием считается наилучшим способом задания интервалов, хотя сами интервалы при этом не используются. Настоящие интервалы применяются в окончательном коде редко, потому что браузеры не гарантируют точность времени между окончанием одного интервала и началом другого и могут пропускать интервалы. Использование тайм-аутов, как

в последнем примере, гарантирует, что этого не случится. Как правило, интервалов лучше избегать.

Системные диалоговые окна

С помощью методов `alert()`, `confirm()` и `prompt()` можно отображать в браузерах системные диалоговые окна. Эти окна не связаны с веб-страницей, отображаемой в браузере, и не содержат HTML-код, а их вид определяется параметрами операционной системы и (или) браузера, но не стилями CSS. Каждое из этих диалоговых окон является синхронным и модальным, то есть выполнение кода приостанавливается на время показа диалогового окна и возобновляется после его закрытия.

Метод `alert()` вы уже много раз видели в книге. Он просто принимает строку, которую нужно показать пользователю. При вызове `alert()` появляется системное окно сообщения с указанным оповещением и кнопкой OK. Например, вызов `alert("Hello world!")` в Internet Explorer операционной системы Windows XP выводит на экран диалоговое окно, показанное на рис. 8.3.

Диалоговые окна оповещений обычно применяют, если нужно уведомить пользователя о чем-то, что он не контролирует, например об ошибке. Единственное, что может сделать пользователь, это закрыть диалоговое окно после прочтения оповещения.

С помощью метода `confirm()` можно вывести на экран диалоговое окно запроса подтверждения. Оно также отображает сообщение для пользователя, но помимо кнопки OK содержит кнопку Cancel (Отмена), благодаря чему пользователь может подтвердить или отменить некоторое действие. Например, вызов `confirm("Are you sure?")` выводит на экран диалоговое окно запроса подтверждения, показанное на рис. 8.4.

Рис. 8.3

Рис. 8.4

Чтобы можно было определить, какую кнопку выбрал пользователь, метод `confirm()` возвращает `true`, если был щелчок на кнопке OK, и `false`, если он щелкнул на кнопке Cancel (Отмена) или закрыл диалоговое окно щелчком на системной кнопке закрытия окна в его углу (со значком ×). Типичный код вызова этого диалогового окна выглядит так:

```
if (confirm("Are you sure?")) {
    alert("I'm so glad you're sure! ");
} else {
    alert("I'm sorry to hear you're not sure. ");
}
```

В этом примере диалоговое окно запроса подтверждения отображается при выполнении условия инструкции `if`. Если пользователь щелкает на кнопке OK, появляется оповещение `"I'm so glad you're sure!"` (Я рад, что вы уверены), если же он выбирает кнопку Cancel (Отмена), выводится оповещение `"I'm sorry to hear you're not sure"` (Жаль, что вы не уверены). Этот шаблонный код часто используется, когда пользователь пытается что-либо удалить, например сообщение электронной почты.

Наконец, метод `prompt()` выводит на экран диалоговое окно, которое запрашивает у пользователя информацию. Вместе с кнопками OK и Cancel (Отмена) оно содержит текстовое поле для ввода данных. Метод `prompt()` принимает два аргумента: текст, который нужно показать пользователю, и значение, указанное в текстовом поле по умолчанию (которое может быть пустой строкой). Так, вызов `prompt("What's your name?", "Michael")` выводит на экран диалоговое окно, показанное на рис. 8.5.

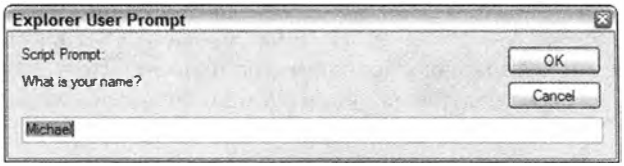

Рис. 8.5

Если пользователь щелкает на кнопке OK, метод `prompt()` возвращает значение, введенное в текстовом поле, а если выбирается кнопка Cancel (Отмена) или окно закрывается иным образом (без щелчка на кнопке OK), метод возвращает `null`. Вот пример:

```
var result = prompt("What's your name? ", "");
if (result !== null) {
    alert("Welcome, " + result);
}
```

Такие системные диалоговые окна полезны, если нужно вывести для пользователя некоторую информацию и запросить подтверждение решения. Не требуя загрузки кода HTML, CSS или JavaScript, они позволяют быстро и просто улучшить веб-приложение.

Браузеры Chrome и Opera с появлением системных диалоговых окнон стали предлагать особую функциональность. Если выполняемый сценарий выводит на экран хотя бы два системных диалоговых окна, в каждое окно после первого добавляется флажок, позволяющий отключить вывод последующих диалоговых окон до перезагрузки страницы (рис. 8.6).

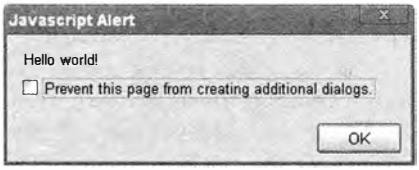

Рис. 8.6

Если установить флажок и закрыть диалоговое окно, последующие системные диалоговые окна трех описанных типов будут блокироваться до перезагрузки страницы. Chrome не сообщает разработчику о том, появилось ли диалоговое окно. При простое браузера счетчик диалоговых окон сбрасывается, так что если два отдельных действия пользователя привели к выводу оповещения, флажок не появляется ни в одном из них; если одно действие пользователя привело к показу двух оповещений подряд, второе окно будет содержать флажок. Со временем аналогичная функциональность была также добавлена в Internet Explorer 9 и Firefox 4.

В JavaScript доступны также диалоговые окна поиска и печати, которые выводятся асинхронно, тут же возвращая управление сценарию. Это те же окна, которые появляются, когда пользователь выбирает в меню браузера команду поиска или печати. Вывести на экран их можно с помощью методов `find()` и `print()` объекта `window`:

```
// отображение диалогового окна печати
window.print();

// отображение диалогового окна поиска
window.find();
```

Эти методы не показывают, что пользователь сделал с диалоговым окном, так что трудно найти для них полезное применение. Поскольку окна асинхронны, они не влияют на счетчик диалоговых окон в Chrome и на них не распространяется запрет на вывод последующих диалоговых окон.

Объект location

Объект `location` считается одним из наиболее полезных в BOM. Он предоставляет сведения о текущем загруженном документе и обеспечивает общий функционал навигации. Он уникален тем, что является свойством и `window`, и `document`, то есть свойства `window.location` и `document.location` указывают на один и тот же объект. С помощью объекта `location` можно не только получить сведения о текущем загруженном документе, но и выполнить синтаксический анализ URL-адреса, разобрав его на отдельные сегменты, доступные в качестве свойств. Эти свойства приведены в следующей таблице (префикс `location` не указан):

Имя свойства	Пример	Описание
hash	"#contents"	Хэш URL-адреса (знак решетки и любое количество других знаков) или пустая строка, если у URL-адреса нет хэша
host	"www.wrox.com:80"	Имя сервера и номер порта при его наличии
hostname	"www.wrox.com"	Имя сервера без номера порта
href	"http://www.wrox.com"	Полный URL-адрес текущей загруженной страницы. Метод toString() объекта location возвращает это значение
pathname	"/WileyCDA/"	Каталог и (или) имя файла в URL-адресе
port	"8080"	Порт запроса, если он указан в URL-адресе. Если URL-адрес не содержит порт, это свойство возвращает пустую строку
protocol	"http:"	Протокол доступа к странице (обычно "http:" или "https:")
search	"?q=javascript"	Строка запроса в URL-адресе. Это свойство возвращает строку, которая начинается с вопросительного знака

Аргументы строки запроса

Большинство элементов объекта `location` можно легко получить с помощью этих свойств, но работать с необработанной строкой запроса неудобно. Хотя свойство `location.search` возвращает все, начиная от вопросительного знака и до конца URL-адреса, аргументы строки запроса по отдельности недоступны. Следующая функция разбирает строку запроса и возвращает объект, содержащий отдельные аргументы:

Листинг LocationExample01.htm


```
function getQueryStringArgs(){

    // получение строки запроса без начального вопросительного знака
    var qs = (location.search.length > 0 ?
              location.search.substring(1) : ""),

        // объект для хранения аргументов
        args = {},

        // получение отдельных элементов
        items = qs.length ? qs.split("&") : [],
        item = null,
        name = null,
        value = null,

        // используется в цикле for
```

```
        i = 0,
        len = items.length;

    // запись каждого элемента в объект args
    for (i=0; i < len; i++){
        item = items[i].split("=");
        name = decodeURIComponent(item[0]);
        value = decodeURIComponent(item[1]);

        if (name.length) {
            args[name] = value;
        }
    }

    return args;
}
```

Первым делом эта функция удаляет начальный вопросительный знак из строки запроса, если свойство `location.search` содержит один или более знаков. Аргументы сохраняются в объекте `args`, который создается с помощью литерала объекта. Затем функция делит строку запроса по знаку амперсанда и сохраняет результат в массиве строк формата `имя=значение`. Цикл `for` делит каждый элемент этого массива по знаку равенства, записывая в новый массив имя аргумента и его значение как первый и второй элемент соответственно. Предполагается, что строка запроса закодирована, поэтому далее эти элементы декодируются с помощью метода `decodeURIComponent()` и присваиваются переменным `name` и `value`. Наконец, `name` добавляется к объекту `args` как свойство со значением `value`. Эту функцию можно использовать следующим образом:

```
// предполагается, что указана строка запроса ?q=javascript&num=10

var args = getQueryStringArgs();

alert(args["q"]);          // "javascript"
alert(args["num"]);        // "10"
```

Аргументы строки запроса теперь представлены свойствами возвращенного объекта, что обеспечивает быстрый доступ к каждому аргументу.

Работа с объектом location

Открыть в браузере другую страницу с помощью объекта `location` можно несколькими способами. Первый, наиболее популярный, — это вызвать метод `assign()` с URL-адресом в качестве аргумента, например:

```
location.assign("http://www.wrox.com");
```

Этот метод немедленно инициирует переход по новому URL-адресу и создает запись в стеке журнала браузера. При назначении URL-адреса свойству `location.href` или

window.location также вызывается метод assign() с указанным значением. Например, обе следующие инструкции эквивалентны явному вызову метода assign():

```
window.location = "http://www.wrox.com";
location.href = "http://www.wrox.com";
```

Из этих трех способов в коде чаще всего встречается последний.

Для изменения текущей загруженной страницы можно использовать свойства hash, search, hostname, pathname и port объекта location, например:

```
// предполагается начальный адрес http://www.wrox.com/WileyCDA/

// изменение URL-адреса на "http://www.wrox.com/WileyCDA/#section1"
location.hash = "#section1";

// изменение URL-адреса на "http://www.wrox.com/WileyCDA/?q=javascript"
location.search = "?q=javascript";

// изменение URL-адреса на "http://www.yahoo.com/WileyCDA/"
location.hostname = "www.yahoo.com";

// изменение URL-адреса на "http://www.yahoo.com/mydir/"
location.pathname = "mydir";

// изменение URL-адреса на "http://www.yahoo.com:8080/WileyCDA/
location.port = "8080"
```

Каждый раз, когда изменяется свойство объекта location (исключая hash), загружается страница с новым URL-адресом.

Изменение значения hash приводит к сохранению новой записи в журнале браузера в Internet Explorer 8+, Firefox, Safari 2+, Opera 9+ и Chrome. В более ранних версиях Internet Explorer свойство hash обновлялось только при щелчке на ссылке с хэшированным URL-адресом, но не на кнопке Back (Назад) или Forward (Вперед).

При изменении URL-адреса одним из описанных способов в стеке журнала браузера сохраняется запись, чтобы пользователь мог вернуться к предыдущей странице, щелкнув в браузере на кнопке Back (Назад). Такое поведение можно запретить с помощью метода replace(), который выполняет переход по переданному ему URL-адресу, но не сохраняет запись в стеке журнала. После вызова replace() пользователь не может вернуться к предыдущей странице. Рассмотрим пример:

Листинг LocationReplaceExample01.htm


```
<!DOCTYPE html>
<html>
<head>
    <title>You won't be able to get back here</title>
</head>
    <body>
```

```
    <p>Enjoy this page for a second, because you won't be coming back here.</p>
    <script type="text/javascript">
        setTimeout(function () {
            location.replace("http://www.wrox.com/");
        }, 1000);
    </script>
</body>
</html>
```

Если загрузить эту страницу в веб-браузере, через секунду будет выполнен переход на сайте www.wrox.com. При этом кнопка Back (Назад) окажется недоступной, и вы не сможете вернуться на страницу примера без повторного ввода полного URL-адреса.

Метод `location.reload()` перезагружает текущую страницу. Если вызвать его без аргументов, страница перезагружается наиболее эффективным образом (из кэша браузера, если она не была изменена с момента последнего запроса). Чтобы перезагрузить страницу с сервера, передайте в метод значение `true`:

```
location.reload();        // перезагрузка - возможно, из кэша
location.reload(true);    // перезагрузка с сервера
```

При значительных задержках в сети или недостатке системных ресурсов код после вызова `reload()` может быть не выполнен, так что этот метод лучше вызывать в последней строке кода.

Объект navigator

Объект `navigator`, представленный в Netscape Navigator 2, обеспечивает стандартный способ идентификации браузера в клиентской системе. Некоторые браузеры поддерживают и другие способы получения той же или похожей информации (такие, как свойство `window.clientInformation` в Internet Explorer или `window.opera` в Opera), но объект `navigator` хорош тем, что он доступен во всех веб-браузерах с поддержкой JavaScript. Как и у других BOM-объектов, набор его свойств зависит от браузера. В следующей таблице приведены все его свойства и методы, а также указано, в каких версиях браузеров они поддерживаются.

Свойство/метод	Описание	IE	Firefox	Safari/Chrome	Opera
appCodeName	Имя браузера. Обычно "Mozilla" — даже для браузеров не от Mozilla	3.0+	1.0+	1.0+	7.0+
appMinorVersion	Дополнительные сведения о версии	4.0+	—	—	9.5+
appName	Полное имя браузера	3.0+	1.0+	1.0+	7.0+

Свойство/ метод	Описание	IE	Firefox	Safari/ Chrome	Opera
appVersion	Версия браузера. Обычно не соответствует фактической версии браузера	3.0+	1.0+	1.0+	7.0+
buildID	Номер сборки браузера	—	2.0+	—	—
cookieEnabled	Указывает, включена ли поддержка cookie-файлов	4.0+	1.0+	1.0+	7.0+
cpuClass	Тип процессора клиентского компьютера ("x86", "68K", "Alpha", "PPC" или "Other")	4.0+	—	—	—
javaEnabled()	Этот метод указывает, включена ли в браузере поддержка Java	4.0+	1.0+	1.0+	7.0+
language	Основной язык браузера	—	1.0+	1.0+	7.0+
mimeTypes	Массив MIME-типов, зарегистрированных в браузере	4.0+	1.0+	1.0+	7.0+
onLine	Указывает, подключен ли браузер к Интернету	4.0+	1.0+	—	9.5+
opsProfile	По-видимому, не используется. Документация недоступна	4.0+	—	—	—
oscpu	Операционная система и (или) процессор устройства, на котором работает браузер	—	1.0+	—	—
platform	Системная платформа, на которой работает браузер	4.0+	1.0+	1.0+	7.0+
plugins	Массив подключаемых модулей, установленных в браузере. В Internet Explorer это массив всех элементов <embed> на странице	4.0+	1.0+	1.0+	7.0+
preference()	Этот метод задает предпочтения пользователя. Доступен только в привилегированном режиме	—	1.5+	—	—
product	Имя продукта (обычно "Gecko")	—	1.0+	1.0+	—
productSub	Дополнительные сведения о продукте (обычно сведения о версии Gecko)	—	1.0+	1.0+	—
registerContent-Handler()	Этот метод регистрирует вебсайт как обработчик для конкретного MIME-типа	—	2.0+	—	—

Продолжение

Свойство/метод	Описание	IE	Firefox	Safari/Chrome	Opera
registerProtocol-Handler()	Этот метод регистрирует веб-сайт как обработчик для конкретного протокола	—	2.0+	—	—
securityPolicy	(Устаревшее свойство). Имя политики безопасности. Оставлено ради обратной совместимости с Netscape Navigator 4	—	1.0+	—	—
systemLanguage	Язык операционной системы	4.0+	—	—	—
taintEnabled()	(Устаревший метод). Указывает, включен ли режим разрушения переменных. Оставлен ради обратной совместимости с Netscape Navigator 3	4.0+	1.0+	—	7.0+
userAgent	Строка пользовательского агента для браузера	3.0+	1.0+	1.0+	7.0+
userLanguage	Язык операционной системы, предлагаемый по умолчанию	4.0+	—	—	7.0+
userProfile	Объект для доступа к сведениям профиля пользователя	4.0+	—	—	—
vendor	Производитель браузера	—	1.0+	1.0+	—
vendorSub	Дополнительные сведения о производителе	—	1.0+	1.0+	—

Свойства объекта `navigator` обычно используются для определения типа браузера, который обрабатывает веб-страницу (см. главу 9).

Обнаружение подключаемых модулей

Нередко требуется выяснить, установлен ли в браузере конкретный подключаемый модуль. В браузерах, отличных от Internet Explorer, это можно сделать с помощью массива `plugins`. Каждый его элемент имеет следующие свойства:

- `name` — имя подключаемого модуля;
- `description` — описание подключаемого модуля;
- `filename` — имя файла подключаемого модуля;
- `length` — количество MIME-типов, обрабатываемых подключаемым модулем.

Обычно свойства `name` достаточно для обнаружения подключаемого модуля, но гарантировать это нельзя. Чтобы идентифицировать подключаемый модуль,

доступные модули перебираются в цикле, а их имена сравниваются с указанным именем, как в этом примере:

Листинг PluginDetectionExample01.htm


```
// обнаружение подключаемого модуля - не работает в Internet Explorer
function hasPlugin(name){
    name = name.toLowerCase();
    for (var i=0; i < navigator.plugins.length; i++){
        if (navigator.plugins[i].name.toLowerCase().indexOf(name) > -1){
            return true;
        }
    }

    return false;
}

// обнаружение подключаемого модуля Flash
alert(hasPlugin("Flash"));

// обнаружение подключаемого модуля QuickTime
alert(hasPlugin("QuickTime"));
```

Метод `hasPlugin()` принимает в качестве аргумента имя искомого подключаемого модуля, которое тут же преобразуется в нижний регистр, чтобы упростить сравнение. Затем свойство `name` каждого элемента массива `plugins` проверяется с помощью метода `indexOf()` на предмет того, содержит ли оно переданное имя. Во избежание ошибок сравнение выполняется в нижнем регистре. Чтобы не было путаницы, аргумент метода должен быть как можно более специфичным. Строки `"Flash"` и `"QuickTime"` достаточно уникальны, чтобы проблем не возникло. Этот метод обнаруживает подключаемые модули в Firefox, Safari, Opera и Chrome.

Каждый объект plugin является также массивом объектов MimeType, доступных с помощью скобочной нотации. Каждый объект MimeType имеет четыре свойства: description — описание MIME-типа; enabledPlugin — указатель на объект plugin; suffixes — строка разделенных запятыми расширений файлов для MIME-типа; type — полная строка MIME-типа.

В браузере Internet Explorer идентифицировать подключаемые модули сложнее, потому что в нем они работают иначе. Единственный способ решить эту задачу в Internet Explorer — попытаться создать экземпляр конкретного подключаемого модуля с помощью фирменного типа `ActiveXObject`. Подключаемые модули реализованы в Internet Explorer как COM-объекты, для идентификации которых используются уникальные строки. Таким образом, чтобы проверить конкретный подключаемый модуль, нужно знать его COM-идентификатор. Например, Flash имеет идентификатор `"ShockwaveFlash.ShockwaveFlash"`. Располагая этой информацией, для обнаружения подключаемого модуля в Internet Explorer можно использовать следующую функцию:

Листинг PluginDetectionExample02.htm


```
// обнаружение подключаемого модуля в Internet Explorer
function hasIEPlugin(name){
    try {
        new ActiveXObject(name);
        return true;
    } catch (ex){
        return false;
    }
}

// обнаружение подключаемого модуля Flash
alert(hasIEPlugin("ShockwaveFlash.ShockwaveFlash"));

// обнаружение подключаемого модуля QuickTime
alert(hasIEPlugin("QuickTime.QuickTime"));
```

Функция `hasIEPlugin()` принимает в качестве единственного аргумента COM-идентификатор и пытается создать экземпляр `ActiveXObject`. Этот код заключен в блок `try-catch`, потому что попытка создать неизвестный COM-объект приводит к ошибке. Если создать объект удается, функция возвращает `true`, в противном случае выполняется блок `catch` и возвращается значение `false`. Две последние инструкции в коде проверяют, доступны ли в Internet Explorer подключаемые модули Flash и QuickTime.

Поскольку эти два универсальных подхода так сильно различаются, обычно на их основе создают функции, проверяющие наличие конкретных, а не любых подключаемых модулей, например:

Листинг PluginDetectionExample03.htm

```
// обнаружение подключаемого модуля Flash в любых браузерах
function hasFlash(){
    var result = hasPlugin("Flash");
    if (!result){
        result = hasIEPlugin("ShockwaveFlash.ShockwaveFlash");
    }
    return result;
}

// обнаружение подключаемого модуля QuickTime в любых браузерах
function hasQuickTime(){
    var result = hasPlugin("QuickTime");
    if (!result){
        result = hasIEPlugin("QuickTime.QuickTime");
    }
    return result;
}

// обнаружение подключаемого модуля Flash
alert(hasFlash());

// обнаружение подключаемого модуля QuickTime
alert(hasQuickTime());
```

Функции `hasFlash()` и `hasQuickTime()` выполняют сначала метод обнаружения подключаемого модуля в браузерах, отличных от Internet Explorer. Если этот метод возвращает `false`, вызывается метод обнаружения подключаемого модуля в Internet Explorer. Если он также возвращает `false`, то и результат всего метода равен `false`. Если какой-либо из методов обнаружения подключаемого модуля возвращает `true`, результат всего метода равен `true`.

Метод refresh() коллекции plugins обновляет ее согласно сведениям о новых установленных подключаемых модулях. Он принимает один аргумент: логическое значение, указывающее, нужно ли перезагрузить страницу. Если аргумент равен true, все страницы с подключаемыми модулями перезагружаются, иначе коллекция plugins обновляется без перезарузки страницы.

Регистрация обработчиков

В Firefox 2 к объекту `navigator` были добавлены методы `registerContentHandler()` и `registerProtocolHandler()`, которые теперь формально определены в HTML 5. С их помощью можно указать, что веб-сайт способен обрабатывать данные конкретных типов — это позволяет направлять их по умолчанию в соответствующие программы, такие как онлайновые средства чтения RSS-каналов и почтовые веб-приложения.

Метод `registerContentHandler()` принимает три аргумента: MIME-тип, URL-адрес страницы, которая может обрабатывать этот MIME-тип, и имя приложения. Например, следующий код регистрирует сайт как обработчик RSS-каналов:

```
navigator.registerContentHandler("application/rss+xml",
    "http://www.somereader.com?feed=%s", "Some Reader");
```

В качестве MIME-типа здесь указаны RSS-каналы. Вторым аргументом является URL-адрес страницы, которой браузер будет передавать URL-адреса RSS-каналов, автоматически подставляя их вместо `%s`. При следующем запросе RSS-канала браузер перейдет по указанному URL-адресу, и веб-приложение сможет обработать запрос надлежащим образом.

В Firefox до версии 4 включительно метод registerContentHandler() поддерживает только три MIME-типа: «application/rss+xml», «application/atom+xml» и «application/vnd.mozilla.maybe.feed». Все они делают одно и то же: регистрируют обработчик для всех RSS- и Atom-каналов.

Похожее можно сделать для протоколов с помощью метода `registerProtocolHandler()`, который также принимает три аргумента: протокол (например, `"mailto"` или `"ftp"`), URL-адрес страницы с обработчиком протокола и имя приложения. Например, следующий код регистрирует веб-приложение как почтовый клиент, предлагаемый по умолчанию:

```
navigator.registerProtocolHandler("mailto",
    "http://www.somemailclient.com?cmd=%s", "Some Mail Client");
```

После выполнения этого кода протокол `mailto` будет обрабатываться почтовым веб-клиентом. Как и в предыдущем примере, вторым аргументом здесь является URL-адрес страницы обработчика, а `%s` представляет оригинальный запрос.

В Firefox 2 метод registerProtocolHandler() реализован, но не работает. В Firefox 3 он работает нормально.

Объект screen

Объект `screen` (который также является свойством `window`) — один из немногих JavaScript-объектов, которые практически не используются в коде. Он просто предоставляет сведения о графических параметрах клиентской системы вне окна браузера, таких как ширина и высота в пикселях. Доступность тех или иных свойств объекта `screen` зависит от браузера. Эти свойства и браузеры, в которых они поддерживаются, указаны в следующей таблице.

Свойство	Описание	IE	Firefox	Safari/ Chrome	Opera
availHeight	Высота экрана в пикселях за вычетом системных элементов, таких как панель задач Windows (только для чтения)	X	X	X	X
availLeft	Первый пиксель слева, не занятый системными элементами (только для чтения)		X	X	
availTop	Первый пиксель сверху, не занятый системными элементами (только для чтения)		X	X	
availWidth	Ширина экрана в пикселях за вычетом системных элементов (только для чтения)	X	X	X	X
bufferDepth	Количество битов, используемых для построения растровых изображений без визуализации (чтение и запись)	X			
colorDepth	Количество битов, используемых для представления цветов; 32 в большинстве систем (только для чтения)	X	X	X	X
deviceXDPI	Фактическое количество точек на дюйм экрана по горизонтали (только для чтения)	X			
deviceYDPI	Фактическое количество точек на дюйм экрана по вертикали (только для чтения)	X			

Свойство	Описание	IE	Firefox	Safari/ Chrome	Opera
fontSmoothing-Enabled	Указывает, включено ли сглаживание шрифтов (только для чтения)	X			
height	Высота экрана в пикселях	X	X	X	X
left	Смещение левой стороны текущего экрана в пикселях		X		
logicalXDPI	Логическое количество точек на дюйм экрана по горизонтали (только для чтения)	X			
logicalYDPI	Логическое количество точек на дюйм экрана по вертикали (только для чтения)	X			
pixelDepth	Глубина цвета в битах (только для чтения)		X	X	X
top	Смещение верхней стороны экрана в пикселях		X		
updateInterval	Интервал обновления экрана в миллисекундах (чтение и запись)	X			
width	Ширина экрана в пикселях	X	X	X	X

Эти сведения часто собираются средствами контроля сайтов для оценки возможностей клиентов, но обычно они не используются для изменения функционала. Иногда с их помощью развертывают окно браузера на весь экран:

```
window.resizeTo(screen.availWidth, screen.availHeight);
```

Как уже отмечалось, многие браузеры отключают возможность изменения размеров окна браузера, так что этот код работает не всегда.

На мобильных устройствах размеры экрана определяются немного иначе. Устройства с системой iOS всегда возвращают размеры экрана в книжной ориентации (1024 × 768). Что касается устройств Android, они настраивают значения `screen.width` и `screen.height` правильно.

Объект history

Объект `history` представляет журнал навигации за все время работы с конкретным окном. Так как это свойство объекта `window`, у каждого окна, вкладки и фрейма браузера есть собственный объект `history`. Из соображений безопасности браузеры не позволяют определять URL-адреса страниц, которые посещал пользователь, но можно перемещаться по их списку вперед и назад, не зная URL-адреса.

Метод go() позволяет перемещаться по журналу пользователя в обоих направлениях и принимает один аргумент: количество страниц, на которое нужно перейти назад или вперед. Если значение отрицательное, выполняется переход назад, как при щелчке на кнопке Back (Назад) в браузере, а если положительное — вперед, как при щелчке на кнопке Forward (Вперед), например:

```
// переход к предыдущей странице
history.go(-1);

// переход к следующей странице
history.go(1);

// переход на две страницы вперед
history.go(2);
```

Аргумент метода go() может также быть строкой — в этом случае браузер переходит назад или вперед к ближайшей странице, адрес которой содержит эту строку. Если в журнале нет записи, соответствующей строке, метод ничего не делает, например:

```
// переход к ближайшей странице wrox.com
history.go("wrox.com");

// переход к ближайшей странице nczonline.net
history.go("nczonline.net");
```

Вместо go() можно использовать методы back() и forward(), которые имитируют щелчки на кнопках Back (Назад) и Forward (Вперед) в браузере:

```
// возврат на одну страницу
history.back();

// переход к следующей странице
history.forward();
```

У объекта history есть также свойство length, которое указывает общее количество элементов в стеке журнала. Первой странице, загруженной в окно, на вкладку или во фрейм, соответствует нулевое значение history.length. Проверив это свойство, можно определить, имеем ли мы дело с начальной страницей сеанса:

```
if (history.length == 0){
    // это первая страница в окне пользователя
}
```

Объект history применяется не очень часто. Обычно с его помощью создают пользовательские кнопки Back (Назад) и Forward (Вперед), а также определяют, является ли страница первой в журнале пользователя. В HTML5 функционал объекта history расширен (см. главу 16).

Записи в стеке журнала создаются при изменении URL-адреса страницы, а в Internet Explorer 8+, Opera, Firefox, Safari 3+ и Chrome еще и при изменении хэша URL-адреса (то есть установка свойства location.hash приводит к вставке новой записи в стек журнала).

Резюме

Объектная модель браузера (BOM) основана на объекте `window`, который представляет окно браузера и видимую область страницы. В ECMAScript он дублируется как объект `Global`, так что все глобальные переменные и функции становятся его свойствами, а все встроенные конструкторы и функции изначально относятся к нему. В этой главе мы обсудили ряд BOM-элементов, перечисленных далее.

- Если используются фреймы, у каждого из них есть свой объект `window` и свои копии всех встроенных конструкторов и функций. Каждый фрейм хранится в коллекции `frames`, которая индексируется по позиции и имени.
- Ссылаться на другие фреймы можно с помощью нескольких указателей на объекты `window`.
- Объект `top` всегда представляет фрейм, самый удаленный от центра, то есть все окно браузера.
- Объект `parent` соответствует фрейму-контейнеру, а `self` — текущему объекту `window`.
- Объект `location` обеспечивает программный доступ к системе навигации браузера. Задавая его свойства, можно изменять URL-адреса по частям или полностью.
- Метод `replace()` позволяет перейти по новому URL-адресу и заменить текущую страницу в журнале браузера.
- Объект `navigator` предоставляет сведения о браузере. Доступные сведения во многом зависят от используемого браузера, хотя некоторые свойства, такие как `userAgent`, поддерживаются во всех браузерах.

Два других объекта, доступных в BOM, имеют очень ограниченное применение. Объект `screen` возвращает сведения о дисплее клиентской системы, которые иногда собирают на веб-сайтах как метрики. Объект `history` предоставляет некоторые возможности для работы со стеком журнала браузера. С его помощью можно определить количество сайтов в стеке журнала и перейти назад или вперед к любой странице в журнале.

9 Распознавание клиента

- Распознавание возможностей
- История распознавания пользовательского агента
- Выбор способа распознавания

Производители браузеров прилагают немалые усилия для согласования способов взаимодействия с ними, но все же каждый браузер имеет уникальные особенности. Даже выпуски одного браузера для разных платформ часто имеют важные различия, хотя технически это одна версия. Из-за этих различий веб-разработчики вынуждены либо использовать возможности, общие для всех целевых браузеров, либо распознавать браузер и обходить его ограничения. Обычно применяется второй подход.

Распознавание клиента остается одним из самых спорных аспектов веб-разработки, при этом большинство сходится во мнении, что браузеры должны поддерживать некую общую для всех функциональность. Возможно, в идеальном мире так бы и было, но в реальности различий и особенностей браузеров достаточно для того, чтобы рассматривать распознавание клиента как неотъемлемый элемент стратегии разработки.

Есть несколько подходов к определению используемого веб-клиента, каждый со своими достоинствами и недостатками, но распознавание клиента должно быть самым последним вариантом решения проблемы. Если возможен более общий подход, используйте его. Разработайте сначала наиболее универсальное решение, а затем расширьте его возможностями, специфичными для конкретных браузеров.

Распознавание возможностей

Наиболее популярным подходом к распознаванию клиента является *распознавание возможностей* (capability detection), цель которого — определить возможности используемого браузера, а не распознать сам браузер. Это предполагает, что идентифицировать конкретный браузер не требуется — достаточно выяснить, доступна ли нужная функциональность. Базовая схема распознавания возможностей такова:

```
if (object.propertyInQuestion){
    // использование object.propertyInQuestion
}
```

Например, DOM-метод `document.getElementById()` в Internet Explorer до версии 5 недоступен, но для решения той же задачи можно использовать нестандартное свойство `document.all`. Распознать эти возможности можно так:

```
function getElement(id){
    if (document.getElementById){
        return document.getElementById(id);
    } else if (document.all){
        return document.all[id];
    } else {
        throw new Error("No way to retrieve element!");
    }
}
```

Функция `getElement()` возвращает элемент с указанным идентификатором. Обычно для этого используется функция `document.getElementById()`, которая и проверяется в первую очередь. Если эта функция существует, она вызывается, в противном случае проверяется наличие свойства `document.all`. Если ни один из этих способов не доступен (что крайне маловероятно), генерируется ошибка.

При распознавании возможностей нужно помнить о двух важных принципах. Как уже отмечалось, типичный способ получения результата следует проверять первым. Так, в предыдущем примере метод `document.getElementById()` проверяется перед свойством `document.all`. Это оптимизирует выполнение кода, предотвращая проверку лишних условий в типичных ситуациях.

Второй важный принцип заключается в том, что проверять нужно в точности ту возможность, которая вам требуется. То, что доступна одна возможность, не означает, что доступна другая, например:

```
function getWindowWidth(){
    if (document.all){        // предполагается, что используется IE
        return document.documentElement.clientWidth;    // НЕПРАВИЛЬНО!!!
    } else {
        return window.innerWidth;
    }
}
```

В этом примере распознавание возможностей применяется неправильно. Функция `getWindowWidth()` сначала проверяет, доступно ли свойство `document.all`, и если да — возвращает значение `document.documentElement.clientWidth`, иначе возвращается значение `window.innerWidth`, которое не поддерживается в Internet Explorer 8 и более ранних версий. Проблема в том, что наличие свойства `document.all` не всегда указывает, что браузером является Internet Explorer. Им также может быть ранняя версия Opera, в которой поддерживаются свойства `document.all` и `window.innerWidth`.

Надежное распознавание возможностей

Для надежного распознавания возможности не всегда достаточно проверить ее доступность — надо еще убедиться, что она работает надлежащим образом. В предыдущем разделе для определения доступности элементов выполняется приведение их типов, но это не гарантирует, что это действительно те элементы, которые нам нужны. Рассмотрим следующую функцию, которая определяет, поддерживает ли объект сортировку:

```
// НЕ ДЕЛАЙТЕ ТАК! Неправильное распознавание возможности -
// проверяется только существование элемента
function isSortable(object){
    return !!object.sort;
}
```

Чтобы определить, можно ли отсортировать объект, эта функция проверяет, есть ли у него метод `sort()`. Проблема в том, что любой объект со свойством `sort` тоже возвращает `true`:

```
var result = isSortable({ sort: true });
```

Поскольку наличие свойства не гарантирует, что объект поддерживает сортировку, лучше проверить, является ли элемент `sort` функцией:

```
// Лучше - код проверяет, является ли sort функцией
function isSortable(object){
    return typeof object.sort == "function";
}
```

В этом коде оператор `typeof` определяет, является ли элемент `sort` функцией, которой можно воспользоваться для сортировки данных в объекте.

Распознавание возможностей с помощью `typeof` — более надежный подход, но и он не гарантирует правильный результат. В частности, объекты среды не обязаны возвращать осмысленные значения при вызове `typeof`. Наиболее вопиющий пример имеет место в Internet Explorer. В большинстве браузеров, где доступен метод `document.createElement()`, следующая функция возвращает `true`:

```
// работает неправильно в Internet Explorer до версии 8 включительно
function hasCreateElement(){
    return typeof document.createElement == "function";
}
```

Однако в Internet Explorer 8 и более ранних версий эта функция возвращает `false`, потому что выражение `typeof document.createElement` интерпретируется как `"object"`, а не `"function"`. Как уже отмечалось, DOM-объекты являются объектами среды, которые в Internet Explorer 8 и более ранних версий реализованы с помощью COM, а не JScript. Это относится и к функции `document.createElement()`, поэтому для нее оператор `typeof` возвращает `"object"`. Internet Explorer 9 возвращает для DOM-методов значение `"function"`.

Можно привести и другие примеры того, как оператор `typeof` возвращает непредвиденный результат в Internet Explorer. В частности, ActiveX-объекты (поддерживаемые только в Internet Explorer) радикально отличаются от других объектов. Проверка свойства такого объекта без оператора `typeof` может вызвать ошибку:

```
// вызывает ошибку в Internet Explorer
var xhr = new ActiveXObject("Microsoft.XMLHttp");
if (xhr.open){    // здесь возникает ошибка
    // какие-то действия
}
```

Простой доступ к функции как к свойству в этом примере приводит к ошибке. Безопаснее использовать оператор `typeof`, однако Internet Explorer возвращает при вызове `typeof xhr.open` значение `"unknown"`. Это означает, что полностью проверка наличия функции у какого-либо объекта браузера должна быть такой:

```
// автор: Питер Мишо (Peter Michaux)
function isHostMethod(object, property) {
  var t = typeof object[property];
  return t=='function' ||
         (!!(t=='object' && object[property])) ||
         t=='unknown';
}
```

Эту функцию можно использовать следующим образом:

```
result = isHostMethod(xhr,"open");        // true
result = isHostMethod(xhr,"foo");         // false
```

На сегодня функция `isHostMethod()` — наиболее надежное средство распознавания возможностей с учетом особенностей браузеров. Имейте в виду, что реализация объектов среды может измениться, поэтому нет никаких гарантий, что эта функция продолжит правильно работать в будущем. Как разработчик, вы сами должны оценивать риск с учетом задачи, которую нужно решить.

Подробное обсуждение распознавания возможностей в JavaScript см. в статье Питера Мишо «Feature Detection: State of the Art Browser Scripting» (Распознавание возможностей: современные сценарии для браузеров) по адресу http://peter.michaux.ca/articles/feature-detection-state-of-the-art-browser-scripting.

Распознавание возможностей — не распознавание браузера

Распознавание конкретной возможности или множества возможностей не всегда позволяет идентифицировать используемый браузер. Следующий код «распознавания браузера», аналоги которого встречаются на многих веб-сайтах, может служить примером неправильного распознавания возможности:

```
// НЕ ДЕЛАЙТЕ ТАК! Код недостаточно конкретен.
var isFirefox = !!(navigator.vendor && navigator.vendorSub);

// НЕ ДЕЛАЙТЕ ТАК! Слишком много предположений.
var isIE = !!(document.all && document.uniqueID);
```

Это классические примеры неправильного распознавания возможностей. В прошлом браузер Firefox можно было идентифицировать по наличию свойств `navigator.vendor` и `navigator.vendorSub`, но затем такие же свойства были реализованы в Safari, так что теперь первый пример может выдать ошибочный результат. Во втором примере по свойствам `document.all` и `document.uniqueID` распознается Internet Explorer, при этом делается необоснованное предположение, что оба свойства продолжат существовать в будущих версиях IE и не будут реализованы в других браузерах. Двойной оператор отрицания применяется в обоих примерах для получения логического значения, которое можно эффективнее хранить и использовать.

Имеет смысл группировать возможности согласно классам браузеров. Если известно, что в приложении нужно использовать конкретный функционал браузера, может быть полезно сразу распознать все требуемые возможности, чтобы не делать это раз за разом, например:

Листинг CapabilitiesDetectionExample01.htm


```
// использует ли браузер подключаемые модули в стиле Netscape?
var hasNSPlugins = !!(navigator.plugins && navigator.plugins.length);

// поддерживает ли браузер базовые возможности DOM Level 1?
var hasDOM1 = !!(document.getElementById && document.createElement &&
                 document.getElementsByTagName);
```

Этот код определяет, поддерживает ли браузер подключаемые модули в стиле Netscape и базовые возможности модели DOM Level 1. Полученные логические значения можно затем использовать по мере надобности, и это будет эффективнее, чем повторное распознавание возможностей.

Распознавание возможностей следует рассматривать как один из этапов решения проблемы, а не как определение конкретного браузера.

Распознавание особенностей

Распознавание особенностей (quirks detection) также используется для идентификации конкретного поведения браузера, но в этот раз нас интересуют не поддерживаемые возможности, а компоненты, которые работают неправильно (в этом контексте «особенность» можно понимать как «дефект»). При распознавании особенностей часто имеет место выполнение небольшого фрагмента кода, позволяющего выявить нестандартное поведение браузера. Например, в Internet Explorer 8 и более ранних версий циклы `for-in` пропускают свойства экземпляра с именами как у свойств прототипа, у которых атрибут `[[Enumerable]]` имеет значение `false`. Эту особенность можно распознать с помощью следующего кода:

Листинг QuirksDetectionExample01.htm


```
var hasDontEnumQuirk = function(){

    var o = { toString : function(){} };
    for (var prop in o){
        if (prop == "toString"){
            return false;
        }
    }

    return true;
}();
```

Для распознавания особенности здесь используется анонимная функция, в которой создается объект с методом `toString()`. В правильных ECMAScript-реализациях значение `toString` должно быть возвращено как свойство в цикле `for-in`.

Другой особенностью, которую часто проверяют, является перебор затененных свойств в Safari до версии 3:

Листинг QuirksDetectionExample01.htm

```
var hasEnumShadowsQuirk = function(){

    var o = { toString : function(){} };
    var count = 0;
    for (var prop in o){
        if (prop == "toString"){
            count++;
        }
    }

    return (count > 1);
}();
```

Если браузер имеет эту особенность, объект с пользовательским методом `toString()` возвращает два экземпляра свойства `toString` в цикле `for-in`.

Особенности обычно специфичны для конкретных браузеров и часто рассматриваются как дефекты, подлежащие исправлению в следующих версиях. Поскольку распознавание особенностей требует выполнения кода, рекомендуется проверять только те из них, которые напрямую связаны с текущей задачей, и делать это в начале сценария, чтобы не беспокоиться об этом позднее.

Распознавание пользовательского агента

Третьим, наиболее спорным способом распознавания клиента является *распознавание пользовательского агента* (user-agent detection), при котором для определения браузера применяется его строка пользовательского агента. Она отправляется как заголовок ответа при каждом HTTP-запросе и доступна в JavaScript в виде свойства `navigator.userAgent`. На стороне сервера ее часто используют для идентификации браузера с целью выбора тех или иных действий, но на стороне клиента распознавание пользовательского агента обычно считается крайним подходом, который уместен, только если недоступно распознавание возможностей и (или) особенностей.

Одна из сомнительных сторон этого подхода — долгая история *спуфинга* (spoofing). Спуфингом называют искажения строки пользовательского агента с целью ввести в заблуждение сервер. Чтобы понять эту проблему, необходимо обсудить историю применения и изменения строки пользовательского агента.

История

В спецификации HTTP версий 1.0 и 1.1 указано, что браузер должен отправлять короткую строку пользовательского агента, содержащую имя и версию браузера. В RFC 2616 (спецификация протокола http 1.1) строка пользовательского агента описана следующим образом:

> Маркеры продуктов требуются для идентификации взаимодействующих приложений по имени и версии. В большинстве полей, где используются маркеры продуктов, можно также указывать через пробел субпродукты, формирующие значительные части приложения. Продукты указываются согласно их значимости для идентификации приложения.

Далее спецификация предписывает задавать строку пользовательского агента как список продуктов в формате «маркер/версия продукта». Однако в реальности строки пользовательских агентов никогда не были такими простыми.

Ранние браузеры

Первый веб-браузер, Mosaic, был выпущен в 1993 году Национальным центром суперкомпьютерных приложений (National Center for Supercomputing Applications, NCSA). Его строка пользовательского агента имела совсем простой формат:

```
Mosaic/0.9
```

Строка могла различаться в зависимости от операционной системы и платформы, но ничего сложного в ней не было: перед косой чертой указывалось название продукта (иногда как «NCSA Mosaic» или что-то подобное), а после — версия продукта.

Разработчики из Netscape Communications присвоили своему веб-браузеру кодовое название Mozilla (сокращение от «Mosaic Killer» — «убийца Mosaic»). В Netscape Navigator 2, первой общедоступной версии нового браузера, строка пользовательского агента имела следующий формат:

```
Mozilla/версия[язык] (платформа; шифрование)
```

Компания Netscape оставила название и версию продукта как начало строки пользовательского агента, но добавила позднее следующую информацию:

- **язык** — код языка, указывающий, где предполагается использовать браузер;
- **платформа** — операционная система и (или) платформа, на которой работает браузер;
- **шифрование** — разновидность шифрования; возможные значения — U (128-разрядное шифрование), I (40-разрядное шифрование) и N (нет шифрования).

Типичная строка пользовательского агента Netscape Navigator 2 выглядела следующим образом:

```
Mozilla/2.02 [fr] (WinNT; I)
```

Эта строка определяет браузер Netscape Navigator 2.02 для франкоязычных стран, запущенный на компьютере с системой Windows NT с 40-разрядным шифрованием. В те времена можно было легко идентифицировать браузер, просто прочитав название продукта в строке пользовательского агента.

Netscape Navigator 3 и Internet Explorer 3

В 1996 году был выпущен браузер Netscape Navigator 3, ставший на некоторое время самым популярным. Строка пользовательского агента претерпела в нем небольшие изменения: маркер языка из нее исчез, но были добавлены необязательные сведения об операционной системе или процессоре. Формат строки стал таким:

```
Mozilla/версия (платформа; шифрование [; описание ОС или ЦП])
```

Типичная строка пользовательского агента Netscape Navigator 3 в системе Windows выглядела так:

```
Mozilla/3.0 (Win95; U)
```

Эта строка соответствует Netscape Navigator 3 в системе Windows 95 со 128-разрядным шифрованием. Как видите, при работе под управлением Windows описание ОС или ЦП отсутствовало.

Вскоре после выпуска Netscape Navigator 3 корпорация Microsoft представила свой первый общедоступный веб-браузер Internet Explorer 3. Поскольку браузер Netscape тогда доминировал на рынке, многие серверы явно идентифицировали его, прежде чем отправлять страницы. Проблемы с доступом к страницам в Internet Explorer помешали бы его распространению, поэтому в Microsof выбрали для строки пользовательского агента формат, совместимый со строкой Netscape:

```
Mozilla/2.0 (compatible; версия MSIE; операционная система)
```

Например, у Internet Explorer 3.02 в системе Windows 95 была такая строка пользовательского агента:

```
Mozilla/2.0 (compatible; MSIE 3.02; Windows 95)
```

Для распознавания браузеров в то время обычно проверялось только имя продукта в строке пользовательского агента, поэтому Internet Explorer успешно выдавал себя за Mozilla, подражая Netscape Navigator. Не всем это понравилось, потому что такой подход нарушал конвенцию об идентификации браузеров, к тому же настоящая версия браузера была скрыта в середине строки.

Эта строка интересна еще и номером версии Mozilla. Казалось бы, вместо номера 2.0 логичнее было бы использовать 3.0, ведь именно эта версия была тогда наиболее популярной. Причина принятого решения остается загадкой. Скорее всего, это была банальная оплошность.

Netscape Communicator 4 и Internet Explorer 4–8

В августе 1997 года был представлен браузер Netscape Communicator 4 (в этом выпуске название браузера было изменено с *Navigator* на *Communicator*). Формат строки пользовательского агента остался в нем таким же, каким был в версии 3:

```
Mozilla/версия (платформа; шифрование [; описание ОС или ЦП])
```

В версии 4 на компьютере с системой Windows 98 строка пользовательского агента выглядела так:

```
Mozilla/4.0 (Win98; I)
```

По мере выпуска исправлений для браузера его версия увеличивалась. Вот, например, строка пользовательского агента для версии 4.79:

```
Mozilla/4.79 (Win98; I)
```

В Internet Explorer 4 строка пользовательского агента также содержала обновленную версию:

```
Mozilla/4.0 (compatible; версия MSIE; операционная система)
```

Например, браузер Internet Explorer 4 в Windows 98 возвращал следующую строку пользовательского агента:

```
Mozilla/4.0 (compatible; MSIE 4.0; Windows 98)
```

С этим изменением возвращаемая версия Mozilla и фактическая версия Internet Explorer совпали, что позволяет легко идентифицировать эти браузеры четвертого поколения. К сожалению, вскоре версии снова разошлись. Когда вышел браузер Internet Explorer 4.5 (доступный только для Mac), версия Mozilla осталась прежней, а версия Internet Explorer изменилась:

```
Mozilla/4.0 (compatible; MSIE 4.5; Mac_PowerPC)
```

В Internet Explorer этот формат использовался вплоть до версии 7:

```
Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1)
```

В Internet Explorer 8 был представлен дополнительный маркер Trident — имя визуализатора. Формат стал следующим:

```
Mozilla/4.0 (compatible; версия MSIE; операционная система;
             Trident/версия Trident)
```

Пример строки:

```
Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 5.1; Trident/4.0)
```

Маркер Trident позволяет определить, когда Internet Explorer 8 работает в режиме совместимости. В этом случае версия MSIE равна 7, но строка пользовательского агента содержит версию Trident:

```
Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; Trident/4.0)
```

Благодаря этому маркеру также можно отличить браузер Internet Explorer 7 (в котором нет маркера Trident) от браузера Internet Explorer 8, работающего в режиме совместимости.

В Internet Explorer 9 версии Mozilla и Trident были увеличены до 5.0. Строка пользовательского агента по умолчанию выглядит в Internet Explorer 9 так:

```
Mozilla/5.0 (compatible; MSIE 9.0; Windows NT 6.1; Trident/5.0)
```

Когда Internet Explorer 9 работает в режиме совместимости, восстанавливаются старые версии Mozilla и MSIE, а версия Trident остается равной 5.0. Например, Internet Explorer 9 в режиме совместимости с Internet Explorer 7 имеет следующую строку пользовательского агента:

```
Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 6.1; Trident/5.0)
```

Все эти изменения были внесены, чтобы расширить возможности передачи сведений о клиенте, не нарушив при этом работу прежних сценариев распознавания пользовательского агента.

Gecko

Визуализатор Gecko лежит в основе Firefox. Первоначально он был частью универсального браузера Mozilla, который позднее стал Netscape 6. Для Netscape 6 была разработана спецификация, определяющая формат строки пользовательского агента во всех будущих версиях. Новый формат существенно отличался от простой строки пользовательского агента, которая применялась до версии 4.x включительно, и был таким:

```
Mozilla/версия Mozilla (платформа; шифрование; ОС или ЦП; язык;
    предварительная версия) Gecko/версия Gecko
    приложение/версия приложения
```

Смысл каждого маркера в этой строке описан в следующей таблице.

Маркер	Обязательность	Описание
Версия Mozilla	Да	Версия Mozilla
Платформа	Да	Платформа, на которой работает браузер. Возможные значения включают Windows, Mac и X11 (для X-windows в Unix)
Шифрование	Да	Разновидность шифрования: U — для 128-разрядного, I — для 40-разрядного или N, если шифрование не используется
ОС ил ЦП	Да	Операционная система, в которой работает браузер, или тип процессора компьютера, на котором запущен браузер. Если платформа — Windows, это версия Windows (например, WinNT, Win95 и т. д.). Если платформа — Macintosh, это тип ЦП (68k, PPC для PowerPC или MacIntel). Если платформа — X11, это имя операционной системы Unix, возвращаемое командой uname –sm
Язык	Да	Язык, для которого создан браузер
Предварительная версия	Нет	Первоначально — предварительный номер версии Mozilla. Теперь — номер версии визуализатора Gecko
Версия Gecko	Да	Версия визуализатора Gecko, представленная датой в формате ггггммдд
Приложение	Нет	Название продукта, в котором используется Gecko. Им может быть Netscape, Firefox и т. д.
Версия приложения	Нет	Версия продукта, в котором используется Gecko; не путайте этот параметр с версией Mozilla и версией Gecko

Чтобы лучше понять формат строки пользовательского агента Gecko, взгляните на ее примеры из различных браузеров, основанных на Gecko.

Netscape 6.21 в Windows XP:

```
Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:0.9.4) Gecko/20011128
    Netscape6/6.2.1
```

SeaMonkey 1.1a в Linux:

```
Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.8.1b2) Gecko/20060823
    SeaMonkey/1.1a
```

Firefox 2.0.0.11 в Windows XP:

```
Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.8.1.11) Gecko/20071127
    Firefox/2.0.0.11
```

Camino 1.5.1 в Mac OS X:

```
Mozilla/5.0 (Macintosh; U; Intel Mac OS X; en; rv:1.8.1.6) Gecko/20070809
    Camino/1.5.1
```

Все эти строки определяют те или иные браузеры на основе Gecko. Часто идентифицировать конкретный браузер не требуется — достаточно узнать, что он основан на Gecko. Версия Mozilla 5.0 не изменялась начиная с выпуска первого браузера на основе Gecko и, вероятно, не изменится.

В Firefox 4 разработчики из Mozilla упростили строку пользовательского агента. Перечислим основные изменения.

- Маркер языка (`en-US` в приведенных примерах) удален.
- Маркер шифрования отсутствует, если используется сильное шифрование (включено по умолчанию). Это означает, что в строках пользовательского агента Mozilla теперь могут содержаться значения `"I"` и `"N"`, но не `"U"`.
- Маркер платформы удален из строк пользовательского агента Windows, так как он избыточен при наличии маркера «ОС или ЦП», который всегда содержит строку `"Windows"`.
- Маркер версии Gecko теперь имеет фиксированное значение `"Gecko/20100101"`.

Пример окончательной строки пользовательского агента Firefox 4:

```
Mozilla/5.0 (Windows NT 6.1; rv:2.0.1) Gecko/20100101 Firefox 4.0.1
```

WebKit

В 2003 году компания Apple анонсировала собственный веб-браузер под названием Safari. Визуализатор Safari, названный WebKit, начался как ответвление от проекта

KHTML — визуализатора браузера Konqueror для Linux. Через пару лет WebKit был преобразован в отдельный проект с открытым исходным кодом.

Разработчики нового браузера и визуализатора столкнулись с той же проблемой, что и создатели Internet Explorer 3: как гарантировать, что пользователям нового браузера будут доступны популярные сайты? Было решено добавить в строку пользовательского агента все сведения, необходимые для совместимости с другим популярным браузером. Итоговый формат оказался таким:

```
Mozilla/5.0 (платформа; шифрование; ОС или ЦП; язык)
    AppleWebKit/версия AppleWebKit (KHTML, like Gecko) Safari/версия Safari
```

Пример строки:

```
Mozilla/5.0 (Macintosh; U; PPC Mac OS X; en) AppleWebKit/124
    (KHTML, like Gecko) Safari/125.1
```

Как видите, получилась еще одна длинная строка пользовательского агента, содержащая не только версию Apple WebKit, но и версию Safari. Сомнения по поводу того, выдавать ли браузер за Mozilla, быстро отпали из соображений совместимости. Все браузеры на основе WebKit (и Gecko) идентифицируют себя как Mozilla 5.0. В качестве версии Safari обычно указывается номер сборки браузера, а представление номера выпуска не обязательно. Например, хотя для Safari 1.25 в строке пользовательского агента указан номер 125.1, однозначное соответствие наблюдается не всегда.

Наиболее интересным и спорным в этой строке пользовательского агента является элемент `"(KHTML, like Gecko)"`, добавленный в Safari до выпуска версии 1.0. Компания Apple получила много недовольных отзывов от разработчиков, которые восприняли это как наглую попытку выдать Safari за Gecko (как если бы добавления версии `Mozilla/5.0` было недостаточно). Ответ Apple был похож на реакцию Microsoft, когда критике подверглась строка пользовательского агента в Internet Explorer: браузер Safari совместим с Mozilla, и веб-сайты не должны блокировать пользователей Safari из-за того, что их браузер якобы не поддерживается.

В третьей версии Safari строка пользовательского агента была немного расширена. Теперь для идентификации фактической версии Safari используется следующий маркер:

```
Mozilla/5.0 (Macintosh; U; PPC Mac OS X; en) AppleWebKit/522.15.5
    (KHTML, like Gecko) Version/3.0.3 Safari/522.15.5
```

Это изменение внесено только в Safari, но не в WebKit, так что в других браузерах на основе WebKit оно может отсутствовать. Вообще говоря, как и в случае Gecko, обычно определяют, что браузер основан на WebKit, а не пытаются идентифицировать именно Safari.

Konqueror

Браузер Konqueror, поставляемый со средой KDE в Linux, основан на визуализаторе KHTML с открытым исходным кодом. Хотя Konqueror доступен только для Linux, он пользуется определенной популярностью. Ради совместимости для Konqueror была выбрана строка пользовательского агента, похожая на Internet Explorer:

```
Mozilla/5.0 (compatible; Konqueror/версия; ОС или ЦП)
```

В Konqueror 3.2 был добавлен идентификатор `KHTML` в соответствии с изменением строки пользовательского агента в WebKit:

```
Mozilla/5.0 (compatible; Konqueror/версия; ОС или ЦП) KHTML/версия KHTML
    (like Gecko)
```

Вот пример строки:

```
Mozilla/5.0 (compatible; Konqueror/3.5; SunOS) KHTML/3.5.0 (like Gecko)
```

Номера версий Konqueror и KHTML обычно совпадают или различаются после второй точки. Например, в Konqueror 3.5 используется модуль KHTML 3.5.1.

Chrome

Веб-браузер Chrome от Google использует WebKit в качестве визуализатора, но с другим интерпретатором JavaScript. Строка пользовательского агента в Chrome содержит все сведения из WebKit и дополнительный раздел версии Chrome. Ее формат таков:

```
Mozilla/5.0 (платформа; шифрование; ОС или ЦП; язык)
    AppleWebKit/версия AppleWebKit (KHTML, like Gecko)
    Chrome/версия Chrome Safari/версия Safari
```

Вот полная строка пользовательского агента для Chrome 7:

```
Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US) AppleWebKit/534.7
    (KHTML, like Gecko) Chrome/7.0.517.44 Safari/534.7
```

Вероятно, далее версии WebKit и Safari всегда будут синхронизированы, но это не гарантируется.

Opera

Строка пользовательского агента в Opera заслуживает наибольшего внимания. По умолчанию она логичнее, чем в любых других современных браузерах, поскольку правильно определяет браузер и его версию. До версии 8 строка пользовательского агента в Opera имела следующий формат:

```
Opera/версия (ОС или ЦП; шифрование) [язык]
```

В Opera 7.54 на компьютере с Windows XP строка пользовательского агента такова:

```
Opera/7.54 (Windows NT 5.1; U) [en]
```

В Opera 8 маркер языка был перемещен внутрь скобок для согласования с другими браузерами:

```
Opera/версия (ОС или ЦП; шифрование; язык)
```

Вот, например, строка пользовательского агента в Opera 8 для Windows XP:

```
Opera/8.0 (Windows NT 5.1; U; en)
```

По умолчанию Opera возвращает строку пользовательского агента в этом простом формате. В настоящее время это единственный из основных браузеров, который полностью идентифицирует себя с помощью названия и версии продукта. Однако как и в других браузерах, применение строки пользовательского агента в Opera связано с проблемами. Несмотря на то что технически она правильна, многие средства идентификации браузеров в Интернете ожидают строку пользовательского агента с названием Mozilla, а некоторые вообще настроены для распознавания Internet Explorer или Gecko. Чтобы не путать такие средства, Opera выдает себя за другой браузер, изменяя собственную строку пользовательского агента.

В Opera 9 это можно сделать двумя способами. Первый — заменить строку пользовательского агента строкой из Firefox или Internet Explorer с добавлением маркера `"Opera"` и номера версии Opera в конце, например:

```
Mozilla/5.0 (Windows NT 5.1; U; en; rv:1.8.1) Gecko/20061208 Firefox/2.0.0
    Opera 9.50

Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; en) Opera 9.50
```

Первая строка идентифицирует Opera 9.5 как Firefox 2 с сохранением сведений о версии Opera. Вторая строка выдает Opera 9.5 за Internet Explorer 6 и также содержит сведения о версии Opera. Эти строки пользовательского агента проходят большинство проверок для Firefox и Internet Explorer, но при желании по ним все же можно распознать Opera.

Второй способ — замаскировать Opera как Firefox или Internet Explorer. При этом строка пользовательского агента в Opera ничем не отличается от строк других браузеров: она не содержит ни маркер `"Opera"`, ни номер версии. Отличить Opera от других браузеров в этом случае невозможно. Еще больше дело запутывает то, что Opera часто задает строки пользовательского агента, специфичные для конкретных сайтов, не уведомляя об этом пользователя. Например, при переходе на сайт My Yahoo! (http://my.yahoo.com) Opera автоматически выдает себя за Firefox. Это делает идентификацию Opera по строке пользовательского агента очень непростой.

До версии 7 браузеры Opera могли интерпретировать строки операционной системы Windows. Например, Windows NT 5.1 на самом деле означает Windows XP, так что в Opera 6 строка пользовательского агента включала маркер Windows XP, а не Windows NT 5.1. Чтобы улучшить совместимость с другими браузерами, в Opera 7 и более поздних версий используется официальная версия операционной системы.

В Opera 10 формат строки пользовательского агента стал следующим:

```
Opera/9.80 (ОС или ЦП; шифрование; язык) Presto/версия Presto Version/версия
```

Версия Opera/9.80 стала фиксированной. На самом деле браузера Opera 9.8 не было, но разработчики Opera опасались, что средства распознавания браузеров могут ошибочно интерпретировать маркер Opera/10.0 как Opera 1, а не Opera 10. В связи с этим в Opera 10 были представлены дополнительные маркеры `Presto` (Presto — это визуализатор Opera) и `Version` (указывает фактическую версию браузера). Например, Opera 10.63 в системе Windows 7 имеет такую строку пользовательского агента:

```
Opera/9.80 (Windows NT 6.1; U; en) Presto/2.6.30 Version/10.63
```

iOS и Android

Веб-браузеры по умолчанию для мобильных операционных систем iOS и Android основаны на WebKit и имеют строки пользовательского агента почти такого же формата, что и их аналоги для настольных компьютеров. Вот формат iOS:

```
Mozilla/5.0 (платформа; шифрование; ОС или ЦП like Mac OS X; язык)
    AppleWebKit/версия AppleWebKit (KHTML, like Gecko)
    Version/версия браузера Mobile/мобильная версия Safari/версия Safari
```

Обратите внимание на строку `"like Mac OS X"`, которая помогает распознавать операционные системы для Mac, и новый маркер `Mobile`. Номер версии после этого маркера обычно бесполезен и применяется, если нужно различить версии WebKit для мобильных устройств и настольных компьютеров. Платформой может быть `"iPhone"`, `"iPod"` или `"iPad"` в зависимости от устройства, например:

```
Mozilla/5.0 (iPhone; U; CPU iPhone OS 3_0 like Mac OS X; en-us)
    AppleWebKit/528.18 (KHTML, like Gecko)
    Version/4.0 Mobile/7A341 Safari/528.16
```

До iOS 3 номера версии операционной системы не было в строке пользовательского агента.

В браузере Android по умолчанию строка пользовательского агента в целом соответствует формату iOS, но не содержит номер версии после маркера `Mobile`, например:

```
Mozilla/5.0 (Linux; U; Android 2.2; en-us; Nexus One Build/FRF91)
    AppleWebKit/533.1 (KHTML, like Gecko) Version/4.0 Mobile Safari/533.1
```

Эта строка получена с телефона Google Nexus One, но такой же формат используется и на других устройствах с системой Android.

Идентификация пользовательского агента

Идентифицировать браузер по строке пользовательского агента может быть непросто из-за запутанной истории и множества нюансов применения этих строк в современных браузерах. Часто сначала требуется определиться с тем, насколько специфичные сведения о браузере вам нужны. Обычно достаточно узнать визуализатор и семейство версий браузера. Например, следующий подход не рекомендуется:

```
if (isIE6 || isIE7) {    // не делайте так!!!
    // код
}
```

В этом примере код выполняется, если используется браузер Internet Explorer версии 6 или 7. Этот подход очень ненадежен, потому что в нем жестко закодированы конкретные версии браузера. А что если запущена версия 8? При выпуске каждой новой версии Internet Explorer этот код придется обновлять. Использование относительных номеров версий решает эту проблему:

```
if (ieVer >= 6){
    // код
}
```

В исправленном примере код выполняется, если Internet Explorer имеет версию не ниже 6. Это гарантирует, что сценарий продолжит правильно работать в будущем. В сценарии, который мы обсудим, такой подход используется для распознавания браузера.

Идентификация визуализатора

Как уже отмечалось, название и версия браузера не так важны, как его визуализатор. Если Firefox, Camino и Netscape используют одну и ту же версию Gecko, их возможности будут одинаковыми. То же можно сказать про WebKit. Наш сценарий будет распознавать пять основных визуализаторов: Internet Explorer, Gecko, WebKit, KHTML и Opera.

Для создания сценария мы воспользуемся расширенным паттерном Модуль, чтобы инкапсулировать код и не засорять глобальную область видимости ненужными переменными. Базовая структура кода такова:

```
var client = function(){

    var engine = {
```

```
        // визуализаторы
        ie: 0,
        gecko: 0,
        webkit: 0,
        khtml: 0,
        opera: 0,

        // конкретная версия
        ver: null
    };

    // распознавание визуализаторов/платформ/устройств

    return {
        engine: engine
};

}();
```

Глобальная переменная `client` будет содержать информацию о браузере. В анонимной функции объявляется локальная переменная `engine`, которой назначается литерал объекта с несколькими параметрами, предлагаемыми по умолчанию. Каждый визуализатор представляется свойством, которое инициализируется значением 0. При распознавании конкретного визуализатора его версия назначается соответствующему свойству как значение с плавающей точкой, а полная версия визуализатора (строка) присваивается свойству `ver`. Такой код можно использовать следующим образом:

```
if (client.engine.ie) {  // если используется IE, client.engine.ie больше 0
    // код, специфичный для IE
} else if (client.engine.gecko > 1.5){
    if (client.engine.ver == "1.8.1"){
        // код, специфичный для этой версии
    }
}
```

При распознавании визуализатора его свойству в объекте `client.engine` присваивается число больше 0, которое соответствует логическому значению `true`. Это позволяет задействовать свойство в условии `if` для определения визуализатора даже без выяснения его конкретной версии. Поскольку каждое из пяти свойств модулей содержит значение с плавающей точкой, некоторые сведения о версии могут быть потеряны. Например, строка `"1.8.1"` преобразуется методом `parseFloat()` в число 1.8. Для хранения полной версии модуля визуализации предусмотрено свойство `ver`.

Чтобы правильно идентифицировать визуализатор, нужно выполнять проверку в определенном порядке, иначе из-за несоответствий пользовательских агентов могут быть получены ошибочные результаты. Первым нужно идентифицировать браузер Opera, потому что он может полностью имитировать строки пользовательского агента других браузеров.

Чтобы идентифицировать Opera, нужно проверить объект `window.opera`, который имеется во всех версиях Opera начиная с пятой и используется для распознавания браузера и прямого взаимодействия с ним. Начиная с Opera 7.6 можно получить версию браузера как строку с помощью метода `version()` — это оптимальный способ определения номера версии Opera. Более ранние версии можно идентифицировать по строке пользовательского агента, поскольку маскировка браузера тогда еще не поддерживалась. Тем не менее на конец 2010 года последней версией Opera была версия 10.63, поэтому маловероятно, что кто-то будет использовать более старую версию, чем 7.6. Начать распознавание визуализатора можно так:

```
if (window.opera){
    engine.ver = window.opera.version();
    engine.opera = parseFloat(engine.ver);
}
```

Строковое представление версии сохраняется в свойстве `engine.ver`, а представление в формате числа с плавающей точкой — в свойстве `engine.opera`. Если браузером является Opera, условие `window.opera` возвращает `true` и выполняется приведенный код, иначе нужно продолжить распознавание браузера.

Далее по логике следует попытаться идентифицировать визуализатор WebKit. Поскольку его строка пользовательского агента содержит маркеры `"Gecko"` и `"KHTML"`, если сначала идентифицировать эти визуализаторы, можно получить неправильный результат.

Только строка пользовательского агента в WebKit содержит маркер `"AppleWebKit"`, так что логичнее всего искать его. Вот как это можно сделать:

```
var ua = navigator.userAgent;

if (window.opera){
    engine.ver = window.opera.version();
    engine.opera = parseFloat(engine.ver);
} else if (/AppleWebKit\/(\S+)/.test(ua)){
    engine.ver = RegExp["$1"];
    engine.webkit = parseFloat(engine.ver);
}
```

Этот код начинается с сохранения строки пользовательского агента в переменной `ua`. Регулярное выражение проверяет наличие маркера `"AppleWebKit"` в строке пользовательского агента и применяет группу захвата к номеру версии. Поскольку фактический номер версии может содержать цифры, точки и буквы, для сопоставления с ним используется специальный знак непустого места (`\S`). Разделителем между номером версии и следующей частью строки пользовательского агента является пробел, так что этот паттерн гарантирует захват версии целиком. Метод `test()` выполняет регулярное выражение для строки пользовательского агента. Если он возвращает `true`, захваченный номер версии сохраняется в свойстве `engine.ver`, а его представление в виде числа с плавающей точкой — в свойстве `engine.webkit`. Соответствия между версиями WebKit и Safari указаны в таблице.

Версия Safari	Минимальная версия WebKit
1.0–1.0.2	85.7
1.0.3	85.8.2
1.1–1.1.1	100
1.2.2	125.2
1.2.3	125.4
1.2.4	125.5.5
1.3	312.1
1.3.1	312.5
1.3.2	312.8
2.0	412
2.0.1	412.7
2.0.2	416.11
2.0.3	417.9
2.0.4	418.8
3.0.4	523.10
3.1	525

Иногда версии Safari не совсем точно соответствуют версиям WebKit и могут отличаться после второй точки. В приведенной таблице указаны наиболее вероятные версии WebKit.

Далее следует попытаться идентифицировать визуализатор KHTML. Его строка пользовательского агента содержит маркер `"Gecko"`, так что нельзя надежно распознать браузер на основе Gecko, не исключив сначала KHTML. Версия KHTML содержится в строке пользовательского агента в формате, похожем на WebKit, поэтому для ее поиска применяется похожее регулярное выражение. Так как Konqueror 3.1 и более ранних версий не включают маркер KHTML, проверяется также версия Konqueror:

```
var ua = navigator.userAgent;

if (window.opera){
    engine.ver = window.opera.version();
    engine.opera = parseFloat(engine.ver);
} else if (/AppleWebKit\/(\S+)/.test(ua)){
    engine.ver = RegExp["$1"];
    engine.webkit = parseFloat(engine.ver);
} else if (/KHTML\/(\S+)/.test(ua) || /Konqueror\/([^;]+)/.test(ua)){
    engine.ver = RegExp["$1"];
    engine.khtml = parseFloat(engine.ver);
}
```

Номер версии KHTML отделен от следующего маркера пробелом, поэтому для захвата всех символов версии также используется знак непустого места. Затем версия в строковом формате сохраняется в свойстве `engine.ver`, а версия в формате числа с плавающей точкой — в свойстве `engine.khtml`. Если в строке пользовательского агента нет маркера KHTML, выполняется сопоставление с маркером Konqueror, косой чертой после него и всеми знаками до точки с запятой.

Исключив WebKit и KHTML, можно перейти к идентификации Gecko. Фактическая версия Gecko содержится в строке пользовательского агента не после маркера `"Gecko"`, а после символов `"rv:"`. Это требует более сложного регулярного выражения, чем предыдущие проверки:

```
var ua = navigator.userAgent;

if (window.opera){
    engine.ver = window.opera.version();
    engine.opera = parseFloat(engine.ver);
} else if (/AppleWebKit\/(\S+)/.test(ua)){
    engine.ver = RegExp["$1"];
    engine.webkit = parseFloat(engine.ver);
} else if (/KHTML\/(\S+)/.test(ua)){
    engine.ver = RegExp["$1"];
    engine.khtml = parseFloat(engine.ver);
} else if (/rv:([^\)]+)\) Gecko\/\d{8}/.test(ua)){
    engine.ver = RegExp["$1"];
    engine.gecko = parseFloat(engine.ver);
}
```

Номер версии Gecko содержится между символами `"rv:"` и закрывающей скобкой, и чтобы извлечь его, регулярное выражение ищет все символы, которые отличаются от закрывающей скобки. Также регулярное выражение ищет строку `"Gecko/"`, за которой следуют восемь цифр. Если совпадение с шаблоном обнаруживается, номер версии извлекается и сохраняется в соответствующих свойствах. Номера версий Gecko и Firefox связаны следующим образом:

Версия Firefox	Минимальная версия Gecko
1.0	1.7.5
1.5	1.8.0
2.0	1.8.1
3.0	1.9.0
3.5	1.9.1
3.6	1.9.2
4.0	2.0.0

Как и в случае Safari и WebKit, соответствия между версиями Firefox и Gecko точны не всегда.

Наконец, можно идентифицировать визуализатор Internet Explorer. Его номер версии находится между маркером "MSIE" и точкой с запятой, так что последнее регулярное выражение простое:

```
var ua = navigator.userAgent;

if (window.opera){
    engine.ver = window.opera.version();
    engine.opera = parseFloat(engine.ver);
} else if (/AppleWebKit\/(\S+)/.test(ua)){
    engine.ver = RegExp["$1"];
    engine.webkit = parseFloat(engine.ver);
} else if (/KHTML\/(\S+)/.test(ua)){
    engine.ver = RegExp["$1"];
    engine.khtml = parseFloat(engine.ver);
} else if (/rv:([^\)]+)\) Gecko\/\d{8}/.test(ua)){
    engine.ver = RegExp["$1"];
    engine.gecko = parseFloat(engine.ver);
} else if (/MSIE ([^;]+)/.test(ua)){
    engine.ver = RegExp["$1"];
    engine.ie = parseFloat(engine.ver);
}
```

Для получения всех символов, отличных от точки с запятой, в последней части этого сценария применяется класс отрицания. Хотя в Internet Explorer номера версий обычно хранятся как стандартные значения с плавающей точкой, возможно, когда-нибудь ситуация изменится. Класс отрицания `[^;]` будет работать, даже если в номере версии появятся дополнительные точки и буквы.

Идентификация браузера

В большинстве случаев для принятия решения достаточно идентифицировать визуализатор, но сам по себе визуализатор не гарантирует доступность нужной JavaScript-функциональности. Браузеры Apple Safari и Google Chrome содержат визуализатор WebKit, но используют разные интерпретаторы JavaScript. Оба браузера задают значение свойства `client.engine.webkit`, но это может быть недостаточно конкретно, поэтому имеет смысл добавить в объект `client` еще один набор свойств:

```
var client = function(){

    var engine = {

        // визуализаторы
        ie: 0,
        gecko: 0,
        webkit: 0,
        khtml: 0,
        opera: 0,

        // конкретная версия
        ver: null
```

```
    };

    var browser = {

        // браузеры
        ie: 0,
        firefox: 0,
        safari: 0,
        konq: 0,
        opera: 0,
        chrome: 0,

        // конкретная версия
        ver: null
    };

    // распознавание визуализаторов/платформ/устройств

    return {
        engine: engine,
        browser: browser
    };

}();
```

Мы добавили в код закрытую переменную `browser` со свойствами для каждого основного браузера, которые инициализируются нулями. Когда удается распознать браузер, его версия в формате числа с плавающей точкой сохраняется в соответствующем свойстве, а полное строковое представление версии — в свойстве `ver`. Как показывает следующий пример, код распознавания браузера перемежается кодом распознавания визуализатора из-за тесной связи между ними в большинстве браузеров:

```
// распознавание визуализаторов/браузеров
var ua = navigator.userAgent;
if (window.opera){
    engine.ver = browser.ver = window.opera.version();
    engine.opera = browser.opera = parseFloat(engine.ver);
} else if (/AppleWebKit\/(\S+)/.test(ua)){
    engine.ver = RegExp["$1"];
    engine.webkit = parseFloat(engine.ver);

    //это Chrome или Safari?
    if (/Chrome\/(\S+)/.test(ua)){
        browser.ver = RegExp["$1"];
        browser.chrome = parseFloat(browser.ver);
    } else if (/Version\/(\S+)/.test(ua)){
        browser.ver = RegExp["$1"];
        browser.safari = parseFloat(browser.ver);
    } else {
        // примерная версия
        var safariVersion = 1;
        if (engine.webkit < 100){
            safariVersion = 1;
        } else if (engine.webkit < 312){
            safariVersion = 1.2;
        } else if (engine.webkit < 412){
```

```
            safariVersion = 1.3;
        } else {
            safariVersion = 2;
        }

        browser.safari = browser.ver = safariVersion;
    }
} else if (/KHTML\/(\S+)/.test(ua) || /Konqueror\/([^;]+)/.test(ua)){
    engine.ver = browser.ver = RegExp["$1"];
    engine.khtml = browser.konq = parseFloat(engine.ver);
} else if (/rv:([^\)]+)\) Gecko\/\d{8}/.test(ua)){
    engine.ver = RegExp["$1"];
    engine.gecko = parseFloat(engine.ver);

    //это Firefox?
    if (/Firefox\/(\S+)/.test(ua)){
        browser.ver = RegExp["$1"];
        browser.fi refox = parseFloat(browser.ver);
    }
} else if (/MSIE ([^;]+)/.test(ua)){
    engine.ver = browser.ver = RegExp["$1"];
    engine.ie = browser.ie = parseFloat(engine.ver);
}
```

Для Opera и Internet Explorer свойствам объекта `browser` присваиваются те же значения, что и свойствам `engine`. В случае Konqueror свойства `browser.kong` и `browser.ver` эквивалентны свойствам `engine.khtml` и `engine.ver` соответственно.

Для идентификации Chrome и Safari в код распознавания визуализатора добавлены дополнительные инструкции `if`. Чтобы извлечь номер версии Chrome, выполняется поиск строки `"Chrome/"` и считываются цифры после нее, а для распознавания Safari считываются цифры после строки `"Version/"`. Поскольку этот механизм работает только для Safari 3 и более поздних версий, в отдельном блоке кода номера версий WebKit сопоставляются с примерными номерами версий Safari (см. таблицу в предыдущем разделе).

Для получения версии Firefox считываются цифры после строки `"Firefox/"`. Это выполняется только при обнаружении визуализатора Gecko.

Используя этот код, можно реализовать следующую логику:

```
if (client.engine.webkit) {        // обнаружен WebKit
    if (client.browser.chrome){
        // какие-то действия для Chrome
    } else if (client.browser.safari){
        // какие-то действия для Safari
    }
} else if (client.engine.gecko){
    if (client.browser.firefox){
        // какие-то действия для Firefox
    } else {
        // какие-то действия для других браузеров с Gecko
    }
}
```

Идентификация платформы

Во многих случаях для выбора нужного варианта кода достаточно определить визуализатор, но иногда нужно узнать еще и платформу, потому что от этого могут зависеть те или иные аспекты работы браузера. Три основные платформы — это Windows, Mac и Unix (включая разновидности Linux). Чтобы можно было идентифицировать их, давайте добавим в наш код еще один объект:

```
var client = function(){

    var engine = {

        // визуализаторы
        ie: 0,
        gecko: 0,
        webkit: 0,
        khtml: 0,
        opera: 0,

        // конкретная версия
        ver: null
    };

    var browser = {

        // браузеры
        ie: 0,
        firefox: 0,
        safari: 0,
        konq: 0,
        opera: 0,
        chrome: 0,

        // конкретная версия
        ver: null
    };

    var system = {
        win: false,
        mac: false,
        x11: false
    };

    // распознавание визуализаторов/платформ/устройств

    return {
        engine: engine,
        browser: browser,
        system: system
    };

}();
```

Этот код содержит новую переменную `system` со свойствами `win`, `mac` и `x11`, которые соответствуют платформам Windows, Mac и Unix. В отличие от визуализаторов, сведения о платформе обычно очень ограниченны и не содержат деталей. Из этих трех платформ версию операционной системы сообщает браузеру только Windows,

поэтому каждое из этих свойств представлено первоначально логическим значением `false`, а не числом.

Чтобы определить платформу, гораздо проще прочитать свойство `navigator.platform`, а не строку пользовательского агента, в которой формат сведений о платформе зависит от браузера. Свойство `navigator.platform` поддерживает значения `"Win32"`, `"Win64"`, `"MacPPC"`, `"MacIntel"`, `"X11"` и `"Linux i686"`, которые одинаковы во всех браузерах. Код идентификации платформы совсем прост:

```
var p = navigator.platform;
system.win = p.indexOf("Win") == 0;
system.mac = p.indexOf("Mac") == 0;
system.x11 = (p.indexOf("X11") == 0) || (p.indexOf("Linux") == 0);
```

Метод `indexOf()` сравнивает начало строки платформы с возможными значениями свойства. Обнаружение подстроки `"Win"` (охватывает варианты `"Win32"` и `"Win64"`) указывает, что используется Windows, подстрока `"Mac"` (`"MacPPC"` или `"MacIntel"`) соответствует платформе Mac, а для идентификации Unix выполняется сравнение со значениями `"X11"` и `"Linux"`.

В ранних версиях Gecko свойство navigator.platform возвращало значение «Windows» для всех платформ Windows и «Macintosh» для всех платформ Mac. В Firefox 1 эти значения были исправлены.

Идентификация операционных систем Windows

На платформе Windows из строки пользовательского агента можно получить сведения об операционной системе. До Windows XP существовали две серии систем Windows: для дома и для бизнеса. Серия для дома называлась просто Windows и включала версии 95, 98 и ME. Серия для бизнеса включала системы Windows NT и Windows 2000. В Windows XP эти две серии были объединены в общей базе кода, основанной на Windows NT. Затем на основе Windows XP была разработана система Windows Vista.

История систем Windows важна для понимания того, как они представляются в строках пользовательского агента разных браузеров. Эти сведения приведены в следующей таблице.

Версия Windows	IE 4+	Gecko	Opera < 7	Opera 7+	WebKit
95	"Windows 95"	"Win95"	"Windows 95"	"Windows 95"	—
98	"Windows 98"	"Win98"	"Windows 98"	"Windows 98"	—
NT 4.0	"Windows NT"	"WinNT4.0"	"Windows NT 4.0"	"Windows NT 4.0"	—

Продолжение

Версия Windows	IE 4+	Gecko	Opera < 7	Opera 7+	WebKit
2000	"Windows NT 5.0"	"Windows NT 5.0"	"Windows 2000"	"Windows NT 5.0"	—
ME	"Win 9x 4.90"	"Win 9x 4.90"	"Windows ME"	"Win 9x 4.90"	—
XP	"Windows NT 5.1"	"Windows NT 5.1"	"Windows XP"	"Windows NT 5.1"	"Windows NT 5.1"
Vista	"Windows NT 6.0"	"Windows NT 6.0"	—	"Windows NT 6.0"	"Windows NT 6.0"
7	"Windows NT 6.1"	"Windows NT 6.1"	—	"Windows NT 6.1"	"Windows NT 6.1"

Из-за разных способов представления систем Windows в строках пользовательского агента идентифицировать их не просто. К счастью, начиная с Windows 2000 формат версий Windows остается практически тем же, изменяются только номера. Для идентификации разных систем Windows мы составим регулярное выражение. Помните, что версии Opera до 7 уже почти не используются, так что можно их не учитывать.

Первым делом следует распознать версии Windows 95 и Windows 98. Единственное отличие строк, возвращаемых Gecko, от строк других браузеров, — это отсутствие букв `"dows"` и пробела между символами `"Win"` и номером версии. Соответствующее регулярное выражение будет таким:

```
/Win(?:dows )?([^do]{2})/
```

Группа захвата в этом выражении возвращает версию операционной системы. Версией может быть любое сочетание из двух символов, не содержащее `"d"` или `"o"` (например, 95, 98, 9x, NT, ME или XP), поэтому используются два знака непустого места.

В Gecko для Windows NT в конец версии добавляются символы `"4.0"`. Вместо точного совпадения с этой строкой имеет смысл искать десятичную дробь:

```
/Win(?:dows )?([^do]{2})(\d+\.\d+)?/
```

Вторая группа захвата в этом регулярном выражении сопоставляется с номером версии NT. Поскольку в Windows 95 и 98 этот номер отсутствует, он указан как необязательный. Единственное различие между этим шаблоном и представлением Windows NT в Opera — пробел между `"NT"` и `"4.0"`, добавить который совсем не сложно:

```
/Win(?:dows )?([^do]{2})\s?(\d+\.\d+)?/
```

После этих изменений регулярное выражение также соответствует строкам систем Windows ME, Windows XP и Windows Vista. Первая группа захвата распознает версии 95, 98, 9x, NT, ME и XP, вторая — Windows ME и все производные Windows NT. Полученную информацию можно использовать для установки свойства `system.win`:

```
if (system.win){
    if (/Win(?:dows )?([^do]{2})\s?(\d+\.\d+)?/.test(ua)){
        if (RegExp["$1"] == "NT"){
            switch(RegExp["$2"]){
                case "5.0":
                    system.win = "2000";
                    break;
                case "5.1":
                    system.win = "XP";
                    break;
                case "6.0":
                    system.win = "Vista";
                    break;
                case "6.1":
                    system.win = "7";
                    break;
                default:
                    system.win = "NT";
                    break;
            }
        } else if (RegExp["$1"] == "9x"){
            system.win = "ME";
        } else {
            system.win = RegExp["$1"];
        }
    }
}
```

Итак, если свойство `system.win` имеет значение `true`, регулярное выражение извлекает более точные сведения о системе из строки пользовательского агента. Возможно, какие-то будущие версии Windows не будут так распознаваться, поэтому нужно проверить, обнаружен ли в строке шаблон. Если да, первая группа захвата содержит значение `"95"`, `"98"`, `"9x"` или `"NT"`. Если захвачено значение `"NT"`, свойству `system.win` присваивается более точный идентификатор распознанной операционной системы; если захвачено значение `"9x"`, свойству присваивается строка `"ME"`, в противном случае ему присваивается само захваченное значение. Это позволяет писать подобный код:

```
if (client.system.win){
    if (client.system.win == "XP") {
        // используется Windows XP
    } else if (client.system.win == "Vista"){
        // используется Windows Vista
    }
}
```

Так как непустой строке соответствует логическое значение `true`, свойство `client.system.win` можно указывать как условие инструкции `if`, но его же можно

использовать как строковое значение для получения дополнительных сведений об операционной системе.

Идентификация мобильных устройств

В 2006–2007 годах мы стали свидетелями бума на рынке браузеров для мобильных устройств. У всех основных браузеров уже есть такие версии, и важно уметь их идентифицировать. Для начала нужно добавить в код соответствующие свойства:

```
var client = function(){

    var engine = {

        // визуализаторы
        ie: 0,
        gecko: 0,
        webkit: 0,
        khtml: 0,
        opera: 0,

        // конкретная версия
        ver: null
    };

    var browser = {

        // браузеры
        ie: 0,
        firefox: 0,
        safari: 0,
        konq: 0,
        opera: 0,
        chrome: 0,

        // конкретная версия
        ver: null
    };

    var system = {
        win: false,
        mac: false,
        x11: false,

        // мобильные устройства
        iphone: false,
        ipod: false,
        ipad: false,
        ios: false,
        android: false,
        nokiaN: false,
        winMobile: false    };

    // распознавание визуализаторов/платформ/устройств
```

```
    return {
        engine: engine,
        browser: browser,
        system: system
    };

}();
```

Для идентификации устройств с iOS достаточно выполнить поиск строк `"iPhone"`, `"iPod"` и `"iPad"`:

```
system.iphone = ua.indexOf("iPhone") > -1;
system.ipod = ua.indexOf("iPod") > -1;
system.ipad = ua.indexOf("iPad") > -1;
```

Распознав одно из этих устройств, полезно также узнать версию iOS. В iOS до версии 3 строка пользовательского агента содержала фрагмент `"CPU like Mac OS X"`, который позднее был изменен на `"CPU iPhone OS 3_0 like Mac OS X"` для iPhone и `"CPU OS 3_2 like Mac OS X"` для iPad. Это означает, что для идентификации iOS требуется такое регулярное выражение:

```
// определение версии iOS
if (system.mac && ua.indexOf("Mobile") > -1){
    if (/CPU (?:iPhone )?OS (\d+_\d+)/.test(ua)){
        system.ios = parseFloat(RegExp.$1.replace("_", "."));
    } else {
        system.ios = 2;    // невозможно определить, поэтому предполагаем
    }
}
```

Если система распознается как Mac OS, а строка пользовательского агента содержит слово `"Mobile"`, значение `system.ios` гарантированно будет ненулевым независимо от версии iOS. В этом случае регулярное выражение определяет, есть ли версия iOS в строке пользовательского агента. Если да, она присваивается свойству `system.ios` как число с плавающей точкой, в противном случае определить версию невозможно, поэтому свойству присваивается число 2, предыдущее значение версии.

Чтобы идентифицировать операционную систему Android, достаточно найти строку `"Android"` и получить номер версии сразу после нее:

```
// определение версии Android
if (/Android (\d+\.\d+)/.test(ua)){
    system.android = parseFloat(RegExp.$1);
}
```

Поскольку все версии Android включают номер версии, это регулярное выражение всегда присваивает свойству `system.android` правильное значение.

Мобильные телефоны Nokia N содержат браузер на основе WebKit со строкой пользовательского агента, типичной для телефонов с WebKit:

```
Mozilla/5.0 (SymbianOS/9.2; U; Series60/3.1 NokiaN95/11.0.026;
    Profile MIDP-2.0 Configuration/CLDC-1.1)
    AppleWebKit/413 (KHTML, like Gecko) Safari/413
```

Телефоны Nokia N возвращают строку пользовательского агента со словом `"Safari"`, но на самом деле их браузер не является Safari, хотя и основан на WebKit. Эту серию телефонов можно определить по наличию значения `"NokiaN"` в строке пользовательского агента:

```
system.nokiaN = ua.indexOf("NokiaN") > -1;
```

Располагая этими сведениями, можно выяснить, с какого устройства с браузером WebKit пользователь получает доступ к странице:

```
if (client.engine.webkit){
    if (client.system.ios){
        // код для iOS
    } else if (client.system.android){
        // код для android
    } else if (client.system.nokiaN){
        // код для nokia
    }
}
```

Последняя основная платформа для мобильных устройств — Windows Mobile, ранее известная как Windows CE и доступная для Pocket PC и смартфонов. Поскольку технически эти устройства относятся к Windows, параметры платформы и операционной системы возвращают правильные значения. В Windows Mobile 5.0 и более ранних версий строки пользовательского агента этих двух устройств очень похожи:

```
Mozilla/4.0 (compatible; MSIE 4.01; Windows CE; PPC; 240x320)
Mozilla/4.0 (compatible; MSIE 4.01; Windows CE; Smartphone; 176x220)
```

Первая строка принадлежит браузеру Internet Explorer 4.01 на Pocket PC, вторая — ему же на смартфоне. Если запустить сценарий идентификации операционной системы Windows для любой из этих строк, свойство `client.system.win` получит значение `"CE"`, так что по этим строкам можно идентифицировать ранние устройства с Windows Mobile.

Проверять значения `"PPC"` и `"Smartphone"` не рекомендуется, потому что эти маркеры удалены из браузеров для Windows Mobile позднее версии 5.0. Часто распознать систему Windows Mobile вполне достаточно.

Windows Phone 7 имеет слегка расширенную строку пользовательского агента следующего формата:

```
Mozilla/4.0 (compatible; MSIE 7.0; Windows Phone OS 7.0; Trident/3.1;
    IEMobile/7.0) Asus;Galaxy6
```

Формат идентификатора Windows здесь отличается от традиционного, так что при обработке такой строки пользовательского агента свойство `client.system.win` получает значение `"Ph"`. Этот факт позволяет получить дополнительные сведения о системе:

```
// windows mobile
if (system.win == "CE"){
    system.winMobile = system.win;
} else if (system.win == "Ph"){
    if(/Windows Phone OS (\d+.\d+)/.test(ua)){;
        system.win = "Phone";
        system.winMobile = parseFloat(RegExp["$1"]);
    }
}
```

Если свойство `system.win` равно `"CE"`, это означает, что мы имеем дело со старой версией Windows Mobile и можно присвоить это значение свойству `system.winMobile` (это все, что мы знаем). Если же свойство `system.win` имеет значение `"Ph"`, вероятно, используется устройство Windows Phone 7 или более поздней версии, поэтому можно узнать номер версии с помощью регулярного выражения. После этого значение `system.win` изменяется на `"Phone"`, а свойству `system.winMobile` присваивается номер версии.

Идентификация игровых систем

Еще один сегмент, где веб-браузеры набирают популярность, — это системы для видеоигр. И Nintendo Wii, и Playstation 3 поддерживают веб-браузеры, которые либо встроены, либо доступны для загрузки. Браузер для Wii на самом деле является разновидностью браузера Opera, а в Playstation применяется браузер, не основанный ни на одном из описанных визуализаторов. Строки пользовательского агента в этих браузерах таковы:

```
Opera/9.10 (Nintendo Wii;U; ; 1621; en)
Mozilla/5.0 (PLAYSTATION 3; 2.00)
```

Первая строка пользовательского агента определяет Opera в Wii. Как видите, она соответствует оригинальному формату браузера Opera, который в Wii не поддерживает маскировку. Вторая строка, взятая из Playstation 3, выдает браузер за Mozilla 5.0 ради совместимости, но не предоставляет подробных сведений. Как ни странно, название устройства в ней содержит только прописные буквы — это наводит на мысль, что в будущем регистр может измениться.

Перед распознаванием этих устройств нужно добавить соответствующие свойства в объект `client.system`:

```
var client = function(){

    var engine = {
```

```
        // визуализаторы
        ie: 0,
        gecko: 0,
        webkit: 0,
        khtml: 0,
        opera: 0,

        // конкретная версия
        ver: null
    };

    var browser = {

        // браузеры
        ie: 0,
        firefox: 0,
        safari: 0,
        konq: 0,
        opera: 0,
        chrome: 0,

        // конкретная версия
        ver: null
    };

    var system = {
        win: false,
        mac: false,
        x11: false,

        // мобильные устройства
        iphone: false,
        ipod: false,
        ipad: false,
        ios: false,
        android: false,
        nokiaN: false,
        winMobile: false,
        // игровые системы
        wii: false,
        ps: false
    };

    // распознавание визуализаторов/платформ/устройств

    return {
        engine: engine,
        browser: browser,
        system: system
    };

}();
```

Следующий код распознает каждую из этих игровых систем:

```
system.wii = ua.indexOf("Wii") > -1;
system.ps = /playstation/i.test(ua);
```

Для Wii хватит простого поиска строки "Wii". Остальной код определит, что браузером является Opera, и возвратит правильный номер его версии в свойстве client.browser.opera. Для распознавания Playstation применяется регулярное выражение, которое сопоставляется со строкой пользовательского агента без учета регистра.

Полный сценарий

Далее приведен полный сценарий распознавания пользовательских агентов, в том числе визуализаторов, платформ, операционных систем Windows, мобильных устройств и игровых систем:

Листинг client.js


```
var client = function(){

    // визуализаторы
    var engine = {
        ie: 0,
        gecko: 0,
        webkit: 0,
        khtml: 0,
        opera: 0,

        // конкретная версия
        ver: null
    };

    // браузеры
    var browser = {

        // браузеры
        ie: 0,
        firefox: 0,
        safari: 0,
        konq: 0,
        opera: 0,
        chrome: 0,

        // конкретная версия
        ver: null
    };

    // платформа/устройство/ОС
    var system = {
        win: false,
        mac: false,
        x11: false,

        // мобильные устройства
        iphone: false,
        ipod: false,
        ipad: false,
        ios: false,
```

```
        android: false,
        nokiaN: false,
        winMobile: false,

        // игровые системы
        wii: false,
        ps: false
    };

    // распознавание визуализаторов/браузеров
    var ua = navigator.userAgent;
    if (window.opera){
        engine.ver = browser.ver = window.opera.version();
        engine.opera = browser.opera = parseFloat(engine.ver);
    } else if (/AppleWebKit\/(\S+)/.test(ua)){
        engine.ver = RegExp["$1"];
        engine.webkit = parseFloat(engine.ver);

        // это Chrome или Safari?
        if (/Chrome\/(\S+)/.test(ua)){
            browser.ver = RegExp["$1"];
            browser.chrome = parseFloat(browser.ver);
        } else if (/Version\/(\S+)/.test(ua)){
            browser.ver = RegExp["$1"];
            browser.safari = parseFloat(browser.ver);
        } else {
            // примерная версия
            var safariVersion = 1;
            if (engine.webkit < 100){
                safariVersion = 1;
            } else if (engine.webkit < 312){
                safariVersion = 1.2;
            } else if (engine.webkit < 412){
                safariVersion = 1.3;
            } else {
                safariVersion = 2;
            }

            browser.safari = browser.ver = safariVersion;
        }
    } else if (/KHTML\/(\S+)/.test(ua) || /Konqueror\/([^;]+)/.test(ua)){
        engine.ver = browser.ver = RegExp["$1"];
        engine.khtml = browser.konq = parseFloat(engine.ver);
    } else if (/rv:([^\)]+)\) Gecko\/\d{8}/.test(ua)){
        engine.ver = RegExp["$1"];
        engine.gecko = parseFloat(engine.ver);

        // это Firefox?
        if (/Firefox\/(\S+)/.test(ua)){
            browser.ver = RegExp["$1"];
            browser.firefox = parseFloat(browser.ver);
        }
    } else if (/MSIE ([^;]+)/.test(ua)){
        engine.ver = browser.ver = RegExp["$1"];
        engine.ie = browser.ie = parseFloat(engine.ver);
```

```
}

// распознавание браузера
browser.ie = engine.ie;
browser.opera = engine.opera;


// распознавание платформы
var p = navigator.platform;
system.win = p.indexOf("Win") == 0;
system.mac = p.indexOf("Mac") == 0;
system.x11 = (p == "X11") || (p.indexOf("Linux") == 0);

// распознавание операционных систем Windows
if (system.win){
    if (/Win(?:dows )?([^do]{2})\s?(\d+\.\d+)?/.test(ua)){
        if (RegExp["$1"] == "NT"){
            switch(RegExp["$2"]){
                case "5.0":
                    system.win = "2000";
                    break;
                case "5.1":
                    system.win = "XP";
                    break;
                case "6.0":
                    system.win = "Vista";
                    break;
                case "6.1":
                    system.win = "7";
                    break;
                default:
                    system.win = "NT";
                    break;
            }
        } else if (RegExp["$1"] == "9x"){
            system.win = "ME";
        } else {
            system.win = RegExp["$1"];
        }
    }
}

// мобильные устройства
system.iphone = ua.indexOf("iPhone") > -1;
system.ipod = ua.indexOf("iPod") > -1;
system.ipad = ua.indexOf("iPad") > -1;
system.nokiaN = ua.indexOf("NokiaN") > -1;

// Windows Mobile
if (system.win == "CE"){
    system.winMobile = system.win;
} else if (system.win == "Ph"){
    if(/Windows Phone OS (\d+.\d+)/.test(ua)){;
        system.win = "Phone";
        system.winMobile = parseFloat(RegExp["$1"]);
```

```
        }
    }
    // определение версии iOS
    if (system.mac && ua.indexOf("Mobile") > -1){
        if (/CPU (?:iPhone )?OS (\d+_\d+)/.test(ua)){
            system.ios = parseFloat(RegExp.$1.replace("_", "."));
        } else {
            system.ios = 2;  // невозможно определить, поэтому предполагаем
        }
    }

    // определение версии Android
    if (/Android (\d+\.\d+)/.test(ua)){
        system.android = parseFloat(RegExp.$1);
    }

    // игровые системы
    system.wii = ua.indexOf("Wii") > -1;
    system.ps = /playstation/i.test(ua);

    // возвращение данных
    return {
        engine: engine,
        browser: browser,
        system: system
    };

}();
```

Использование сценария

Как уже отмечалось, распознавание пользовательского агента считается крайним способом распознавания клиента. Всегда следует сначала прибегнуть к распознаванию возможностей и (или) особенностей. Перечислим подходящие ситуации для распознавания пользовательского агента.

- Если возможность или особенность нельзя правильно распознать напрямую. Например, если в браузере реализована функция-заглушка на будущее, проверка ее существования мало чем поможет.
- Если браузер имеет разные возможности на разных платформах. В этом случае может потребоваться идентифицировать платформу.
- Если нужно точно определить браузер с целью мониторинга.

Резюме

Распознавание клиента — один из самых спорных аспектов применения JavaScript. Из-за различий браузеров часто приходится выбирать одну из ветвей кода на основе

возможностей браузера. Есть несколько подходов к распознаванию клиента, но чаще всего используются три.

- **Распознавание возможностей**. Этот подход основан на проверке конкретных возможностей браузера перед их использованием. Например, сценарий может проверять доступность функции перед ее вызовом. Это позволяет не беспокоиться о конкретных типах и версиях браузеров, а довольствоваться сведениями о наличии или отсутствии возможности. Распознавание возможностей не предполагает точную идентификацию браузера или его версии.
- **Распознавание особенностей**. По сути, особенности — это дефекты в реализациях браузеров, такие как возврат затененных свойств в цикле `for-in` в ранних версиях WebKit. Распознавание особенности часто подразумевает выполнение небольшого тестового фрагмента кода. Поскольку этот подход менее эффективен, чем распознавание возможностей, он применяется, только если конкретная особенность может нарушить работу сценария. Распознавание особенности не идентифицирует браузер или его версию.
- **Распознавание пользовательского агента**. Этот подход идентифицирует браузер по его строке пользовательского агента, которая описывает браузер и часто содержит сведения о его типе, версии, а также платформе и операционной системе. Формат строк пользовательского агента неоднократно изменялся, при этом во многие браузеры включались строки, выдающие их за другие браузеры. Распознавание пользовательского агента — нетривиальная задача, особенно если учесть, что Opera может маскировать строку пользовательского агента. Несмотря на это по строке пользовательского агента можно определить визуализатор и платформу, на которой он работает, включая мобильные устройства и игровые системы.

При определении клиента предпочтительнее распознавать возможности, а если это невозможно или бессмысленно — особенности. Распознавание пользовательского агента следует рассматривать как крайнюю меру из-за его зависимости от строки пользовательского агента.

10 Объектная модель документа

- DOM как иерархия узлов
- Работа с узлами разных типов
- Использование DOM с учетом особенностей браузеров

Объектная модель документа (Document Object Model, DOM) — это прикладной программный интерфейс (API) для HTML- и XML-документов. DOM представляет документ как иерархическое дерево узлов, позволяя добавлять, удалять и изменять отдельные части страницы. Основанная первоначально на ранних инновационных технологиях DHTML (Dynamic HTML) от Netscape и Microsoft, DOM обеспечивает по-настоящему кроссплатформенный и не зависящий от языка способ представления страниц и изменения их разметки.

Спецификация DOM Level 1, определяющая базовые запросы и операции со структурой документа, получила статус рекомендации W3C в октябре 1998 года. В этой главе мы рассмотрим применение DOM Level 1 для работы с HTML-страницами в браузере и ее реализацию в JavaScript. Все новые версии Internet Explorer, Firefox, Safari, Chrome и Opera содержат эффективные DOM-реализации.

Все DOM-объекты представлены в Internet Explorer 8 и более ранних версий COM-объектами. Это означает, что они работают не так, как встроенные JavaScript-объекты. Различия между ними рассматриваются в этой главе.

Иерархия узлов

Любой HTML- или XML-документ можно представить с помощью DOM как иерархию узлов. Есть несколько типов узлов, каждый из которых соответствует разным данным и (или) элементам разметки в документе. Каждый тип узлов обладает определенными характеристиками, данными и методами и может быть связан с другими узлами. Эти связи формируют иерархию, которая позволяет изобразить разметку в виде дерева с конкретным узлом в качестве корня. Возьмем для примера следующую HTML-страницу:

```
<html>
    <head>
        <title>Sample Page</title>
    </head>
    <body>
        <p>Hello World!</p>
    </body>
</html>
```

Этому простому HTML-документу соответствует иерархия узлов, показанная на рис. 10.1.

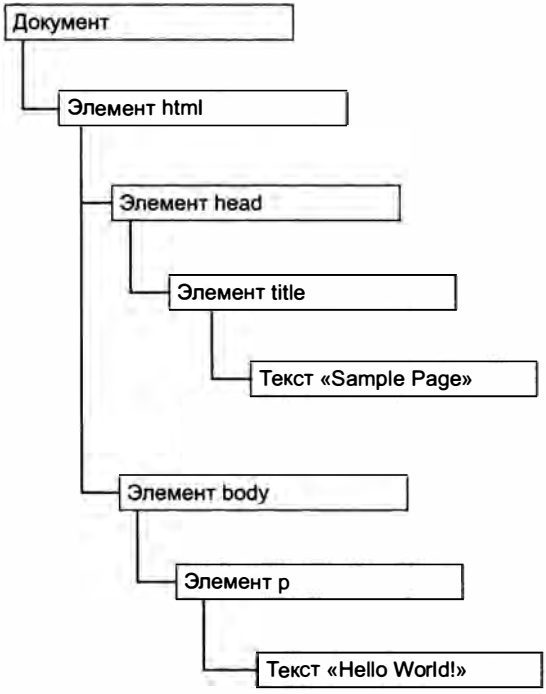

Рис. 10.1

Корнем каждого документа является его узел. В этом примере у документа есть единственный дочерний узел — элемент `<html>`, который называется *элементом документа* (document element). Элемент документа — это самый внешний элемент в документе, содержащий все остальные элементы. В документе может быть только один элемент документа. В HTML-страницах им всегда является элемент `<html>`, а в XML-разметке, где нет предопределенных элементов, им может быть любой элемент.

Всем элементам разметки соответствуют узлы в дереве: HTML-элементам — узлы элементов, атрибутам — узлы атрибутов, типу документа — узел типа документа, а комментариям — узлы комментариев. Всего существует 12 типов узлов, которые наследуются от одного базового типа.

Тип Node

В DOM Level 1 определен интерфейс `Node`, который реализуют все типы DOM-узлов. В JavaScript он представлен типом `Node`, который доступен во всех браузерах, кроме Internet Explorer. Все типы узлов в JavaScript наследуются от `Node`, благодаря чему имеют ряд общих базовых свойств и методов.

У каждого узла есть свойство `nodeType`, указывающее тип узла. Типы узлов определены в типе `Node` с помощью 12 числовых констант:

- `Node.ELEMENT_NODE` (1);
- `Node.ATTRIBUTE_NODE` (2);
- `Node.TEXT_NODE` (3);
- `Node.CDATA_SECTION_NODE` (4);
- `Node.ENTITY_REFERENCE_NODE` (5);
- `Node.ENTITY_NODE` (6);
- `Node.PROCESSING_INSTRUCTION_NODE` (7);
- `Node.COMMENT_NODE` (8);
- `Node.DOCUMENT_NODE` (9);
- `Node.DOCUMENT_TYPE_NODE` (10);
- `Node.DOCUMENT_FRAGMENT_NODE` (11);
- `Node.NOTATION_NODE` (12).

Можно легко определить тип узла, сравнив его с одной из этих констант:

```
if (someNode.nodeType == Node.ELEMENT_NODE){ // не работает в IE до версии 9
    alert("Node is an element.");
}
```

В этом примере свойство someNode.nodeType сравнивается с константой Node.ELEMENT_NODE. Если они равны, значит, someNode является элементом. К сожалению, в Internet Explorer 8 недоступен конструктор типа Node, из-за чего при выполнении этого кода возникнет ошибка. Ради совместимости лучше сравнивать свойство nodeType с числовыми значениями:

```
if (someNode.nodeType == 1){    // работает во всех браузерах
    alert("Node is an element.");
}
```

Не все типы узлов поддерживаются в веб-браузерах. Чаще всего в сценариях используются узлы элементов и текста. Поддержку и применение узлов каждого типа мы рассмотрим позже.

Свойства nodeName и nodeValue

Свойства nodeName и nodeValue предоставляют сведения об узле. Значения этих свойств полностью зависят от типа узла, поэтому рекомендуется всегда проверять его перед их использованием:

```
if (someNode.nodeType == 1){
    value = someNode.nodeName;    // имя тега элемента
}
```

Этот код проверяет, является ли узел элементом. Если да, значение nodeName присваивается переменной. У элементов свойство nodeName всегда содержит имя тега элемента, а nodeValue всегда имеет значение null.

Отношения узлов

Все узлы в документе связаны с другими узлами. Эти связи можно описать в терминах традиционных семейных отношений, как если бы дерево документа было генеалогическим древом. Так, в HTML элемент <body> является дочерним по отношению к элементу <html> и, наоборот, элемент <html> считается родительским для <body>. Элементы <head> и <body> являются одноуровневыми, потому что у них общий непосредственный родитель, элемент <html>.

У каждого узла есть свойство childNodes, содержащее объект NodeList, который похож на массив и используется для хранения упорядоченного списка узлов, доступных по позиции. NodeList не является экземпляром Array, хотя его значения доступны с помощью скобочной нотации и у него имеется свойство length. На самом деле, объекты NodeList — это запросы к DOM-структуре, поэтому ее изменения отражаются в них автоматически. Иначе говоря, NodeList — это динамически обновляемый объект, а не «снимок» узлов на момент первого доступа к нему.

В следующем примере показано, как реализовать доступ к узлам в NodeList с помощью скобочной нотации и метода item():

```
var firstChild = someNode.childNodes[0];
var secondChild = someNode.childNodes.item(1);
var count = someNode.childNodes.length;
```

Хотя поддерживается и скобочная нотация, и метод `item()`, большинство разработчиков предпочитают первый способ из-за его сходства с массивами. Свойство `length` указывает количество узлов в `NodeList` в текущий момент времени. Объект `NodeList` можно преобразовать в массив с помощью метода `Array.prototype.slice()`, что уже было показано для объекта `arguments`. Рассмотрим пример:

```
// не работает в IE8 и более ранних версий
var arrayOfNodes = Array.prototype.slice.call(someNode.childNodes,0);
```

В Internet Explorer 8 и более ранних версий при выполнении этого кода возникает ошибка из-за того, что `NodeList` реализован как COM-объект и не может быть использован вместо JScript-объекта. Чтобы преобразовать `NodeList` в массив в Internet Explorer, необходимо перебрать его элементы вручную. Следующая функция работает во всех браузерах:

```
function convertToArray(nodes){
    var array = null;
    try {
        array = Array.prototype.slice.call(nodes, 0);  // и IE9+, и не IE
    } catch (ex) {
        array = new Array();
        for (var i=0, len=nodes.length; i < len; i++){
            array.push(nodes[i]);
        }
    }

    return array;
}
```

Функция `convertToArray()` сначала пытается создать массив простейшим способом. Если при этом возникает ошибка (то есть используется Internet Explorer до версии 8 включительно), она перехватывается блоком `catch` и массив создается вручную. Это еще один пример распознавания особенностей.

У каждого узла есть свойство `parentNode`, указывающее на его родительский узел в дереве документа. У всех узлов в списке `childNodes` один родитель, так что все их свойства `parentNode` указывают на один и тот же узел. Сами узлы в списке `childNodes` являются одноуровневыми. Переходить от одного узла в этом списке к другому можно с помощью свойств `previousSibling` и `nextSibling`. Свойство `previousSibling` у первого узла в списке и свойство `nextSibling` у последнего узла в списке равны `null`:

```
if (someNode.nextSibling === null){
    alert("Last node in the parent's childNodes list.");
} else if (someNode.previousSibling === null){
    alert("First node in the parent's childNodes list.");
}
```

Если дочерний узел единственный, оба эти свойства равны `null`.

Свойства `firstChild` и `lastChild` родительского узла указывают на первый и последний узлы в его списке `childNodes`. Значение `someNode.firstChild` всегда равно `someNode.childNodes[0]`, а значение `someNode.lastChild` всегда равно `someNode.childNodes[someNode.childNodes.length-1]`. Если дочерний узел только один, свойства `firstChild` и `lastChild` совпадают; если дочерних узлов нет, оба эти свойства равны `null`. С помощью всех описанных отношений можно легко перемещаться между узлами в структуре документа (рис. 10.2).

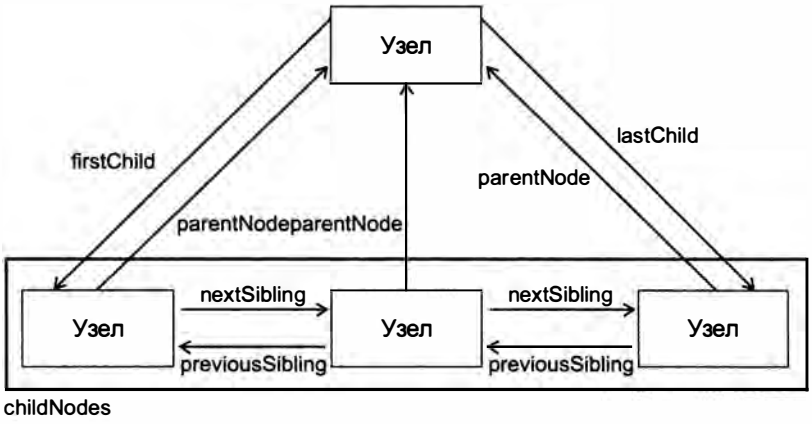

Рис. 10.2

Как видите, можно достигнуть любого узла в дереве документа, просто используя отношения между ними, так что свойство `childNodes` предоставляется скорее ради удобства. В этом смысле на него похож метод `hasChildNodes()`, который возвращает `true`, если у узла есть хотя бы один дочерний узел, и более эффективен, чем запрос свойства `length` списка `childNodes`.

Наконец, у всех узлов есть свойство `ownerDocument`, которое указывает на узел, представляющий весь документ. Узлы принадлежат документу, в котором они были созданы (обычно это документ, где они находятся), и не могут относиться одновременно к двум или более документам. Это свойство обеспечивает быстрый доступ к узлу документа без перебора всех узлов между текущим и корневым.

Не у всех типов узлов могут быть дочерние узлы, хотя все они наследуются от типа Node. Различия типов узлов будут описаны позже.

Манипулирование узлами

Все указатели отношений доступны только для чтения, поэтому для манипулирования узлами применяются методы. Чаще всего используется метод `appendChild()`, который добавляет узел в конец списка `childNodes`. При его вызове обновляются

все указатели отношений в добавленном узле, родительском узле и прежнем последнем дочернем узле в списке childNodes. Метод appendChild() возвращает новый добавленный узел, например:

```
var returnedNode = someNode.appendChild(newNode);
alert(returnedNode == newNode);            // true
alert(someNode.lastChild == newNode);      // true
```

Если узел, переданный в метод appendChild(), уже содержится в документе, он удаляется из предыдущего места и помещается в новое место. Хотя DOM-дерево связано лишь указателями, никакой DOM-узел не может располагаться в нескольких местах документа. Например, если передать в метод appendChild() первый дочерний узел, он станет последним:

```
// предполагается, что у someNode несколько дочерних узлов
var returnedNode = someNode.appendChild(someNode.firstChild);
alert(returnedNode == someNode.firstChild);        // false
alert(returnedNode == someNode.lastChild);         // true
```

Если нужно поместить узел в конкретное место списка childNodes, можно использовать метод insertBefore(), который принимает два аргумента: вставляемый узел и опорный узел. Вставляемый узел добавляется перед опорным и возвращается методом. Если опорный узел равен null, метод insertBefore() работает так же, как appendChild():

```
// вставка узла в качестве последнего дочернего узла
returnedNode = someNode.insertBefore(newNode, null);
alert(newNode == someNode.lastChild);    // true

// вставка в качестве первого дочернего узла
returnedNode = someNode.insertBefore(newNode, someNode.firstChild);
alert(returnedNode == newNode);          // true
alert(newNode == someNode.firstChild);   // true

// вставка узла перед последним дочерним узлом
returnedNode = someNode.insertBefore(newNode, someNode.lastChild);
alert(newNode == someNode.childNodes[someNode.childNodes.length-2]); // true
```

В то время как методы appendChild() и insertBefore() вставляют узлы, не удаляя другие узлы, метод replaceChild() используется для замены узла. В качестве аргументов он принимает вставляемый узел и заменяемый узел. Заменяемый узел полностью удаляется из дерева документа и возвращается из метода, а его место занимает новый узел, например:

```
// замена первого дочернего узла
var returnedNode = someNode.replaceChild(newNode, someNode.firstChild);

// замена последнего дочернего узла
returnedNode = someNode.replaceChild(newNode, someNode.lastChild);
```

Когда узел вставляется с помощью `replaceChild()`, все указатели отношений узла копируются из узла, который он заменяет. Хотя замененный узел технически по-прежнему принадлежит тому же документу, у него больше нет конкретного места в документе.

Удалить узел можно с помощью метода `removeChild()`. Он принимает удаляемый узел как аргумент и возвращает удаленный узел:

```
// удаление первого дочернего узла
var formerFirstChild = someNode.removeChild(someNode.firstChild);

// удаление последнего дочернего узла
var formerLastChild = someNode.removeChild(someNode.lastChild);
```

Как и в случае метода `replaceChild()`, узел, удаленный методом `removeChild()`, все еще принадлежит документу, но не имеет конкретного положения в нем.

Все четыре метода, упомянутые в этом разделе, работают с непосредственными дочерними узлами конкретного узла, то есть для их использования необходимо знать родительский узел (он доступен через свойство `parentNode`). Не у всех типов узлов могут быть дочерние узлы, и попытки использовать эти методы с такими узлами приводят к ошибкам.

Другие методы

Другие два метода есть у узлов всех типов. Первый метод, `cloneNode()`, создает точную копию узла, для которого он вызван. Единственным аргументом `cloneNode()` является логическое значение, указывающее, нужно ли выполнять глубокое копирование. Если оно равно `true`, узел клонируется со всем его поддеревом, в противном случае клонируется только начальный узел. Клонированный узел, который возвращается методом, принадлежит документу, но не имеет родительского узла. Такой узел отсутствует в документе, пока его не добавили с помощью метода `appendChild()`, `insertBefore()` или `replaceChild()`. Рассмотрим, например, следующий HTML-код:

```
<ul>
    <li>item 1</li>
    <li>item 2</li>
    <li>item 3</li>
</ul>
```

Вот примеры двух режимов работы метода `cloneNode()` (предполагается, что переменная `myList` содержит ссылку на элемент `<ul>`):

```
var deepList = myList.cloneNode(true);
alert(deepList.childNodes.length);      // 3 (IE до версии 9)
                                        // или 7 (другие браузеры)

var shallowList = myList.cloneNode(false);
alert(shallowList.childNodes.length);   // 0
```

В этом примере переменной `deepList` присваивается глубокая копия `myList`. Это означает, что `deepList` содержит три элемента списка, каждый из которых включает текст. Переменной `shallowList` назначается поверхностная копия `myList`, так что у нее нет дочерних узлов. Значения `deepList.childNodes.length` различаются в браузерах из-за того, что в Internet Explorer 8 и более ранних версий не создаются узлы для свободного пространства в коде.

Метод cloneNode() не копирует JavaScript-свойства, добавленные в DOM-узлы, такие как обработчики событий. Копируются только атрибуты и, возможно, дочерние узлы, а все остальное теряется. В Internet Explorer есть дефект, из-за которого обработчики событий также клонируются, поэтому рекомендуется удалять их перед клонированием.

Второй метод называется `normalize()` и служит для работы с текстовыми узлами в поддереве документа. В результате синтаксического анализа или манипуляций с DOM можно получить одноуровневые текстовые узлы или текстовые узлы без текста. Метод `normalize()` ищет такие узлы среди потомков узла, для которого он вызван. Пустые текстовые узлы при этом удаляются, а соседние текстовые узлы одного уровня объединяются в один текстовый узел. К этому методу мы еще вернемся.

Тип Document

Узел документа представляется в JavaScript с помощью типа `Document`. В браузерах объект документа является экземпляром типа `HTMLDocument` (производного от `Document`) и соответствует всей HTML-странице. Объект `document` доступен глобально как свойство объекта `window`. Узел `Document` имеет следующие свойства:

- `nodeType` имеет значение 9;
- `nodeName` имеет значение `"#document"`;
- `nodeValue` имеет значение `null`;
- `parentNode` имеет значение `null`;
- `ownerDocument` имеет значение `null`;
- дочерними узлами могут быть объекты `DocumentType` (не более одного), `Element` (не более одного), `ProcessingInstruction` и `Comment`.

Тип `Document` может представлять HTML-страницы или другие XML-документы, но чаще всего он фигурирует в коде как объект `document` типа `HTMLDocument`. Этот объект можно использовать для получения сведений о странице, а также для изменения ее вида и структуры.

Конструктор и прототип типа Document доступны в Firefox, Safari, Chrome и Opera, но не в Internet Explorer до версии 9 включительно. Конструктор и прототип типа HTMLDocument доступны во всех браузерах, в том числе в Internet Explorer начиная с версии 8.

Дочерние узлы узла Document

Хотя в спецификации DOM сказано, что дочерними узлами у `Document` могут быть узлы `DocumentType`, `Element`, `ProcessingInstruction` и `Comment`, среди них есть также два встроенных ярлыка. Первым является свойство `documentElement`, которое всегда указывает на элемент `<html>` в HTML-странице. Элемент `document` также всегда есть в списке `childNodes`, но свойство `documentElement` предоставляет более быстрый и прямой доступ к элементу `<html>`. Рассмотрим следующую простую страницу:

```
<html>
    <body>

    </body>
</html>
```

Когда эта страница обрабатывается браузером, у документа есть только один дочерний узел, элемент `<html>`. Он доступен с помощью свойства `documentElement` и списка `childNodes`:

```
var html = document.documentElement;     // получение ссылки на элемент <html>
alert(html === document.childNodes[0]);  // true
alert(html === document.firstChild);     // true
```

Этот пример показывает, что значения `documentElement`, `firstChild` и `childNodes[0]` одинаковы: все три указывают на элемент `<html>`.

Как экземпляр `HTMLDocument`, объект `document` также имеет свойство `body`, указывающее на элемент `<body>`. Неудивительно, что оно используется в JavaScript очень часто:

```
var body = document.body;        // получение ссылки на элемент <body>
```

Свойства `document.documentElement` и `document.body` поддерживаются во всех основных браузерах.

Дочерним узлом у `Document` может быть также узел `DocumentType`. Тег `<!DOCTYPE>` считается самостоятельной сущностью, отдельной от других частей документа и доступной как свойство `doctype` (`document.doctype` в браузерах):

```
var doctype = document.doctype;  // получение ссылки на элемент <!DOCTYPE>
```

Поддержка свойства `document.doctype` зависит от браузера.

- **Internet Explorer 8 и более ранних версий**. Если тип документа указан, он неправильно интерпретируется как комментарий и обрабатывается как узел `Comment`. Свойство `document.doctype` всегда имеет значение `null`.
- **Internet Explorer 9+ и Firefox**. Если тип документа указан, он становится первым дочерним узлом документа. Свойством `document.doctype` является узел

DocumentType, который также доступен как document.firstChild или document.childNodes[0].

- **Safari, Chrome и Opera**. Если тип документа указан, он обрабатывается, но не считается дочерним узлом документа. Свойством document.doctype является узел DocumentType, но он отсутствует в списке document.childNodes.

Из-за несогласованной поддержки свойства document.doctype пользы от него немного.

Комментарии вне элемента <html> технически являются дочерними узлами документа, но браузеры распознают и представляют их по-разному. Рассмотрим следующую HTML-страницу:

```
<!-- первый комментарий -->
<html>
    <body>

    </body>
</html>
<!-- второй комментарий -->
```

Казалось бы, у этой страницы три дочерних узла: комментарий, элемент <html> и еще один комментарий. Можно было бы ожидать, что и список document.childNodes будет содержать три элемента, но на практике браузеры обрабатывают комментарии вне элемента <html> совершенно по-разному.

- Internet Explorer 8 и более ранних версий, Safari 3.1 и более поздних версий, Opera и Chrome создают узел комментария для первого комментария, но не для второго. Первый комментарий становится первым узлом в списке document.childNodes.
- В Internet Explorer 9 и более поздних версий для первого комментария создается узел комментария в списке document.childNodes, а для второго — в списке document.body.childNodes.
- Firefox и Safari до версии 3.1 игнорируют оба комментария.

Опять же, из-за несогласованной реализации доступ к комментариям вне элемента <html> практически бесполезен.

Как правило, методы appendChild(), removeChild() и replaceChild() не используются с объектом document, поскольку тип документа (если он есть) доступен только для чтения, а дочерний элемент у документа может быть только один и уже существует.

Сведения о документе

В качестве экземпляра типа HTMLDocument объект document имеет несколько свойств, которых нет у стандартных объектов Document. Эти свойства предоставляют сведения о загруженной веб-странице. Первое из них, title, содержит текст элемента <title>, отображаемый в заголовке или на вкладке окна браузера. С помощью этого

свойства можно получить заголовок текущей страницы или изменить его, при этом элемент <title> не изменяется, например:

```
// получение заголовка документа
var originalTitle = document.title;

// задание заголовка документа
document.title = "New page title"; // новый заголовок страницы
```

Следующие три свойства связаны с запросами веб-страниц. Свойство URL содержит полный URL-адрес страницы (который отображается в адресной строке), свойство domain — только доменное имя страницы, а referrer — URL-адрес страницы, с которой был выполнен переход на текущую страницу. Свойство referrer может быть пустой строкой, если текущая страница была открыта без ссылки. Все эти сведения есть в HTTP-заголовке запроса, а указанные свойства просто предоставляют доступ к ним в JavaScript:

```
// получение полного URL-адреса
var url = document.URL;

// получение домена
var domain = document.domain;

// получение источника ссылки
var referrer = document.referrer;
```

Свойства URL и domain связаны. Например, если свойство document.URL содержит адрес http://www.wrox.com/WileyCDA/, значением document.domain будет www.wrox.com.

Из этих трех свойств задать можно только свойство domain, при этом действуют некоторые ограничения, связанные с безопасностью. Если URL-адрес содержит поддомен, например p2p.wrox.com, значением domain может быть только "wrox.com" (то же верно, если URL-адрес содержит префикс "www", например www.wrox.com). Этому свойству нельзя назначить домен, который не содержится в URL-адресе:

```
// страница с сайта p2p.wrox.com

document.domain = "wrox.com";          // успех

document.domain = "nczonline.net";     // ошибка!
```

Возможность задать свойство document.domain полезна, если на странице есть обычный фрейм или встроенный фрейм (iframe) из другого поддомена. Страницы из разных поддоменов не могут взаимодействовать с помощью JavaScript из-за ограничений безопасности. Если присвоить свойству document.domain на всех страницах одно и то же значение, им станут доступны JavaScript-объекты других страниц. Например, если на странице, загруженной с адреса www.wrox.com, есть встроенный фрейм со страницей p2p.wrox.com, их значения document.domain будут разными, из-за чего JavaScript-объекты внешней страницы будут недоступны внутренней

странице, и наоборот. Если у обеих страниц свойство `document.domain` будет иметь значение `"wrox.com"`, страницы смогут взаимодействовать.

Другое ограничение запрещает детализацию свойства домена после того, как ему было назначено более общее значение. Это означает, например, что нельзя сначала присвоить свойству `document.domain` значение `"wrox.com"`, а затем восстановить значение `"p2p.wrox.com"`. Попытка сделать это приведет к ошибке:

```
// страница с сайта p2p.wrox.com

document.domain = "wrox.com";        // обобщение - успех

document.domain = "p2p.wrox.com";    // детализация - ошибка!
```

Это ограничение действует во всех браузерах, но в Internet Explorer — только начиная с версии 8.

Получение элементов

Вероятно, самой востребованной DOM-операцией является получение ссылки на конкретный элемент или множество элементов для выполнения каких-либо действий с ними. Для этого тип `Document` предоставляет методы `getElementById()` и `getElementsByTagName()`.

Метод `getElementById()` принимает идентификатор элемента, который нужно получить, и возвращает этот элемент или `null`, если его не существует. Сравнение идентификатора с атрибутом `id` элемента на странице выполняется с учетом регистра. Возьмем для примера следующий элемент:

```
<div id="myDiv">Some text</div>
```

Этот элемент можно получить так:

```
var div = document.getElementById("myDiv");    // получение ссылки на <div>
```

Однако следующий код возвращает `null` во всех браузерах, кроме Internet Explorer 7 и более ранних версий:

```
var div = document.getElementById("mydiv");  // работает только в IE
                                             // до версии 7 включительно
```

В Internet Explorer до версии 8 идентификаторы обрабатываются без учета регистра, так что `"myDiv"` и `"mydiv"` считаются одинаковыми значениями.

Если страница содержит несколько элементов с одним идентификатором, метод `getElementById()` возвращает первый из них. В Internet Explorer 7 и более ранних версий он также возвращает элементы форм (`<input>`, `<textarea>`, `<button>` и `<select>`), у которых атрибут `name` соответствует указанному идентификатору. Если один из этих элементов с совпадающим атрибутом `name` располагается в документе раньше

элемента с указанным идентификатором, Internet Explorer возвращает элемент формы, например:

```
<input type="text" name="myElement" value="Text field">
<div id="myElement">A div</div>
```

В Internet Explorer 7 метод `document.getElementById()` возвращает для этого HTML-кода ссылку на элемент `<input>`, а в других браузерах — на элемент `<div>`. Для предотвращения этой проблемы в Internet Explorer следует проверять, нет ли у полей форм атрибутов `name`, совпадающих с идентификаторами других элементов.

Для получения ссылок на элементы также часто используется метод `getElementsByTagName()`. Он принимает имя тега элементов, которые нужно получить, и возвращает объект `NodeList`, содержащий эти элементы. В HTML-документах этот метод возвращает объект `HTMLCollection`, который очень похож на `NodeList` тем, что тоже обновляется динамически. Например, следующий код возвращает объект `HTMLCollection`, содержащий все элементы `<img>` на странице:

```
var images = document.getElementsByTagName("img");
```

Этот код сохраняет объект `HTMLCollection` в переменной `images`. Подобно `NodeList`, элементы объекта `HTMLCollection` доступны с помощью скобочной нотации и метода `item()`, а количество элементов можно получить с помощью свойства `length`:

```
alert(images.length);          // количество изображений
alert(images[0].src);          // атрибут src первого изображения
alert(images.item(0).src);     // атрибут src первого изображения
```

У объекта `HTMLCollection` есть дополнительный метод `namedItem()`, позволяющий обращаться к элементам в коллекции по атрибуту `name`. Предположим, на странице есть такой элемент `<img>`:

```
<img src="myimage.gif" name="myImage">
```

Получить ссылку на этот элемент `<img>` из переменной `images` можно следующим образом:

```
var myImage = images.namedItem("myImage");
```

Таким образом, элементы в объекте `HTMLCollection` доступны по индексу и по имени, что позволяет легко извлекать их. Именованные элементы также доступны с помощью скобочной нотации:

```
var myImage = images["myImage"];
```

Как видите, скобочную нотацию можно использовать и с числовыми, и со строковыми индексами, при этом в первом случае неявно вызывается метод `item()`, а во втором — `namedItem()`.

Чтобы получить все элементы в документе, передайте в метод `getElementsByTagName()` звездочку (`*`). Как правило, в JavaScript и CSS (Cascading Style Sheets) звездочка означает «все». Вот пример:

```
var allElements = document.getElementsByTagName("*");
```

Этот код возвращает объект `HTMLCollection`, содержащий все элементы в исходном порядке, то есть первым элементом является `<html>`, вторым — `<head>`, и т. д. В Internet Explorer 8 и более ранних версий комментарии также считаются элементами, из-за чего вызов `getElementsByTagName("*")` возвращает узлы комментариев. В Internet Explorer 9 комментарии не считаются элементами и не возвращаются.

Хотя в спецификации сказано, что имена тегов чувствительны к регистру, метод getElementsByTagName() не учитывает регистр для совместимости с существующими HTML-страницами. При работе с XML-страницами (включая XHTML) метод getElementsByTagName() переключается в режим, чувствительный к регистру.

Третий метод, `getElementsByName()`, доступен только для типа `HTMLDocument`. Как можно догадаться, он возвращает все элементы с конкретным значением атрибута `name`. Чаще всего он используется с переключателями, которые должны иметь одно имя на всех, чтобы серверу отправлялось правильное значение:

```
<fieldset>
    <legend>Which color do you prefer?</legend>
    <ul>
        <li><input type="radio" value="red" name="color" id="colorRed">
            <label for="colorRed">Red</label></li>
        <li><input type="radio" value="green" name="color" id="colorGreen">
            <label for="colorGreen">Green</label></li>
        <li><input type="radio" value="blue" name="color" id="colorBlue">
            <label for="colorBlue">Blue</label></li>
    </ul>
</fieldset>
```

Здесь у всех кнопок-переключателей атрибут `name` имеет значение `"color"`, хотя их идентификаторы различны. Идентификаторы позволяют применить элементы `<label>` к переключателям, а атрибут `name` гарантирует, что серверу будет отправлено только одно значение из трех. Эти переключатели можно получить следующим образом:

```
var radios = document.getElementsByName("color");
```

Как и `getElementsByTagName()`, метод `getElementsByName()` возвращает объект `HTMLCollection`, но теперь метод `namedItem()` этого объекта всегда возвращает первый элемент (потому что все элементы имеют одно имя).

Специальные коллекции

У объекта document есть несколько специальных коллекций. Каждая из них является объектом HTMLCollection и предоставляет быстрый доступ к часто используемым частям документа:

- document.anchors — содержит все элементы <a> с атрибутом name;
- document.applets — содержит все элементы <applet> (эта коллекция устарела, потому что использовать элемент <applet> больше не рекомендуется);
- document.forms — содержит все элементы <form> (то же, что document.getElementsByTagName("form"));
- document.images — содержит все элементы <img> (то же, что document.getElementsByTagName("img"));
- document.links — содержит все элементы <a> с атрибутом href.

Эти специальные коллекции доступны в любой момент и, как все объекты HTMLCollection, динамически обновляются в соответствии с содержимым текущего документа.

Соответствие спецификации DOM

DOM включает много частей и уровней, поэтому иногда требуется точно определить, какие DOM-компоненты реализованы в браузере. Сведения об этом содержит свойство document.implementation. В DOM Level 1 определен только один метод объекта document.implementation, а именно hasFeature(). Он принимает два аргумента: имя и версию DOM-компонента, который нужно проверить. Если браузер поддерживает запрошенный компонент указанной версии, метод возвращает true, например:

```
var hasXmlDom = document.implementation.hasFeature("XML", "1.0");
```

Значения, которые можно проверить, перечислены в таблице.

Компонент	Поддерживаемые версии	Описание
Core	1.0, 2.0, 3.0	Основные возможности DOM, регламентирующие представление документа в виде иерархического дерева
XML	1.0, 2.0, 3.0	Расширение компонента Core для работы с XML, обеспечивающее поддержку разделов CDATA, инструкций по обработке и сущностей
HTML	1.0, 2.0	Расширение компонента XML для работы с HTML, обеспечивающее поддержку элементов и сущностей, специфичных для HTML

Продолжение

Компонент	Поддерживаемые версии	Описание
Views	2.0	Компонент для форматирования документа на основе стилей
StyleSheets	2.0	Компонент, сопоставляющий таблицы стилей с документами
CSS	2.0	Поддержка CSS Level 1
CSS2	2.0	Поддержка CSS Level 2
Events	2.0, 3.0	Универсальные DOM-события
UIEvents	2.0, 3.0	События пользовательского интерфейса
MouseEvents	2.0, 3.0	События мыши (щелчок, наведение и т. д.)
MutationEvents	2.0, 3.0	События, генерируемые при изменении DOM-дерева
HTMLEvents	2.0	События HTML 4.01
Range	2.0	Объекты и методы для манипулирования диапазоном в DOM-дереве
Traversal	2.0	Методы для обхода DOM-дерева
LS	3.0	Компонент для синхронной загрузки и сохранения DOM-дерева
LS-Async	3.0	Компонент для асинхронной загрузки и сохранения DOM-дерева
Validation	3.0	Методы для изменения DOM-дерева с соблюдением правил

Недостаток метода `hasFeature()` в том, что разработчики сами решают, соответствует ли реализация конкретного компонента спецификации DOM. Если он возвращает `true`, это вовсе не означает, что соблюдены все требования спецификации. Например, Safari 2.x и более ранних версий возвращает `true` для нескольких компонентов, которые реализованы не полностью. Прежде чем использовать те или иные части DOM, имеет смысл не только вызвать метод `hasFeature()`, но и распознать требуемые возможности.

Запись документа

С помощью объекта `document` можно добавлять данные в поток вывода веб-страницы, используя методы `write()`, `writeln()`, `open()` и `close()`. Методы `write()` и `writeln()` принимают в качестве аргумента строку, которую нужно добавить в поток вывода. Метод `write()` добавляет ее как есть, а метод `writeln()` дополняет текст знаком перевода строки (`\n`). Эти два метода можно использовать во время загрузки страницы для динамического добавления контента, например:

Листинг DocumentWriteExample01.htm


```
<html>
<head>
    <title>document.write() Example</title>
</head>
<body>
    <p>The current date and time is:
    <script type="text/javascript">
        document.write("<strong>" + (new Date()).toString() + "</strong>");
    </script>
    </p>
</body>
</html>
```

Этот код выводит при загрузке страницы текущие дату и время. Дата заключается в элемент `<strong>`, который обрабатывается так же, как если бы он содержался в HTML-коде страницы. Это означает, что для него создается DOM-элемент, к которому позднее можно получать доступ. Так обрабатывается любой HTML-код, выводимый с помощью метода `write()` или `writeln()`.

Методы `write()` и `writeln()` часто используются для динамического включения внешних ресурсов, таких как JavaScript-файлы. При включении JavaScript-файлов необходимо проследить за тем, чтобы не добавить строку `"</script>"`, потому что она будет интерпретирована как конец сценария, из-за чего остальной код не выполнится, например:

Листинг DocumentWriteExample02.htm


```
<html>
<head>
    <title>document.write() Example</title>
</head>
<body>
    <script type="text/javascript">
        document.write("<script type=\"text/javascript\" src=\"file.js\">" +
            "</script>");
</script>
</body>
</html>
```

Хотя этот файл выглядит правильно, тег `"</script>"` в аргументе метода `write()` интерпретируется как конец главного сценария, из-за чего на странице выводятся знаки `");`. Чтобы исправить ситуацию, нужно изменить строку, как было указано в главе 2:

Листинг DocumentWriteExample03.htm

```
<html>
<head>
    <title>Document.write() Example</title>
</head>
<body>
    <script type="text/javascript">
```

```
        document.write("<script type=\"text/javascript\" src=\"file.js\">" +
            "<\/script>");
    </script>
</body>
</html>
```

Теперь строка `"<\/script>"` не интерпретируется как закрывающий тег внешнего тега `<script>` и лишний контент на странице не выводится.

В предыдущих примерах метод `document.write()` использовался для вывода контента непосредственно на странице во время ее визуализации. Если вызвать метод `document.write()` после завершения загрузки страницы, она будет перезаписана, например:

Листинг DocumentWriteExample04.htm

```
<html>
<head>
    <title>Document.write() Example</title>
</head>
<body>
    <p> This is some content that you won't get to see because it will be
overwritten.</p>
    <script type="text/javascript">
        window.onload = function(){
            document.write("Hello World!");
        };
    </script>
</body>
</html>
```

Чтобы отложить выполнение функции до полной загрузки страницы, здесь используется обработчик события `window.onload` (события обсуждаются в главе 13). По завершении загрузки весь контент страницы перезаписывается строкой `"Hello World!"`.

Методы `open()` и `close()` открывают и закрывают поток вывода веб-страницы соответственно. При использовании методов `write()` и `writeln()` во время загрузки страницы это не требуется.

Запись в XHTML-документы строгой версии не поддерживается. Со страницами, для которых указан тип контента application/xml+xhtml, эти методы работать не будут.

Тип Element

После типа `Document` чаще всего в веб-программировании используется тип `Element`. Он представляет XML- или HTML-элемент, обеспечивая доступ к его сведениям, таким как имя тега, дочерние элементы и атрибуты. Узел `Element` имеет следующие свойства:

- `nodeType` имеет значение 1;
- `nodeName` содержит имя тега элемента;

- nodeValue имеет значение null;
- parentNode может указывать на узел Document или Element;
- дочерними узлами могут быть объекты Element, Text, Comment, ProcessingInstruction, CDATASection и EntityReference.

Имя тега доступно как свойство nodeName или tagName; оба свойства возвращают одно и то же значение (ради ясности обычно используется tagName). Возьмем для примера следующий элемент:

```
<div id="myDiv"></div>
```

Получить этот элемент и его имя тега можно так:

```
var div = document.getElementById("myDiv");
alert(div.tagName);         // "DIV"
alert(div.tagName == div.nodeName);    // true
```

Элемент имеет имя тега div и идентификатор "myDiv", но div.tagName на самом деле возвращает "DIV", а не "div". При работе с HTML имена тегов всегда представляются в верхнем регистре, а с XML (включая XHTML) всегда используется исходный регистр. Если вы не знаете, в каком документе — HTML или XML — будет использоваться ваш сценарий, имеет смысл унифицировать регистр имен тегов перед сравнением:

```
if (element.tagName == "div"){    // НЕ ДЕЛАЙТЕ ТАК! Возможны ошибки!
    // какие-то действия
}

if (element.tagName.toLowerCase() == "div"){    // Лучше - работает везде
    // какие-то действия
}
```

Этот пример демонстрирует два варианта сравнения имени тега со свойством tagName. Первый менее надежен, потому что он не работает в HTML-документах. Второй подход с преобразованием имени тега в нижний регистр работает и в XML-, и в HTML-документах.

Конструктор и прототип типа Element доступны во всех современных браузерах, в том числе в Internet Explorer начиная с версии 8. В более старых браузерах, таких как Safari до версии 2 и Opera до версии 8, конструктор типа Element недоступен.

Элементы HTML

Все HTML-элементы представляются объектами типа HTMLElement или его подтипов. Тип HTMLElement унаследован непосредственно от Element и содержит несколько дополнительных свойств. Каждое свойство соответствует одному из следующих стандартных атрибутов, доступных у каждого HTML-элемента:

- `id` — уникальный идентификатор элемента в документе;
- `title` — дополнительные сведения об элементе, обычно отображаемые во всплывающей подсказке;
- `lang` — язык содержимого элемента (используется редко);
- `dir` — направление языка (`"ltr"` — слева направо или `"rtl"` — справа налево; также используется редко);
- `className` — эквивалент атрибута `class`, который указывает для элемента CSS-класс. Это свойство нельзя было назвать `class`, потому что `class` — зарезервированное слово в ECMAScript (см. главу 1).

Каждое из этих свойств можно использовать для получения и изменения значения соответствующего атрибута. Рассмотрим следующий HTML-элемент:

Листинг HTMLElementsExample01.htm

```
<div id="myDiv" class="bd" title="Body text" lang="en" dir="ltr"></div>
```

Все атрибуты этого элемента можно получить следующим образом:

```
var div = document.getElementById("myDiv");
alert(div.id);              // "myDiv"
alert(div.className);       // "bd"
alert(div.title);           // "Body text"
alert(div.lang);            // "en"
alert(div.dir);             // "ltr"
```

Также можно изменить атрибуты, присвоив свойствам новые значения:

Листинг HTMLElementsExample01.htm

```
div.id = "someOtherId";
div.className = "ft";
div.title = "Some other text";
div.lang = "fr";
div.dir = "rtl";
```

Не все свойства при перезаписи изменяют вид страницы. Изменения свойств `id` и `lang` незаметны для пользователя (если они не используются в стилях CSS), а изменение свойства `title` проявляется только при наведении указателя мыши на элемент. Свойство `dir` переключает способ выравнивания текста (по левому или правому краю), а изменения `className` видны, если новый стиль CSS отличается от прежнего.

Как уже отмечалось, каждый HTML-элемент представляется экземпляром типа `HTMLElement` или более специфичного подтипа. Доступные элементы и соответствующие им типы указаны в таблице (элементы, выделенные курсивом, устарели). Имейте в виду, что в JavaScript-сценариях эти типы доступны в Opera, Safari, Chrome и Firefox, но не в Internet Explorer до версии 8.

Элемент	Тип	Элемент	Тип
A	HTMLAnchorElement	INPUT	HTMLInputElement
ABBR	HTMLElement	INS	HTMLModElement
ACRONYM	HTMLElement	*ISINDEX*	*HTMLIsIndexElement*
ADDRESS	HTMLElement	KBD	HTMLElement
APPLET	*HTMLAppletElement*	LABEL	HTMLLabelElement
AREA	HTMLAreaElement	LEGEND	HTMLLegendElement
B	HTMLElement	LI	HTMLLIElement
BASE	HTMLBaseElement	LINK	HTMLLinkElement
BASEFONT	*HTMLBaseFontElement*	MAP	HTMLMapElement
BDO	HTMLElement	*MENU*	*HTMLMenuElement*
BIG	HTMLElement	META	HTMLMetaElement
BLOCKQUOTE	HTMLQuoteElement	NOFRAMES	HTMLElement
BODY	HTMLBodyElement	NOSCRIPT	HTMLElement
BR	HTMLBRElement	OBJECT	HTMLObjectElement
BUTTON	HTMLButtonElement	OL	HTMLOListElement
CAPTION	HTMLTableCaptionElement	OPTGROUP	HTMLOptGroupElement
CENTER	*HTMLElement*	OPTION	HTMLOptionElement
CITE	HTMLElement	P	HTMLParagraphElement
CODE	HTMLElement	PARAM	HTMLParamElement
COL	HTMLTableColElement	PRE	HTMLPreElement
COLGROUP	HTMLTableColElement	Q	HTMLQuoteElement
DD	HTMLElement	*S*	*HTMLElement*
DEL	HTMLModElement	SAMP	HTMLElement
DFN	HTMLElement	SCRIPT	HTMLScriptElement
DIR	*HTMLDirectoryElement*	SELECT	HTMLSelectElement
DIV	HTMLDivElement	SMALL	HTMLElement
DL	HTMLDListElement	SPAN	HTMLElement
DT	HTMLElement	*STRIKE*	*HTMLElement*
EM	HTMLElement	STRONG	HTMLElement
FIELDSET	HTMLFieldSetElement	STYLE	HTMLStyleElement
FONT	*HTMLFontElement*	SUB	HTMLElement
FORM	HTMLFormElement	SUP	HTMLElement
FRAME	HTMLFrameElement	TABLE	HTMLTableElement
FRAMESET	HTMLFrameSetElement	TBODY	HTMLTableSectionElement

Продолжение

Элемент	Тип	Элемент	Тип
H1	HTMLHeadingElement	TD	HTMLTableCellElement
H2	HTMLHeadingElement	TEXTAREA	HTMLTextAreaElement
H3	HTMLHeadingElement	TFOOT	HTMLTableSectionElement
H4	HTMLHeadingElement	TH	HTMLTableCellElement
H5	HTMLHeadingElement	THEAD	HTMLTableSectionElement
H6	HTMLHeadingElement	TITLE	HTMLTitleElement
HEAD	HTMLHeadElement	TR	HTMLTableRowElement
HR	HTMLHRElement	TT	HTMLElement
HTML	HTMLHtmlElement	*U*	*HTMLElement*
I	HTMLElement	UL	HTMLUListElement
IFRAME	HTMLIFrameElement	VAR	HTMLElement
IMG	HTMLImageElement		

У каждого из этих типов есть атрибуты и методы. Многие из типов обсуждаются в этой книге.

Получение атрибутов

У каждого элемента могут быть атрибуты, которые обычно предоставляют дополнительные сведения о нем или его содержимом. Три основных DOM-метода для работы с атрибутами — `getAttribute()`, `setAttribute()` и `removeAttribute()`. Они предназначены для работы с любыми атрибутами, включая те, которые определены как свойства типа `HTMLElement`, например:

```
var div = document.getElementById("myDiv");
alert(div.getAttribute("id"));        // "myDiv"
alert(div.getAttribute("class"));     // "bd"
alert(div.getAttribute("title"));     // "Body text"
alert(div.getAttribute("lang"));      // "en"
alert(div.getAttribute("dir"));       // "ltr"
```

В метод `getAttribute()` передается фактическое имя атрибута, так что для получения значения атрибута `class` используется имя `"class"` (а не `"className"`, которое необходимо при доступе к атрибуту через свойство объекта). Если атрибут с указанным именем не существует, метод `getAttribute()` всегда возвращает `null`.

С помощью метода `getAttribute()` можно также получать значения пользовательских атрибутов, отсутствующих в языке HTML. Рассмотрим следующий элемент:

```
<div id="myDiv" my_special_attribute="hello!"></div>
```

В этом элементе определен пользовательский атрибут `my_special_attribute` со значением `"hello!"`, которое можно получить с помощью метода `getAttribute()`, как и любое другое значение:

```
var value = div.getAttribute("my_special_attribute");
```

Имена атрибутов нечувствительны к регистру, так что `"ID"` и `"id"` считаются одним и тем же атрибутом. Имейте также в виду, что согласно спецификации HTML5 для успешного прохождения проверки к пользовательским атрибутам нужно добавлять префикс `data-`.

Все атрибуты элемента также доступны как свойства объекта DOM-элемента. Мы уже обсудили пять свойств объекта `HTMLElement`, которые непосредственно соответствуют атрибутам, но к нему добавляются и все остальные общепризнанные (непользовательские) атрибуты. Рассмотрим следующий элемент:

```
<div id="myDiv" align="left" my_special_attribute="hello"></div>
```

Поскольку `id` и `align` являются в HTML общепризнанными атрибутами элемента `<div>`, они будут представлены свойствами объекта элемента, а пользовательский атрибут `my_special_attribute` не будет. Исключение — Internet Explorer до версии 8 включительно, где также создаются свойства и для пользовательских атрибутов, например:

Листинг ElementAttributesExample02.htm


```
alert(div.id);                      // "myDiv"
alert(div.my_special_attribute);    // undefined (за исключением IE)
alert(div.align);                   // "left"
```

У двух категорий атрибутов их значения в виде свойств не соответствуют значениям, которые возвращает метод `getAttribute()`. Первый — это атрибут `style`, который используется для указания CSS-стиля элемента. При доступе с помощью метода `getAttribute()` атрибут `style` содержит CSS-текст, тогда как при доступе к нему с помощью свойства возвращается объект. Свойство `style` используется для программного доступа к стилю элемента (см. главу 12), поэтому оно не соответствует атрибуту `style`.

Вторая категория атрибутов, которые работают иначе, — это атрибуты обработчиков событий, такие как `onclick`. При использовании с элементом атрибут `onclick` содержит JavaScript-код, который в виде строки возвращается методом `getAttribute()`. Однако при доступе к свойству `onclick` оно возвращает JavaScript-функцию (или `null`, если атрибут не указан). Свойство `onclick` и другие свойства обработки событий реализованы так, чтобы им можно было назначать функции.

Из-за этих различий программисты часто пренебрегают методом `getAttribute()` при работе с DOM в JavaScript, используя вместо него исключительно свойства объектов. А метод `getAttribute()` применяется преимущественно для получения значений пользовательских атрибутов.

В Internet Explorer до версии 8 метод getAttribute() при вызове для атрибута style и атрибутов обработки событий, таких как onclick, всегда возвращает одно и то же значение, как если бы доступ к ним осуществлялся с помощью свойства. Так, вызов getAttribute(«style») возвращает объект, а getAttribute(«onclick») — функцию. Хотя в Internet Explorer 8 это несоответствие исправлено, оно лишний раз убеждает не использовать метод getAttribute() для получения HTML-атрибутов.

Задание атрибутов

У метода `getAttribute()` есть обратный метод `setAttribute()`, принимающий два аргумента: имя атрибута, который нужно задать, и его значение. Если атрибут уже существует, метод `setAttribute()` обновляет его значение. Если атрибут не существует, метод `setAttribute()` создает его и присваивает ему значение, например:

Листинг ElementAttributesExample01.htm


```
div.setAttribute("id", "someOtherId");
div.setAttribute("class", "ft");
div.setAttribute("title", "Some other text");
div.setAttribute("lang","fr");
div.setAttribute("dir", "rtl");
```

Метод `setAttribute()` работает с HTML-атрибутами и пользовательскими атрибутами одинаково. При его вызове имена атрибутов преобразуются в нижний регистр (например, `"ID"` преобразуется в `"id"`).

Задавать значения атрибутов можно и с помощью соответствующих свойств:

```
div.id = "someOtherId";
div.align = "left";
```

Имейте в виду, что добавление пользовательского свойства к DOM-элементу не делает его автоматически атрибутом элемента:

```
div.mycolor = "red";
alert(div.getAttribute("mycolor"));    // null (кроме Internet Explorer)
```

Этот код добавляет к элементу пользовательское свойство `mycolor` со значением `"red"`. В большинстве браузеров оно не становится автоматически атрибутом элемента, поэтому вызов метода `getAttribute()` с целью получения одноименного атрибута возвращает `null`. В Internet Explorer пользовательские свойства считаются атрибутами элемента, и наоборот.

В Internet Explorer 7 и более ранних версий метод setAttribute() работает неправильно. Попытка задать с его помощью атрибут class или style либо свойство обработчика события игнорируется. Хотя эти проблемы были устранены в Internet Explorer 8, лучше всегда задавать эти атрибуты с помощью свойств.

Метод `removeAttribute()` полностью удаляет атрибут элемента, не ограничиваясь очисткой его значения:

```
div.removeAttribute("class");
```

Этот метод используется не очень часто, но может быть полезен для отбора нужных атрибутов при сериализации DOM-элемента.

В Internet Explorer 6 и более ранних версий метод removeAttribute() не поддерживается.

Свойство attributes

Тип `Element` является единственным типом DOM-узла, у которого есть свойство `attributes`. Оно содержит коллекцию `NamedNodeMap`, динамически обновляемую подобно `NodeList`. Каждый атрибут элемента представляется в объекте `NamedNodeMap` узлом `Attr`. У объекта `NamedNodeMap` есть следующие методы:

- `getNamedItem(`*имя*`)` — возвращает узел, у которого свойство `nodeName` равно указанному имени;
- `removeNamedItem(`*имя*`)` — удаляет из списка узел, у которого свойство `nodeName` равно указанному имени;
- `setNamedItem(`*узел*`)` — добавляет узел в список, индексируя его по свойству `nodeName`;
- `item(`*позиция*`)` — возвращает узел в указанной позиции.

Каждым элементом свойства `attributes` является узел, у которого свойство `nodeName` содержит имя атрибута, а `nodeValue` — значение атрибута. Например, получить значение атрибута `id` элемента можно следующим образом:

```
var id = element.attributes.getNamedItem("id").nodeValue;
```

То же самое можно сделать, используя скобочную нотацию:

```
var id = element.attributes["id"].nodeValue;
```

С помощью скобочной нотации можно также задавать значения атрибутов:

```
element.attributes["id"].nodeValue = "someOtherId";
```

Метод `removeNamedItem()` работает так же, как и метод `removeAttribute()` элемента: он просто удаляет атрибут с указанным именем. Единственное его отличие в том, что он возвращает узел `Attr`, который представляет атрибут:

```
var oldAttr = element.attributes.removeNamedItem("id");
```

Метод `setNamedItem()` позволяет добавить новый атрибут к элементу, но используется он редко. В качестве параметра он принимает добавляемый узел атрибута:

```
element.attributes.setNamedItem(newAttr);
```

Вообще говоря, вместо этих методов атрибутов лучше использовать методы `getAttribute()`, `removeAttribute()` и `setAttribute()`, потому что они проще.

Свойство `attributes` полезно, если нужно перебрать атрибуты элемента. Чаще всего это требуется при сериализации DOM-структуры в XML- или HTML-строку. Следующий код перебирает все атрибуты элемента и составляет строку формата *имя="значение" имя="значение"*:

Листинг ElementAttributesExample03.htm


```
function outputAttributes(element){
    var pairs = new Array(),
        attrName,
        attrValue,
        i,
        len;

    for (i=0, len=element.attributes.length; i < len; i++){
        attrName = element.attributes[i].nodeName;
        attrValue = element.attributes[i].nodeValue;
        pairs.push(attrName + "=\"" + attrValue + "\"");
    }
    return pairs.join(" ");
}
```

Эта функция сохраняет пары имен и значений в массиве, а затем объединяет их, добавляя между ними пробел (эта методика часто используется при сериализации данных в длинные строки). Цикл `for`, который выполняется до индекса `attributes.length`, перебирает каждый атрибут, добавляя его имя и значение в строку. Два момента в этом коде заслуживают особого внимания.

- Браузеры возвращают элементы объекта `attributes` в разном порядке, поэтому расположение атрибутов в HTML- или XML-коде может отличаться от их очередности в объекте `attributes`.
- Internet Explorer 7 и более ранних версий возвращает все возможные атрибуты HTML-элемента, даже если они не заданы. Это означает, что часто возвращается более 100 атрибутов.

Для решения проблемы с Internet Explorer 7 и более ранних версий можно расширить эту функцию, чтобы она возвращала только заданные атрибуты. У каждого узла атрибута есть свойство `specified`, которому присваивается значение `true` при задании атрибута в HTML-коде или с помощью метода `setAttribute()`. В Internet Explorer у остальных атрибутов оно равно `false`, тогда как в других браузерах объект `attributes` содержит только заданные атрибуты (то есть свойство `specified` равно `true` у всех узлов атрибутов). Это позволяет подкорректировать функцию:

Листинг ElementAttributesExample04.htm


```
function outputAttributes(element){
    var pairs = new Array(),
        attrName,
        attrValue,
        i,
        len;

    for (i=0, len=element.attributes.length; i < len; i++){
        attrName = element.attributes[i].nodeName;
        attrValue = element.attributes[i].nodeValue;
        if (element.attributes[i].specified){
            pairs.push(attrName + "=\"" + attrValue + "\"");
        }
    }
    return pairs.join(" ");
}
```

Теперь в Internet Explorer 7 и более ранних версий эта функция будет возвращать только заданные атрибуты.

Создание элементов

Элементы можно создавать с помощью метода `document.createElement()`, который принимает имя тега создаваемого элемента. В HTML-документах регистр имени тега не учитывается, а в XML-документах (включая XHTML) — учитывается. Например, создать элемент `<div>` можно следующим образом:

```
var div = document.createElement("div");
```

Метод `createElement()` создает элемент и задает его свойство `ownerDocument`, после чего можно манипулировать атрибутами элемента, добавлять к нему дочерние элементы и т. д., например:

```
div.id = "myNewDiv";
div.className = "box";
```

Задание этих атрибутов для нового элемента только настраивает его, но не влияет на его отображение в браузере, потому что он не является частью дерева документа. Добавить элемент в дерево документа можно с помощью метода `appendChild()`, `insertBefore()` или `replaceChild()`. Следующий код добавляет новый элемент в элемент `<body>` документа:

Листинг CreateElementExample01.htm

```
document.body.appendChild(div);
```

Как только элемент добавлен в дерево документа, браузер сразу же его визуализирует. Любые последующие изменения элемента немедленно отражаются в браузере.

Internet Explorer поддерживает альтернативный формат метода `createElement()`, позволяющий сразу полностью задать элемент, включая атрибуты:

```
var div = document.createElement("<div id=\"myNewDiv\"" +
    " class=\"box\"></div>");
```

Этот формат помогает обходить перечисленные далее проблемы, связанные с динамическим созданием элементов в Internet Explorer 7.

- Для динамически созданных элементов `<iframe>` невозможно задать атрибут `name`.
- Динамически созданные элементы `<input>` невозможно сбросить с помощью метода `reset()` формы (см. главу 14).
- Динамически созданные элементы `<button>`, у которых атрибут `type` имеет значение `"reset"`, не сбрасывают форму.
- Динамически созданные переключатели с общим именем не объединяются в группу. Предполагается, что переключатели с общим именем должны определять разные значения одного параметра, но для динамически созданных переключателей эта зависимость не задается.

Все эти проблемы можно обойти, передав в метод `createElement()` полный HTML-код тега:

```
if (client.browser.ie && client.browser.ie <= 7){

    // создание встроенного тега с атрибутом name
    var iframe = document.createElement("<iframe name=\"myframe\">" +
        "</iframe>");

    // создание элемента input
    var input = document.createElement("<input type=\"checkbox\">");

    // создание кнопки
    var button = document.createElement("<button type=\"reset\"></button>");

    // создание пары переключателей
    var radio1 = document.createElement("<input type=\"radio\"" +
        " name=\"choice\"" + " value=\"1\">");
    var radio2 = document.createElement("<input type=\"radio\"" +
        " name=\"choice\"" + " value=\"2\">");

}
```

Как и при обычном вызове метода `createElement()`, в этом случае возвращается ссылка на DOM-элемент, который можно изменить или добавить в документ. Использовать альтернативный формат рекомендуется только для обхода одной из указанных проблем с Internet Explorer 7 и более ранних версий, потому что он требует распознавания браузера. Другие браузеры этот формат не поддерживают.

Дочерние узлы элементов

У элементов может быть любое количество дочерних узлов и более дальних потомков. Свойство `childNodes` содержит все непосредственные дочерние узлы элемента, которыми могут быть другие элементы, текстовые узлы, комментарии или инструкции по обработке. Способы идентификации этих узлов в браузерах существенно различаются. Рассмотрим, например, следующий код:

```
<ul id="myList">
    <li>Item 1</li>
    <li>Item 2</li>
    <li>Item 3</li>
</ul>
```

Internet Explorer 8 и более ранних версий распознает три дочерних узла у элемента `<ul>`: по одному для каждого из элементов `<li>`. Все остальные браузеры идентифицируют семь узлов: три элемента `<li>` и четыре узла `text`, представляющих свободное пространство между элементами. Если удалить свободное пространство, все браузеры возвращают одинаковое количество дочерних узлов, а именно три:

```
<ul id="myList"><li>Item 1</li><li>Item 2</li><li>Item 3</li></ul>
```

При перемещении по дочерним узлам с помощью свойства `childNodes` важно помнить об этих различиях. Часто перед той или иной операцией требуется проверить тип узла:

```
for (var i=0, len=element.childNodes.length; i < len; i++){
    if (element.childNodes[i].nodeType == 1){
        // обработка
    }
}
```

Этот код перебирает все дочерние узлы конкретного элемента, запуская обработку, только если тип узла равен 1 (то есть узел является элементом).

Для получения дочерних узлов и других потомков с конкретным именем тега можно использовать имеющийся у элементов метод `getElementsByTagName()`. Он работает почти так же, как аналогичный метод документа, но начинает поиск с элемента, поэтому возвращаются только его потомки. В предыдущем примере со списком `<ul>` все элементы `<li>` можно получить так:

```
var ul = document.getElementById("myList");
var items = ul.getElementsByTagName("li");
```

В данном случае все потомки элемента `<ul>` относятся к одному уровню, но если бы уровней было больше, этот код возвратил бы элементы `<li>` всех уровней.

Тип Text

Узлы `Text` представляются типом `Text` и содержат обычный текст, который интерпретируется буквально и может включать экранированные HTML-символы, но не HTML-код. Узел `Text` имеет следующие свойства:

- `nodeType` имеет значение 3;
- `nodeName` имеет значение `"#text"`;
- `nodeValue` содержит текст узла;
- `parentNode` указывает на узел `Element`;
- дочерние узлы не поддерживаются.

Для доступа к тексту в узле `Text` можно использовать свойства `nodeValue` и `data`, которые содержат одно и то же значение, а изменение одного из свойств дублируется в другом. Для работы с текстом в узле предназначены следующие методы:

- `appendData(`*текст*`)` — добавляет указанный текст в конец узла;
- `deleteData(`*смещение*, *количество*`)` — удаляет указанное количество символов, начиная с указанного смещения;
- `insertData(`*смещение*, *текст*`)` — вставляет текст в позиции, заданной смещением;
- `replaceData(`*смещение*, *количество*, *текст*`)` — заменяет указанное количество символов указанным текстом, начиная со смещения;
- `splitText(`*смещение*`)` — разделяет текстовый узел на два в позиции, заданной смещением;
- `substringData(`*смещение*, *количество*`)` — извлекает из текста указанное количество символов начиная со смещения.

Свойство `length` возвращает количество символов в узле, которое совпадает со значениями `nodeValue.length` и `data.length`.

По умолчанию у элементов, способных содержать контент, может быть не более одного текстового узла, например:

```
<!-- элемент пуст - текстового узла нет -->
<div></div>

<!-- элемент содержит пробел - один текстовый узел -->
<div> </div>

<!-- элемент содержит текст - один текстовый узел -->
<div>Hello World!</div>
```

В первом элементе `<div>` нет контента, поэтому и текстового узла у него нет. Если между открывающим и закрывающим тегами имеется какой-либо контент, пусть даже пробел, для него создается текстовый узел. Таким образом, у второго и третьего

элементов есть по одному дочернему текстовому узлу с пробелом и текстом "Hello World!" в качестве значения nodeValue. Для доступа к этому узлу можно использовать следующий код:

```
var textNode = div.firstChild;    // или div.childNodes[0]
```

Получив ссылку на текстовый узел, его можно изменить:

Листинг TextNodeExample01.htm

```
div.firstChild.nodeValue = "Some other message"; // какой-то другой текст
```

Пока текстовый узел находится в дереве документа, изменения узла вступают в силу немедленно. Отметим также, что значения текстовых узлов кодируются в формате HTML или XML (в зависимости от типа документа), то есть любые знаки «меньше», «больше» и кавычки экранируются:

Листинг TextNodeExample02.htm

```
// выводится как "Some &lt;strong&gt;other&lt;/strong&gt; message"
div.firstChild.nodeValue = "Some <strong>other</strong> message";
```

Это эффективный способ кодирования строки в формате HTML перед вставкой в DOM-документ.

Конструктор и прототип типа Text доступны в Internet Explorer 8, Firefox, Safari, Chrome и Opera.

Создание текстовых узлов

Текстовые узлы можно создавать с помощью метода document.createTextNode(). В качестве аргумента он принимает строку, которую нужно вставить в узел. Как и при задании значения существующего текстового узла, текст при этом кодируется в формате HTML или XML, например:

```
var textNode = document.createTextNode("<strong>Hello</strong> world!");
```

При создании текстового узла задается его свойство ownerDocument, но он не отображается в окне браузера, пока не будет добавлен к узлу в дереве документа. Следующий код создает элемент и добавляет в него сообщение:

Листинг TextNodeExample03.htm

```
var element = document.createElement("div");
element.className = "message";

var textNode = document.createTextNode("Hello world!");
element.appendChild(textNode);

document.body.appendChild(element);
```

Этот код создает элемент `<div>`, назначает ему класс `"message"`, а затем создает текстовый узел и добавляет его к уже имеющемуся элементу. Наконец, элемент добавляется в тело документа, при этом элемент и текстовый узел отображаются в браузере.

Как правило, у элементов имеется только один дочерний текстовый узел, но их может быть несколько, например:

Листинг TextNodeExample04.htm


```
var element = document.createElement("div");
element.className = "message";

var textNode = document.createTextNode("Hello world!");
element.appendChild(textNode);

var anotherTextNode = document.createTextNode("Yippee!");
element.appendChild(anotherTextNode);

document.body.appendChild(element);
```

Если один текстовый узел добавляется вслед за другим, их содержимое выводится на экран без пробела.

Нормализация текстовых узлов

Одноуровневые текстовые узлы в DOM-документах могут вносить путаницу, потому что любую простую строку можно представить одним текстовым узлом. Тем не менее текстовые узлы в DOM-документах можно часто встретить по соседству. Объединить такие узлы с помощью метода `normalize()`, который относится к типу `Node` и благодаря этому доступен для узлов всех типов. При вызове этого метода для родителя двух или более текстовых узлов они сливаются в один узел со свойством `nodeValue`, содержащим объединенные значения свойств `nodeValue` исходных узлов, например:

Листинг TextNodeExample05.htm

```
var element = document.createElement("div");
element.className = "message";

var textNode = document.createTextNode("Hello World!");
element.appendChild(textNode);

var anotherTextNode = document.createTextNode("Yippee!");
element.appendChild(anotherTextNode);

document.body.appendChild(element);

alert(element.childNodes.length);        // 2

element.normalize();
alert(element.childNodes.length);        // 1
alert(element.firstChild.nodeValue);     // "Hello World!Yippee!"
```

При синтаксическом анализе документа браузер никогда не создает одноуровневые текстовые узлы. Они могут появляться только при программном DOM-манипулировании.

Иногда метод normalize() вызывает критический сбой в работе Internet Explorer 6. Возможно, дефект был устранен в одном из исправлений, но подтверждений этому нет. В Internet Explorer 7 эта проблема не возникает.

Разделение текстовых узлов

У типа `Text` есть также метод `splitText()`, который создает из одного текстового узла два, разделяя значение `nodeValue` по указанному смещению. В исходном текстовом узле остается текст до смещения, а остальной текст сохраняется в новом текстовом узле, который возвращается из метода. Свойство `parentNode` нового узла имеет такое же значение, что и у исходного узла. Рассмотрим пример:

Листинг TextNodeExample06.htm


```
var element = document.createElement("div");
element.className = "message";

var textNode = document.createTextNode("Hello world!");
element.appendChild(textNode);

document.body.appendChild(element);

var newNode = element.firstChild.splitText(11);
alert(element.firstChild.nodeValue);     // "Hello"
alert(newNode.nodeValue);                // " world!"
alert(element.childNodes.length);        // 2
```

В этом примере текстовый узел со значением `"Hello world!"` делится на два узла в позиции 11, которая соответствует пробелу между словами. После этого исходный текстовый узел содержит строку `"Hello"`, а новый — текст `" world!"` (с начальным пробелом).

Разделение текстовых узлов чаще всего применяется при анализе DOM для извлечения нужных данных.

Тип Comment

Комментарии представляются в DOM экземплярами типа `Comment` со следующими свойствами:

- `nodeType` имеет значение 8;
- `nodeName` имеет значение `"#comment"`;
- `nodeValue` содержит комментарий;

- `parentNode` указывает на узел `Document` или `Element`;
- дочерние узлы не поддерживаются.

Тип `Comment` наследуется от того же базового типа, что и `Text`, поэтому у него есть такие же методы манипулирования строками, исключая `splitText()`. Подобно типу `Text`, получить фактический комментарий можно с помощью свойства `nodeValue` или `data`.

Узел комментария доступен из родительского узла. Рассмотрим следующий HTML-код:

```
<div id="myDiv"><!-- Комментарий --></div>
```

Этот комментарий является дочерним узлом элемента `<div>` и доступен следующим образом:

Листинг CommentNodeExample01.htm


```
var div = document.getElementById("myDiv");
var comment = div.firstChild;
alert(comment.data);    // "Комментарий"
```

Узел комментария можно создать с помощью метода `document.createComment()`, передав в него текст комментария:

```
var comment = document.createComment("Комментарий");
```

Разработчики редко создают и используют узлы комментариев, потому что функционально они почти бесполезны. Кроме того, браузеры не распознают комментарии после закрывающего тега `</html>`. Если вам нужен доступ к узлам комментариев, убедитесь, что они являются потомками элемента `<html>`.

Конструктор и прототип типа Comment доступны в Firefox, Safari, Chrome и Opera. В Internet Explorer 8 узлы комментариев считаются элементами с именем тега «!», поэтому их можно получать методом getElementsByTagName(). В Internet Explorer 9 для создания комментариев можно использовать специальный конструктор HTMLCommentElement, но элементами они не считаются.

Тип CDATASection

CDATA-разделы специфичны для XML-документов и представляются типом `CDATASection`. Подобно типу `Comment`, он наследуется от базового типа `Text` и содержит все его методы манипулирования строками, кроме `splitText()`. Перечислим свойства узла `CDATASection`:

- `nodeType` имеет значение 4;
- `nodeName` имеет значение `"#cdata-section"`;

- nodeValue представляет собой содержимое CDATA-раздела;
- parentNode указывает на узел Document или Element;
- дочерние узлы не поддерживаются.

CDATA-разделы допустимы только в XML-документах и ошибочно интерпретируются в большинстве браузеров как узлы Comment или Element. Рассмотрим пример:

```
<div id="myDiv"><![CDATA[Какой-то контент.]]></div>
```

В этом примере узел CDATASection должен быть первым дочерним узлом элемента , но ни один из четырех основных браузеров его так не интерпретирует. Даже правильные XHTML-страницы со встроенными разделами CDATA обрабатываются браузерами некорректно.

Создать раздел CDATA в XML-документе можно с помощью метода document.createCDataSection(), передав ему содержимое узла.

Конструктор и прототип типа CDATASection доступны в Firefox, Safari, Chrome и Opera. В Internet Explorer до версии 9 включительно он не поддерживается.

Тип DocumentType

Тип DocumentType используется в браузерах нечасто и поддерживается только в Firefox, Safari и Opera. Объект DocumentType содержит все сведения о типе документа. Свойства объекта DocumentType:

- nodeType имеет значение 10;
- nodeName содержит имя типа документа;
- nodeValue имеет значение null;
- parentNode указывает на узел Document;
- дочерние узлы не поддерживаются.

В DOM Level 1 объекты DocumentType нельзя создавать динамически; они создаются только при синтаксическом анализе кода документа. В браузерах, которые поддерживают этот тип, объект DocumentType хранится в свойстве document.doctype. DOM Level 1 определяет три свойства объектов DocumentType: name — имя типа документа, entities — набор NamedNodeMap, который содержит сущности, описываемые типом документа, и notations — набор NamedNodeMap, который содержит обозначения, описываемые типом документа. Поскольку документы, которые отображаются в браузерах, чаще всего имеют тип HTML или XHTML, списки entities и notations обычно пусты (они заполняются только встроенными типами документов). На практике полезно только свойство name, которому присваивается имя типа документа, то есть текст сразу после <!DOCTYPE. Рассмотрим следующий строгий тип документа HTML 4.01:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
  "http://www.w3.org/TR/html4/strict.dtd">
```

У этого типа документа свойство `name` имеет значение `"HTML"`:

```
alert(document.doctype.name);    // "HTML"
```

В Internet Explorer 8 и более ранних версий тип `DocumentType` не поддерживается и свойство `document.doctype` всегда имеет значение `null`. Кроме того, эти браузеры ошибочно интерпретируют объявление типа документа как комментарий и создают для него узел комментария. Internet Explorer 9 правильно назначает свойству `document.doctype` объект, но не предоставляет доступ к типу `DocumentType`.

Тип DocumentFragment

Тип `DocumentFragment` — единственный тип узла, у которого нет соответствия в разметке. В DOM фрагмент документа определен как «облегченный» документ, который может содержать узлы и манипулировать ими без накладных расходов, связанных с целым документом. Узлы `DocumentFragment` обладают следующими свойствами:

- `nodeType` имеет значение 11;
- `nodeName` имеет значение `"#document-fragment"`;
- `nodeValue` имеет значение `null`;
- `parentNode` имеет значение `null`;
- дочерними узлами могут быть узлы `Element`, `ProcessingInstruction`, `Comment`, `Text`, `CDATASection` и `EntityReference`.

Фрагмент документа невозможно добавить в документ. Вместо этого он выступает в роли хранилища других узлов, которые может потребоваться добавить. Фрагменты документа создаются методом `document.createDocumentFragment()`:

```
var fragment = document.createDocumentFragment();
```

Фрагменты документа наследуют все методы от типа `Node` и обычно используются для выполнения разных операций с DOM, которые затем применяются к документу. При добавлении узла во фрагмент документа этот узел удаляется из дерева документа и больше не отображается в браузере. Новые узлы, добавляемые во фрагмент документа, тоже не становятся частью дерева документа. Содержимое фрагмента документа можно добавить в документ с помощью метода `appendChild()` или `insertBefore()`. Когда фрагмент документа передается как аргумент в один из этих методов, в документ добавляются все дочерние узлы фрагмента, но сам он никогда не добавляется в дерево документа. Рассмотрим, например, следующий HTML-код:

```
<ul id="myList"></ul>
```

Предположим, что нам нужно добавить в этот элемент `<ul>` три элемента списка. При добавлении элементов по отдельности браузер каждый раз визуализировал бы страницу заново. Чтобы избежать этого, можно создать фрагмент документа с элементами списка и добавить их за один раз:

Листинг DocumentFragmentExample01.htm


```
var fragment = document.createDocumentFragment();
var ul = document.getElementById("myList");
var li = null;

for (var i=0; i < 3; i++){
    li = document.createElement("li");
    li.appendChild(document.createTextNode("Item " + (i+1)));
    fragment.appendChild(li);
}

ul.appendChild(fragment);
```

Этот код начинается с создания фрагмента документа и получения ссылки на элемент `<ul>`. Затем в цикле `for` мы создаем три элемента списка с порядковыми номерами. Для этого в теле цикла мы создаем элемент `<li>` с текстовым узлом и добавляем его к фрагменту документа с помощью метода `appendChild()`. По завершении цикла все элементы списка добавляются в элемент `<ul>` с помощью метода `appendChild()`, которому передается фрагмент документа. При вызове метода все дочерние узлы фрагмента документа удаляются из него и помещаются в элемент `<ul>`.

Тип Attr

Атрибуты элемента представляются в DOM типом `Attr`. Его конструктор и прототип доступны во всех браузерах, в том числе в Internet Explorer начиная с версии 8. Технически атрибуты являются узлами, которые хранятся в свойстве `attributes` элемента. Узлы атрибутов обладают следующими свойствами:

- `nodeType` имеет значение 11;
- `nodeName` содержит имя атрибута;
- `nodeValue` содержит значение атрибута;
- `parentNode` имеет значение `null`;
- в HTML дочерние узлы не поддерживаются;
- в XML дочерними узлами могут быть объекты `Text` и `EntityReference`.

Хотя атрибуты являются узлами, они не считаются частями дерева DOM-документа. Большинство разработчиков редко обращаются к ним напрямую, предпочитая методы `getAttribute()`, `setAttribute()` и `removeAttribute()`.

У объекта `Attr` есть три свойства: `name` — имя атрибута (то же, что `nodeName`); `value` — значение атрибута (то же, что `nodeValue`); `specified` — логическое значение,

указывающее, был ли атрибут задан в коде или имеет значение, предлагаемое по умолчанию.

Узел атрибута можно создать с помощью метода `document.createAttribute()`, передав в него имя атрибута. Например, следующий код добавляет к элементу атрибут `align`:

Листинг AttrExample01.htm


```
var attr = document.createAttribute("align");
attr.value = "left";
element.setAttributeNode(attr);

alert(element.attributes["align"].value);          // "left"
alert(element.getAttributeNode("align").value);    // "left"
alert(element.getAttribute("align"));              // "left"
```

В первой строке создается узел атрибута, при этом его свойство `name` задается методом `createAttribute()`, так что назначать его позже не потребуется. Затем свойству `value` присваивается значение `"left"`. Для добавления нового атрибута к элементу вызывается метод `setAttributeNode()`. Когда атрибут добавлен, к нему можно обращаться с помощью свойства `attributes`, а также методов `getAttributeNode()` и `getAttribute()`. В первых двух случаях возвращается фактический узел `Attr` атрибута, тогда как метод `getAttribute()` возвращает только значение атрибута.

Трудно придумать уважительную причину для непосредственного доступа к узлам атрибутов. Вместо этого лучше использовать методы getAttribute(), setAttribute() и removeAttribute().

Работа с DOM

Во многих случаях работать с DOM довольно легко, и с помощью JavaScript можно легко создавать разметку, которая обычно пишется на HTML. Однако иногда DOM не так проста, как кажется. Из-за многих скрытых особенностей и несоответствий браузеров одни части DOM более сложны, чем другие.

Динамические сценарии

Элемент `<script>` позволяет вставить в страницу JavaScript-сценарий из внешнего файла с помощью атрибута `src` или добавить код непосредственно. Динамическими называют такие сценарии, которые не существуют при загрузке страницы, а добавляются позднее с помощью DOM. Как и в случае элемента `<script>`, это можно сделать двумя способами: прочитать код из внешнего файла или вставить его непосредственно.

В динамической загрузке внешнего JavaScript-файла нет ничего необычного. Возьмем для примера следующий элемент `<script>`:

```
<script type="text/javascript" src="client.js"></script>
```

Этот элемент <script> включает в код страницы сценарий распознавания клиента из главы 9. Ему соответствует следующий DOM-код:

```
var script = document.createElement("script");
script.type = "text/javascript";
script.src = "client.js";
document.body.appendChild(script);
```

Внешний файл загружается с сервера, только когда элемент <script> добавляется в код страницы в последней строке. Его также можно добавить в элементе <head> с тем же эффектом. На основе этого кода можно создать универсальную функцию загрузки сценариев:

```
function loadScript(url){
    var script = document.createElement("script");
    script.type = "text/javascript";
    script.src = url;
    document.body.appendChild(script);
}
```

Теперь внешние JavaScript-файлы можно загружать следующим образом:

```
loadScript("client.js");
```

После загрузки сценарий становится доступен в остальной части страницы, но как узнать, когда он полностью загружен? К сожалению, стандартного способа сделать это не существует. Можно, например, использовать определенные события, которые мы обсудим в главе 13.

Другой способ добавить в страницу JavaScript-код — встраивание, например:

```
<script type="text/javascript">
    function sayHi(){
        alert("Hello");
}
</script>
```

Логично было бы предположить, что соответствующий DOM-код будет таким:

```
var script = document.createElement("script");
script.type = "text/javascript";
script.appendChild(document.createTextNode(
    "function sayHi(){alert('Hello');}"));
document.body.appendChild(script);
```

Этот код работает в Firefox, Safari, Chrome и Opera, но в Internet Explorer он приводит к ошибке, потому что Internet Explorer интерпретирует элементы <script> как специальные и блокирует доступ к их дочерним узлам в стиле DOM. Вместо этого можно назначить JavaScript-код свойству text, которое есть у всех элементов <script>:

Листинг DynamicScriptExample01.htm


```
var script = document.createElement("script");
script.type = "text/javascript";
script.text = "function sayHi(){alert('Hello');}";
document.body.appendChild(script);
```

Этот обновленный код работает в Internet Explorer, Firefox, Opera, а также в Safari 3 и более поздних версий. Safari до версии 3 реализует свойство `text` неправильно, но позволяет назначить нужный код текстовому узлу. Следовательно, если нужно обеспечить поддержку ранних версий Safari, можно использовать такой код:

```
var script = document.createElement("script");
script.type = "text/javascript";
var code = "function sayHi(){alert('Hello');}";
try {
    script.appendChild(document.createTextNode("code"));
} catch (ex){
    script.text = "code";
}
document.body.appendChild(script);
```

Здесь мы сначала пробуем стандартный в DOM способ с текстовым узлом, потому что он работает во всех браузерах, кроме Internet Explorer. Если возникает ошибка, значит, мы имеем дело с Internet Explorer и должны использовать свойство `text`. Теперь можно создать универсальную функцию:

Листинг DynamicScriptExample02.htm


```
function loadScriptString(code){
    var script = document.createElement("script");
    script.type = "text/javascript";
    try {
        script.appendChild(document.createTextNode(code));
    } catch (ex){
        script.text = code;
    }
    document.body.appendChild(script);
}
```

Используется эта функция следующим образом:

```
loadScriptString("function sayHi(){alert('Hello');}");
```

Код, загруженный таким образом, выполняется в глобальной области видимости и доступен сразу же после завершения сценария. По сути, этот подход эквивалентен передаче строки кода в метод `eval()` в глобальной области видимости.

Динамические стили

CSS-стили добавляются в HTML-код с помощью элементов `<link>` и `<style>`. Первый включает CSS из внешнего файла, а второй используется для указания встроенных

стилей. Как и динамические сценарии, динамические стили отсутствуют в коде страницы при ее первоначальной загрузке и добавляются, только когда страница загружена.

Вот пример типичного элемента `<link>`:

```
<link rel="stylesheet" type="text/css" href="styles.css">
```

Его также можно легко создать с помощью следующего DOM-кода:

```
var link = document.createElement("link");
link.rel = "stylesheet";
link.type = "text/css";
link.href = "styles.css";
var head = document.getElementsByTagName("head")[0];
head.appendChild(link);
```

Этот код нормально работает во всех основных браузерах. Имейте в виду, что элементы `<link>` нужно добавлять в элемент `<head>`, а не `<body>`, чтобы все браузеры обрабатывали их правильно. Код загрузки стилей можно обобщить с помощью следующей функции:

```
function loadStyles(url){
    var link = document.createElement("link");
    link.rel = "stylesheet";
    link.type = "text/css";
    link.href = url;
    var head = document.getElementsByTagName("head")[0];
    head.appendChild(link);
}
```

Использовать эту функцию можно так:

```
loadStyles("styles.css");
```

Стили из внешнего файла загружаются асинхронно, то есть независимо от другого JavaScript-кода. Определять, когда завершается загрузка стилей, обычно не требуется, но это можно сделать с помощью событий (см. главу 13).

Другой способ определить стили — использовать элемент `<style>`, содержащий встроенный CSS-код, например:

```
<style type="text/css">
body {
    background-color: red;
}
</style>
```

По идее, с той же целью можно использовать следующий DOM-код:

Листинг DynamicStyleExample01.htm


```
var style = document.createElement("style");
style.type = "text/css";
style.appendChild(document.createTextNode("body{background-color:red}"));
var head = document.getElementsByTagName("head")[0];
head.appendChild(style);
```

Этот код работает в Firefox, Safari, Chrome и Opera, но не в Internet Explorer. Подобно узлу `<script>`, Internet Explorer считает узел `<style>` специальным и запрещает доступ к его дочерним узлам. При этом генерируется такая же ошибка, что и при попытке добавить дочерний узел в элемент `<script>`. Обойти эту проблему можно с помощью свойства `styleSheet`, содержащего, в свою очередь, свойство `cssText`, которому можно назначить CSS-код (оба эти свойства обсуждаются в главе 12):

```
var style = document.createElement("style");
style.type = "text/css";
try{
    style.appendChild(document.createTextNode(
        "body{background-color:red}"));
} catch (ex){
    style.styleSheet.cssText = "body{background-color:red}";
}
var head = document.getElementsByTagName("head")[0];
head.appendChild(style);
```

Как и при динамическом добавлении встроенных сценариев, ошибка в Internet Explorer перехватывается здесь в блоке `catch`, в котором затем используется способ задания стилей, специфичный для Internet Explorer. Универсальное решение будет таким:

Листинг DynamicStyleExample02.htm


```
function loadStyleString(css){
    var style = document.createElement("style");
    style.type = "text/css";
    try{
        style.appendChild(document.createTextNode(css));
    } catch (ex){
        style.styleSheet.cssText = css;
    }
    var head = document.getElementsByTagName("head")[0];
    head.appendChild(style);
}
```

Вызвать эту функцию можно следующим образом:

```
loadStyleString("body{background-color:red}");
```

Стили, заданные таким образом, добавляются в страницу незамедлительно и сразу же изменяют ее вид.

Если вы пишете код специально для Internet Explorer, будьте осторожны со свойством styleSheet.cssText. Если при повторном использовании элемента <style> попытаться задать это свойство более одного раза, может произойти критический сбой в работе браузера. К нему также может привести присваивание свойству cssText пустой строки. Остается надеяться, что этот дефект будет исправлен в будущем.

Работа с таблицами

Элемент `<table>` является в HTML одним из самых сложных. Чтобы создать таблицу, обычно нужно добавить большое количество тегов строк, ячеек, заголовков и т. д. Неудивительно, что для создания и изменения таблиц с использованием базовых DOM-методов может потребоваться много кода. Допустим, мы хотим создать с помощью DOM следующую HTML-таблицу:

```
<table border="1" width="100%">
    <tbody>
        <tr>
            <td>Cell 1,1</td>
            <td>Cell 2,1</td>
        </tr>
        <tr>
            <td>Cell 1,2</td>
            <td>Cell 2,2</td>
        </tr>
    </tbody>
</table>
```

Если использовать только базовые DOM-методы, получится примерно такой код:

```
// создание таблицы
var table = document.createElement("table");
table.border = 1;
table.width = "100%";

// создание тела таблицы
var tbody = document.createElement("tbody");
table.appendChild(tbody);

// создание первой строки
var row1 = document.createElement("tr");
tbody.appendChild(row1);
var cell1_1 = document.createElement("td");
cell1_1.appendChild(document.createTextNode("Cell 1,1"));
row1.appendChild(cell1_1);
var cell2_1 = document.createElement("td");
cell2_1.appendChild(document.createTextNode("Cell 2,1"));
row1.appendChild(cell2_1);

// создание второй строки
var row2 = document.createElement("tr");
tbody.appendChild(row2);
var cell1_2 = document.createElement("td");
cell1_2.appendChild(document.createTextNode("Cell 1,2"));
row2.appendChild(cell1_2);
```

```
var cell2_2= document.createElement("td");
cell2_2.appendChild(document.createTextNode("Cell 2,2"));
row2.appendChild(cell2_2);

// добавление таблицы в тело документа
document.body.appendChild(table);
```

Этот код довольно объемен и сложен для понимания. Чтобы упростить создание таблиц, в DOM HTML для элементов `<table>`, `<tbody>` и `<tr>` определены дополнительные свойства и методы.

Для элемента `<table>` доступны следующие свойства и методы:

- `caption` — указатель на элемент `<caption>` (если он существует);
- `tBodies` — коллекция `HTMLCollection`, содержащая элементы `<tbody>`;
- `tFoot` — указатель на элемент `<tfoot>` (если он существует);
- `tHead` — указатель на элемент `<thead>` (если он существует);
- `rows` — коллекция `HTMLCollection`, содержащая все строки таблицы;
- `createTHead()` — создает элемент `<thead>`, помещает его в таблицу и возвращает ссылку на него;
- `createTFoot()` — создает элемент `<tfoot>`, помещает его в таблицу и возвращает ссылку на него;
- `createCaption()` — создает элемент `<caption>`, помещает его в таблицу и возвращает ссылку на него;
- `deleteTHead()` — удаляет элемент `<thead>`;
- `deleteTFoot()` — удаляет элемент `<tfoot>`;
- `deleteCaption()` — удаляет элемент `<caption>`;
- `deleteRow(`*позиция*`)` — удаляет строку в указанной позиции;
- `insertRow(`*позиция*`)` — вставляет строку в коллекцию `rows` в указанной позиции.

Для элемента `<tbody>` доступны следующие свойства и методы:

- `rows` — набор `HTMLCollection`, содержащий строки элемента `<tbody>`;
- `deleteRow(`*позиция*`)` — удаляет строку в указанной позиции;
- `insertRow(`*позиция*`)` — вставляет строку в коллекцию `rows` в указанной позиции и возвращает ссылку на нее.

Для элемента `<tr>` доступны следующие свойства и методы:

- `cells` — коллекция `HTMLCollection`, содержащая ячейки элемента `<tr>`;
- `deleteCell(`*позиция*`)` — удаляет ячейку в указанной позиции;
- `insertCell(`*позиция*`)` — вставляет ячейку в коллекцию `cells` в указанной позиции и возвращает ссылку на нее.

Эти свойства и методы значительно сокращают код создания таблицы. Например, предыдущий пример можно переписать с их помощью следующим образом (новый код выделен):

```
// создание таблицы
var table = document.createElement("table");
table.border = 1;
table.width = "100%";

// создание тела таблицы
var tbody = document.createElement("tbody");
table.appendChild(tbody);

// создание первой строки
tbody.insertRow(0);
tbody.rows[0].insertCell(0);
tbody.rows[0].cells[0].appendChild(document.createTextNode("Cell 1,1"));
tbody.rows[0].insertCell(1);
tbody.rows[0].cells[1].appendChild(document.createTextNode("Cell 2,1"));

// создание второй строки
tbody.insertRow(1);
tbody.rows[1].insertCell(0);
tbody.rows[1].cells[0].appendChild(document.createTextNode("Cell 1,2"));
tbody.rows[1].insertCell(1);
tbody.rows[1].cells[1].appendChild(document.createTextNode("Cell 2,2"));

// добавление таблицы в тело документа
document.body.appendChild(table);
```

Код создания элементов `<table>` и `<tbody>` остался прежним. Изменилось только создание двух строк, для чего теперь используются свойства и методы DOM HTML. Чтобы создать первую строку, для элемента `<tbody>` вызывается метод `insertRow()` с аргументом 0, который указывает позицию новой строки в таблице. После этого строка доступна как `tbody.rows[0]`.

Ячейки создаются подобным образом — с помощью метода `insertCell()`, который вызывается для элемента `<tr>` с аргументом, указывающим позицию новой ячейки. Например, первая ячейка в первой строке доступна после создания как `tbody.rows[0].cells[0]`.

Использование этих свойств и методов для создания таблиц делает код более логичным и упрощает его чтение, хотя и первый подход вполне приемлем.

Использование объектов NodeList

Знание особенностей объекта `NodeList` и похожих на него объектов `NamedNodeMap` и `HTMLCollection` критически важно для хорошего понимания DOM в целом. Все эти наборы динамически обновляются при изменении структуры документа, благодаря чему всегда содержат наиболее актуальные сведения. На самом деле все

объекты NodeList являются запросами, которые выполняются для DOM-документа при обращении к ним. Например, выполнение следующего кода приводит к бесконечному циклу:

```
var divs = document.getElementsByTagName("div"),
    i,
    div;

for (i=0; i < divs.length; i++){
    div = document.createElement("div");
    document.body.appendChild(div);
}
```

Первая строка этого кода возвращает объект HTMLCollection, содержащий все элементы в документе. Так как эта коллекция динамически обновляется, в нее заносится каждый новый элемент , добавляемый на страницу. Браузер не ведет список всех созданных коллекций, а обновляет их, только когда они используются в коде, из-за чего возникает интересная проблема с циклом. В начале каждой его итерации оценивается условие i < divs.length, при этом выполняется запрос на получение всех элементов . Поскольку в теле цикла в документ добавляется новый элемент , значение divs.length увеличивается при каждой итерации цикла и всегда превышает i.

Всякий раз, когда нужно перебрать коллекцию NodeList, лучше инициализировать вторую переменную значением ее длины и сравнивать итератор с этой переменной:

```
var divs = document.getElementsByTagName("div"),
    i,
    len,
    div;

for (i=0, len=divs.length; i < len; i++){
    div = document.createElement("div");
    document.body.appendChild(div);
}
```

Здесь переменной len присваивается значение divs.length на момент начала цикла. Так как при каждой итерации оно остается неизменным, это предотвращает бесконечный цикл. Этот прием использовался в данной главе при демонстрации рекомендуемого способа перебора объектов NodeList.

Обращаться к объекту NodeList без необходимости не следует, потому что при этом каждый раз выполняется запрос документа. Для экономии ресурсов старайтесь кэшировать часто используемые значения, полученные из NodeList.

Резюме

DOM — это API, который не зависит от языка и предназначен для доступа к HTML- и XML-документам и выполнения различных операций с ними и их содержимым.

DOM Level 1 представляет HTML- и XML-документы в виде иерархии узлов, манипулируя которыми с помощью JavaScript можно изменять вид и структуру документов.

DOM определяет несколько типов узлов.

- В основе всех узлов лежит тип `Node`, который является абстрактным представлением отдельной части документа; от него наследуются все остальные типы узлов.
- Тип `Document` представляет весь документ и располагается в корне иерархии. В JavaScript к этому типу относится объект `document`, с помощью которого можно запрашивать и получать узлы разными способами.
- Все элементы HTML или XML в документе имеют тип `Element`, который обеспечивает средства для манипулирования их содержимым и атрибутами.
- Существуют типы узлов для текстового контента, комментариев, типов документов, CDATA-разделов и фрагментов документов.

В большинстве случаев DOM работает без неожиданностей, но при использовании элементов `<script>` и `<style>` часто возникают осложнения. Поскольку эти элементы содержат сценарии и стили, они часто обрабатываются в браузерах не так, как другие элементы.

Крайне важно понимать, как DOM влияет на общее быстродействие кода. Манипуляции с DOM-элементами входят в число самых ресурсоемких JavaScript-операций, что особенно заметно в случае объектов `NodeList`, которые обновляются при каждом обращении к ним. По этой причине количество операций с DOM желательно свести к минимуму.

11 Расширения DOM

- Selectors
- Использование расширений DOM HTML5
- Работа с фирменными расширениями DOM

Хотя DOM API определен довольно четко, его функционал часто расширяют с помощью стандартизированных и фирменных компонентов. До 2008 года почти все DOM-расширения в браузерах были фирменными. Некоторые из них, ставшие стандартами де-факто, консорциум W3C описал в формальных спецификациях.

Двумя основными стандартами DOM-расширений являются Selectors и HTML5. Оба они были разработаны для стандартизации актуальных API и методик разработки в соответствии с интересами и требованиями сообщества. Дополнительные DOM-свойства определены также в более компактной спецификации, которая называется Element Traversal. Хотя Selectors и особенно HTML5 охватывают большое количество DOM-расширений, по-прежнему продолжают использоваться фирменные расширения, которые также рассматриваются в этой главе.

Selectors

Одной из наиболее востребованных возможностей JavaScript-библиотек является получение DOM-элементов с помощью CSS-селекторов. Например, библиотека jQuery (www.jquery.com) полностью построена на этом подходе, который заменяет методы `getElementById()` и `getElementsByTagName()`.

Консорциум W3C разработал спецификацию Selectors API (www.w3.org/TR/selectors-api), чтобы стандартизировать встроенную поддержку CSS-запросов в браузерах.

Ранее для реализации CSS-запросов в JavaScript-библиотеках приходилось разрабатывать синтаксические анализаторы CSS-кода и использовать существующие DOM-методы для перемещения по документу и идентификации запрошенных узлов. Несмотря на все усилия разработчиков по оптимизации обработки таких запросов на JavaScript, результаты оставляли желать лучшего. Созданный для решения проблемы встроенный API позволил осуществлять синтаксический анализ запросов и навигацию по дереву на уровне браузера на компилируемом языке, что существенно повысило быстродействие кода.

В основе Selectors API Level 1 лежат методы `querySelector()` и `querySelectorAll()`, доступные для типов `Document` и `Element`. Selectors API Level 1 полностью реализован в Internet Explorer 8+, Firefox 3.5+, Safari 3.1+, Chrome и Opera 10+.

Метод querySelector()

Метод `querySelector()` принимает CSS-запрос и возвращает первый соответствующий ему элемент или значение `null`, если таких элементов нет, например:

Листинг SelectorsAPIExample01.htm


```
// получение элемента body
var body = document.querySelector("body");

// получение элемента с идентификатором "myDiv"
var myDiv = document.querySelector("#myDiv");

// получение первого элемента класса "selected"
var selected = document.querySelector(".selected");

// получение первого изображения класса "button"
var img = document.body.querySelector("img.button");
```

При вызове метода `querySelector()` для типа `Document` сопоставление с шаблоном начинается с элемента документа; при вызове для типа `Element` поиск совпадения начинается с элемента и выполняется в нисходящем порядке только среди его потомков.

CSS-запрос может быть сколь угодно сложным. Если он имеет неправильный синтаксис или содержит неподдерживаемый селектор, возникает ошибка.

Метод querySelectorAll()

Метод `querySelectorAll()` принимает CSS-запрос и возвращает все соответствующие ему узлы в статическом экземпляре `NodeList`.

Если точнее, возвращается объект `NodeList` со всеми свойствами и методами, но на самом деле он реализован как «снимок» элементов на текущий момент времени, а не как динамический запрос документа. Благодаря этому он работает гораздо быстрее, чем обычные объекты `NodeList`.

Любой вызов метода `querySelectorAll()` с допустимым CSS-запросом возвращает объект `NodeList` независимо от количества соответствующих запросу элементов; если соответствий нет, объект `NodeList` будет пустым.

Как и `querySelector()`, метод `querySelectorAll()` доступен для типов `Document`, `DocumentFragment` и `Element`. Вот несколько примеров:

Листинг SelectorsAPIExample02.htm


```
// получение всех элементов <em> в <div> (аналог getElementsByTagName("em"))
var ems = document.getElementById("myDiv").querySelectorAll("em");

// получение всех элементов класса "selected"
var selecteds = document.querySelectorAll(".selected");

// получение всех элементов <strong> в элементах <p>
var strongs = document.querySelectorAll("p strong");
```

Возвращенный объект `NodeList` можно перебирать, извлекая из него отдельные элементы с помощью метода `item()` или скобочной нотации:

```
var i, len, strong;
for (i=0, len=strongs.length; i < len; i++){
    strong = strongs[i];        // или strongs.item(i)
    strong.className = "important";
}
```

Если CSS-селектор не поддерживается браузером или имеет неправильный синтаксис, вызов метода `querySelectorAll()` завершается ошибкой.

Метод matchesSelector()

В Selectors API Level 2 представлен метод `matchesSelector()` для типа `Element`. Он принимает CSS-селектор и возвращает `true`, если элемент соответствует селектору, и `false` в противном случае, например:

```
if (document.body.matchesSelector("body.page1")){
    // true
}
```

Так можно легко проверить, будет ли возвращен элемент методом `querySelector()` или `querySelectorAll()`, если у вас уже есть ссылка на элемент.

На момент написания книги (середина 2011 года) ни один из браузеров не поддерживал метод `matchesSelector()`, но в некоторых из них были экспериментальные реализации. В Internet Explorer 9+ этот метод доступен как `msMatchesSelector()`, в Firefox 3.6+ он называется `mozMatchesSelector()`, а в Safari 5+ и Chrome — `webkitMatchesSelector()`. Для работы с ними можно создать функцию-оболочку:

Листинг SelectorsAPIExample03.htm

```
function matchesSelector(element, selector){
    if (element.matchesSelector){
        return element.matchesSelector(selector);
    } else if (element.msMatchesSelector){
        return element.msMatchesSelector(selector);
    } else if (element.mozMatchesSelector){
        return element.mozMatchesSelector(selector);
    } else if (element.webkitMatchesSelector){
        return element.webkitMatchesSelector(selector);
    } else {
        throw new Error("Not supported.");
    }
}

if (matchesSelector(document.body, "body.page1")){
    // какие-то действия
}
```

Element Traversal

В отличие от других браузеров, Internet Explorer до версии 9 не создает текстовые узлы для свободного пространства между элементами, из-за чего возникают несоответствия при использовании таких свойств, как `childNodes` и `firstChild`. Для преодоления этих различий с соблюдением требований модели DOM была определена спецификация Element Traversal API (www.w3.org/TR/ElementTraversal/).

Этот API добавляет ко всем DOM-элементам пять новых свойств:

- `childElementCount` — возвращает количество дочерних элементов (исключая текстовые узлы и комментарии);
- `firstElementChild` — указывает на первый дочерний элемент (версия свойства `firstChild` только для элементов);
- `lastElementChild` — указывает на последний дочерний элемент (версия свойства `lastChild` только для элементов);
- `previousElementSibling` — указывает на предыдущий элемент того же уровня (версия свойства `previousSibling` только для элементов);
- `nextElementSibling` — указывает на следующий элемент того же уровня (версия свойства `nextSibling` только для элементов).

Эти свойства упрощают перебор DOM-элементов, позволяя не беспокоиться о текстовых узлах для свободного пространства.

Например, традиционный кроссбраузерный способ перебора всех дочерних элементов конкретного элемента выглядит так:

```
var i,
    len,
    child = element.firstChild;
while(child != element.lastChild){
    if (child.nodeType == 1){    // это элемент?
        processChild(child);
    }
    child = child.nextSibling;
}
```

Использование свойств из спецификации Element Traversal позволяет упростить код:

```
var i,
    len,
    child = element.firstElementChild;
while(child != element.lastElementChild){
    processChild(child);    // уже известно, что это элемент
    child = child.nextElementSibling;
}
```

Element Traversal API реализован в Internet Explorer 9+, Firefox 3.5+, Safari 4+, Chrome и Opera 10+.

HTML5

HTML5 представляет радикальный отход от традиций HTML. Никакие предыдущие спецификации HTML не описывали JavaScript-интерфейсы, а определяли исключительно разметку, при этом связи между HTML и JavaScript регламентировала спецификация DOM.

Что касается спецификации HTML5, то она содержит множество JavaScript API, разработанных для использования с новыми элементами разметки. Некоторые из этих API перекрываются с DOM и определяют DOM-расширения, подлежащие реализации в браузерах.

Спецификация HTML5 очень широка, поэтому в данном разделе рассматриваются только те ее части, которые относятся ко всем DOM-узлам. Другие части HTML5 мы обсудим позже.

Новые средства работы с классами

За время существования HTML4 веб-разработчики стали чаще использовать атрибут `class` для указания стилистической и семантической информации об элементах. Многие JavaScript-сценарии включают код динамического изменения CSS-классов и поиска элементов, относящихся к конкретному классу. HTML5 поддерживает ряд новых средств, которые упрощают работу с классами.

Метод getElementsByClassName()

Одним из наиболее популярных новшеств в HTML5 стал метод `getElementsByClassName()`, который доступен для объекта `document` и всех HTML-элементов. Благодаря встроенной реализации он значительно превосходит по быстродействию свои аналоги из JavaScript-библиотек, основанные на использовании DOM.

Метод `getElementsByClassName()` принимает строку с одним или несколькими именами классов и возвращает объект `NodeList` с элементами, к которым применены все эти классы. Порядок следования классов в строке не имеет значения, например:

```
// получение всех элементов, относящихся к классам "username
// " и "current", без учета порядка следования классов
var allCurrentUsernames =
    document.getElementsByClassName("username current");

// получение всех элементов класса "selected" в поддереве myDiv
var selected =
    document.getElementById("myDiv").getElementsByClassName("selected");
```

Метод `getElementsByClassName()` возвращает только те элементы, которые находятся в поддереве опорного элемента. Если он вызывается для объекта `document`, возвращаются все элементы указанных классов в документе.

С помощью этого метода можно подключать события к элементам, распознавая их по классу, а не по идентификатору или по имени тега. Помните, что он возвращает объект `NodeList`, а значит, ему присущи те же проблемы с быстродействием, что и методу `getElementsByTagName()` и другим DOM-методам, которые возвращают `NodeList`.

Метод `getElementsByClassName()` реализован в Internet Explorer 9+, Firefox 3+, Safari 3.1+, Chrome и Opera 9.5+.

Свойство classList

Свойство `className` используется для добавления, удаления и замены имен классов. Оно содержит строку, которую нужно задавать целиком при каждом изменении, даже самом малом. Рассмотрим пример:

```
<div class="bd user disabled">...</div>
```

Этому элементу `<div>` назначены три класса. Чтобы удалить один из них, нужно разделить атрибут `class` на отдельные классы, убрать нежелательный класс, а затем составить новую строку из оставшихся классов, например:

```
// удаление класса "user"

// сначала получаем список имен классов
var classNames = div.className.split(/\s+/);

// ищем имя удаляемого класса
```

```
var pos = -1,
    i,
    len;
for (i=0, len=classNames.length; i < len; i++){
    if (classNames[i] == "user"){
        pos = i;
        break;
    }
}

// удаляем имя класса
classNames.splice(i,1);

// составляем строку из оставшихся имен классов
div.className = classNames.join(" ");
```

Весь этот код необходим для удаления класса `"user"` из атрибута `class` элемента `<div>`. Подобный алгоритм нужно использовать и для замены или поиска классов. При добавлении классов путем конкатенации их имен нужно следить за тем, чтобы они не повторялись. Многие JavaScript-библиотеки содержат методы, помогающие решать эти задачи.

Чтобы упростить работу с именами классов и сделать ее более безопасной, в HTML5 ко всем элементам добавлено свойство `classList`, которое является экземпляром новой коллекции `DOMTokenList`. Как и другие DOM-коллекции, `DOMTokenList` имеет свойство `length`, содержащее количество элементов в коллекции, и поддерживает доступ к отдельным элементам с помощью метода `item()` или скобочной нотации. Кроме того, у нее есть несколько дополнительных методов:

- `add(`*значение*`)` — добавляет указанное строковое значение в список (если значение уже существует, оно не добавляется);
- `contains(`*значение*`)` — указывает, есть ли в списке указанное значение (возвращает `true`, если есть, и `false` в противном случае);
- `remove(`*значение*`)` — удаляет указанное строковое значение из списка;
- `toggle(`*значение*`)` — удаляет указанное значение, если оно уже есть в списке, и добавляет значение, если оно отсутствует.

Таким образом, можно заменить весь код из предыдущего примера одной строкой:

```
div.classList.remove("user");
```

Этот код гарантирует, что изменение не повлияет на остальные имена классов. Другие методы также значительно упрощают базовые операции с классами, например:

```
// удаление класса "disabled"
div.classList.remove("disabled");

// добавление класса "current"
div.classList.add("current");
```

```
// переключение класса "user"
div.classList.toggle("user");

// идентификация классов элемента
if (div.classList.contains("bd") && !div.classList.contains("disabled")){
    // какие-то действия
}

// перебор имен классов
for (var i=0, len=div.classList.length; i < len; i++){
    doSomething(div.classList[i]);
}
```

Свойство `classList` делает ненужным свойство `className`, если только вы не собираетесь полностью удалить или перезаписать атрибут `class` элемента. Свойство `classList` реализовано в Firefox 3.6+ и Chrome.

Управление фокусом

В HTML5 добавлены средства, помогающие управлять выделением DOM-элементов. Прежде всего, это свойство `document.activeElement`, которое всегда содержит указатель на выделенный DOM-элемент. Элемент может быть выделен автоматически во время загрузки страницы, в результате действий пользователя (как правило, при нажатии клавиши табуляции) или программно методом `focus()`, например:

```
var button = document.getElementById("myButton");
button.focus();
alert(document.activeElement === button);    // true
```

По умолчанию при загрузке документа в первый раз свойству `document.activeElement` присваивается значение `document.body`, но до полной загрузки документа оно имеет значение `null`.

Второе новшество — метод `document.hasFocus()`, который возвращает логическое значение, указывающее, содержит ли документ выделенный элемент:

```
var button = document.getElementById("myButton");
button.focus();
alert(document.hasFocus());        // true
```

С помощью этого метода можно узнать, взаимодействует ли пользователь со страницей.

Возможности идентифицировать выделенный элемент и узнать, есть ли он в документе, крайне важны для эффективного доступа к веб-приложению. Правильное управление фокусом делает работу с приложением более удобной, и прямое определение выделенного элемента намного предпочтительнее в этом плане, чем прежние бессистемные подходы.

Эти свойства реализованы в Internet Explorer 4+, Firefox 3+, Safari 4+, Chrome и Opera 8+.

Изменения типа HTMLDocument

В HTML5 также расширен тип `HTMLDocument`. Как и другие DOM-расширения, определенные в HTML5, его изменения основаны на фирменных расширениях, реализованных во многих браузерах. Таким образом, хотя стандартизировать расширения стали сравнительно недавно, в ряде браузеров они доступны не первый день.

Свойство readyState

В Internet Explorer 4 было представлено свойство `readyState` объекта `document`. Позднее оно было добавлено в другие браузеры и в итоге вошло в спецификацию HTML5. Оно может иметь следующие значения:

- `loading` — документ загружается;
- `complete` — документ полностью загружен.

Свойство `document.readyState` полезно как индикатор загрузки документа. Пока оно широко не применялось, вместо него задействовали обработчик события `onload`, в котором устанавливали флаг, указывающий, что документ загружен. Используется это свойство следующим образом:

```
if (document.readyState == "complete"){
    // какие-то действия
}
```

Свойство `readyState` реализовано в Internet Explorer 4+, Firefox 3.6+, Safari, Chrome и Opera 9+.

Режим совместимости

С выпуском Internet Explorer 6 и появлением возможности визуализировать документ в стандартном режиме или режиме совместимости потребовалось определять, в каком режиме браузер отображает страницу. В Internet Explorer исключительно для этого к объекту `document` было добавлено свойство `compatMode`. В стандартном режиме оно имеет значение `"CSS1Compat"`, а в режиме совместимости — `"BackCompat"`:

```
if (document.compatMode == "CSS1Compat"){
    alert("Standards mode");   // "Стандартный режим"
} else {
    alert("Quirks mode");      // "Режим совместимости"
}
```

Позднее это свойство было реализовано в Firefox, Safari 3.1+, Opera и Chrome, а затем стандартизировано в HTML5.

Свойство head

В HTML5 представлено свойство `document.head`, которое указывает на элемент `<head>` документа, логически дополняя свойство `document.body`. Его можно использовать как альтернативу старому способу получения ссылки на элемент `<head>`:

```
var head = document.head || document.getElementsByTagName("head")[0];
```

Если свойство `document.head` недоступно, этот код вызывает страховочный метод `getElementsByTagName()`.

Свойство `document.head` реализовано в Chrome и Safari 5.

Свойства кодировки

HTML5 описывает несколько свойств для работы с кодировкой документа. Свойство `charset` указывает фактическую кодировку документа и позволяет задать новую. По умолчанию оно имеет значение `"UTF-16"`, которое можно изменить с помощью элементов `<meta>`, заголовков ответа или непосредственно, например:

```
alert(document.charset);    // "UTF-16"
document.charset = "UTF-8";
```

Свойство `defaultCharset` указывает, какой по умолчанию должна быть кодировка, определяя ее по исходным параметрам браузера и системы. Если изменить кодировку, предлагаемую по умолчанию, значения `charset` и `defaultCharset` будут разными, например:

```
if (document.charset != document.defaultCharset){
    // Выбрана пользовательская кодировка
    alert("Custom character set being used.");
}
```

Эти свойства помогают управлять кодировкой документа и при грамотном применении обеспечивают правильное отображение страницы или приложения.

Свойство `document.charset` поддерживается в Internet Explorer, Firefox, Safari, Opera и Chrome, а свойство `document.defaultCharset` — в Internet Explorer, Safari и Chrome.

Пользовательские атрибуты данных

В HTML5 можно использовать нестандартные атрибуты с префиксом `data-` для добавления сведений, не влияющих на визуализацию или семантику элементов. Имена этих атрибутов могут быть любыми, но должны начинаться с префикса `data-`, например:

```
<div id="myDiv" data-appId="12345" data-myname="Nicholas"></div>
```

Такие атрибуты доступны через свойство `dataset` элемента. Оно содержит экземпляр типа `DOMStringMap`, в котором хранятся пары имен и значений. Каждый атрибут формата `data-`*имя* представляется одноименным свойством без префикса `data-` (например, атрибуту `data-myname` соответствует свойство `myname`):

```
// методы в этом примере используются исключительно для демонстрации

var div = document.getElementById("myDiv");

// получение значений
var appId = div.dataset.appId;
var myName = div.dataset.myname;

// задание значений
div.dataset.appId = 23456;
div.dataset.myname = "Michael";

// существует ли значение "myname"?
if (div.dataset.myname){
    alert("Hello, " + div.dataset.myname);
}
```

Пользовательские атрибуты данных могут связать с элементом какую-то невизуальную информацию. Их часто используют для отслеживания ссылок, а также для идентификации частей страницы в гибридных веб-приложениях.

Пока что эта возможность реализована только в Firefox 6+ и Chrome.

Вставка разметки

Хотя DOM обеспечивает детализированный контроль над узлами в документе, вставлять новые элементы HTML-разметки с помощью DOM неудобно. Вместо того чтобы создавать последовательность DOM-узлов и связывать их в правильном порядке, гораздо проще (и быстрее) вставить в документ строку HTML-кода. Для этого в HTML5 определены описываемые далее DOM-расширения.

Свойство innerHTML

В режиме чтения свойство `innerHTML` возвращает HTML-код, представляющий все дочерние узлы элемента, в том числе элементы, комментарии и текстовые узлы. При записи свойства `innerHTML` все дочерние узлы элемента заменяются новым DOM-поддеревом. Рассмотрим следующий HTML-код:

```
<div id="content">
    <p>This is a <strong>paragraph</strong> with a list following it.</p>
    <ul>
        <li>Item 1</li>
        <li>Item 2</li>
        <li>Item 3</li>
    </ul>
</div>
```

Для элемента в этом примере свойство innerHTML возвращает следующую строку:

```
<p>This is a <strong>paragraph</strong> with a list following it.</p>
<ul>
    <li>Item 1</li>
    <li>Item 2</li>
    <li>Item 3</li>
</ul>
```

Точный текст, возвращаемый свойством innerHTML, зависит от браузера. Internet Explorer и Opera обычно преобразуют все теги в верхний регистр, а Safari, Chrome и Firefox возвращают HTML без изменений, с пробелами и отступами. Не полагайтесь на то, что значение свойства innerHTML будет одинаковым во всех браузерах.

В режиме записи свойство innerHTML составляет из назначенной ему строки DOM-поддерево и заменяет им все дочерние узлы элемента. Поскольку строка интерпретируется как HTML-код, все теги в ней преобразуются в элементы стандартным для браузера способом (который также зависит от браузера). Если строка не содержит HTML-тегов, в свойстве сохраняется обычный текст:

```
div.innerHTML = "Hello world!";
```

Если свойству innerHTML назначается строка с HTML-тегами, выполняется синтаксический анализ, например:

```
div.innerHTML = "Hello & welcome, <b>\"reader\"!</b>";
```

Эта операция дает следующий результат:

```
<div id="content">Hello &amp; welcome, <b>&quot;reader&quot;!</b></div>
```

После задания свойства innerHTML новые узлы можно использовать так же, как и любые другие узлы в документе.

При задании свойства innerHTML браузер преобразует указанную HTML-строку в соответствующее DOM-поддерево. При последующем чтении того же свойства innerHTML обычно возвращается другая строка, потому что она является результатом сериализации DOM-поддерева, созданного из первоначальной строки.

У свойства innerHTML есть некоторые ограничения. Так, большинство браузеров не поддерживают выполнение кода в элементах <script>, вставленных через свойство innerHTML. Это возможно лишь в Internet Explorer 8 и более ранних версий, если указан атрибут defer и элементу <script> предшествует так называемый *визуальный элемент* (scoped element). <script> и <style>, а также комментарии к визуальным элементам не относятся, поскольку они не отображаются на странице. Internet Explorer игнорирует такие элементы в начале строк, вставленных с помощью свойства innerHTML, то есть следующий код не сработает:

```
div.innerHTML = "<script defer>alert('hi');<\/script>";  // не работает
```

Здесь свойству innerHTML присваивается значение, которое начинается с невизуального элемента, поэтому вся строка становится пустой. Чтобы этот сценарий работал должным образом, нужно добавить перед ним визуальный элемент, например текстовый узел, или элемент без закрывающего тега, такой как <input>. Все следующие варианты работают правильно:

```
div.innerHTML = "_<script defer>alert('hi');<\/script>";
div.innerHTML = "<div> </div><script defer>alert('hi');<\/script>";
div.innerHTML =
    "<input type=\"hidden\"><script defer>alert('hi');<\/script>";
```

Первая строка добавляет перед элементом <script> текстовый узел, который, возможно, позднее потребуется удалить, чтобы не искажать вид страницы. Во втором примере с этой же целью используется элемент с неразрывным пробелом. Пустого элемента недостаточно — он должен иметь некоторое содержимое, чтобы был создан текстовый узел. Этот узел также может потребоваться удалить для восстановления правильной разметки. Наконец, в третьем примере используется скрытое поле <input>. Поскольку оно не влияет на разметку, обычно это оптимальный выбор.

Большая часть браузеров поддерживает вставку элементов <style> через свойство innerHTML:

```
div.innerHTML =
    "<style type=\"text/css\">body {background-color: red; }</style>";
```

В Internet Explorer 8 и более ранних версий элементу <style> должен предшествовать визуальный элемент, например:

```
div.innerHTML =
    "_<style type=\"text/css\">body {background-color: red; }</style>";
div.removeChild(div.firstChild);
```

Свойство innerHTML недоступно для элементов <col>, <colgroup>, <frameset>, <head>, <html>, <style>, <table>, <tbody>, <thead>, <tfoot> и <tr>. В Internet Explorer 8 и более ранних версий его также нет у элемента <title>.

Firefox предъявляет более строгие требования к свойству innerHTML в XHTML с типом контента application/xhtml+xml. Если попытаться назначить ему XHTML-код с нарушениями формата, операция просто игнорируется, а ошибка не возвращается.

Если свойству innerHTML нужно назначить HTML-код из внешнего источника, его следует предварительно санировать. В Internet Explorer 8 для этого добавлен метод window.toStaticHTML(). Он принимает HTML-строку и удаляет из нее все узлы сценариев и атрибуты обработчиков событий, возвращая санированный HTML-код, например:

```
var text = "<a href=\"#\" onclick=\"alert('hi')\">Click Me</a>";
var sanitized = window.toStaticHTML(text);    // только Internet Explorer 8
alert(sanitized);    // "<a href=\"#\">Click Me</a>"
```

В этом примере метод `toStaticHTML()` удаляет из HTML-ссылки атрибут `onclick`. Internet Explorer 8 — единственный браузер, в который встроен этот метод, поэтому по мере возможности текст, назначаемый свойству `innerHTML`, следует проверять вручную.

Свойство outerHTML

При чтении свойства `outerHTML` оно возвращает HTML-код элемента, которому принадлежит, и всех его дочерних узлов. При записи оно заменяет узел, которому принадлежит, DOM-поддеревом, соответствующим полученной HTML-строке. Рассмотрим следующий HTML-код:

Листинг OuterHTMLExample01.htm


```
<div id="content">
    <p>This is a <strong>paragraph</strong> with a list following it.</p>
    <ul>
        <li>Item 1</li>
        <li>Item 2</li>
        <li>Item 3</li>
    </ul>
</div>
```

Если вызвать свойство `outerHTML` этого элемента `<div>`, будет возвращен точно такой же код, в том числе сам элемент `<div>`. Имейте в виду, что из-за особенностей синтаксического анализа и интерпретации HTML-кода в разных браузерах результаты могут различаться (наблюдаются те же различия, что и при работе со свойством `innerHTML`).

Задается свойство `outerHTML` следующим образом:

```
div.outerHTML = "<p>This is a paragraph.</p>";
```

Этот код эквивалентен следующему DOM-коду:

```
var p = document.createElement("p");
p.appendChild(document.createTextNode("This is a paragraph."));
div.parentNode.replaceChild(p, div);
```

В этом примере новый элемент `<p>` заменяет исходный элемент `<div>` в DOM-дереве.

Свойство `outerHTML` поддерживается в Internet Explorer 4+, Safari 4+, Chrome и Opera 8+. Firefox до версии 7 его все еще не поддерживает.

Метод insertAdjacentHTML()

Вставлять разметку можно с помощью метода `insertAdjacentHTML()`, который также появился в Internet Explorer. Он принимает два аргумента: позицию, в которой нужно вставить разметку, и HTML-код. Первым аргументом может быть одно из следующих значений:

- `"beforebegin"` — вставляет HTML-код непосредственно перед элементом на том же уровне;
- `"afterbegin"` — вставляет HTML-код внутри элемента как новый дочерний узел или как несколько дочерних узлов перед первым уже существующим;
- `"beforeend"` — вставляет HTML-код внутри элемента как новый дочерний узел или как несколько дочерних узлов вслед за последним уже существующим;
- `"afterend"` — вставляет HTML-код сразу после элемента на том же уровне.

Все эти значения нечувствительны к регистру. Второй аргумент метода обрабатывается как HTML-строка (подобно свойствам `innerHTML/outerHTML`). Если интерпретировать его не удается, возникает ошибка. Вот некоторые примеры:

```
// вставка узла перед элементом на том же уровне
element.insertAdjacentHTML("beforebegin", "<p>Hello world!</p>");

// вставка узла в качестве первого дочернего
element.insertAdjacentHTML("afterbegin", "<p>Hello world!</p>");

// вставка узла в качестве последнего дочернего
element.insertAdjacentHTML("beforeend", "<p>Hello world!</p>");

// вставка узла после элемента на том же уровне
element.insertAdjacentHTML("afterend", "<p>Hello world!</p>");
```

Метод `insertAdjacentHTML()` поддерживается в Internet Explorer, Firefox 8+, Safari, Opera и Chrome.

Проблемы с памятью и быстродействием

Замена дочерних узлов с помощью методов, описываемых в этом разделе, может вызывать в браузерах проблемы с памятью, особенно в Internet Explorer. Проблемы возникают при удалении элементов поддерева, которым назначены обработчики событий или другие JavaScript-объекты. Если у элемента есть обработчик событий (или свойство, содержащее JavaScript-объект) и этот элемент удаляется из дерева документа с помощью одного из описанных свойств, связь между элементом и обработчиком событий остается в памяти. Если это повторяется много раз, потребление памяти существенно увеличивается. При использовании свойств `innerHTML`, `outerHTML` и метода `insertAdjacentHTML()` рекомендуется вручную удалять все обработчики событий и свойства-объекты у элементов, которые будут удалены (обработчики событий обсуждаются в главе 13).

Тем не менее обычно использовать эти свойства выгодно, особенно `innerHTML`. Если нужно вставить много нового HTML-кода, эффективнее сделать это с помощью `innerHTML`, а не многочисленных DOM-операций, создающих узлы и связи между ними. Это объясняется тем, что при назначении строки свойству `innerHTML` (или

outerHTML) всегда создается HTML-анализатор. Он реализуется на уровне браузера (часто на C++) и работает гораздо быстрее, чем JavaScript. И все же создание и уничтожение HTML-анализатора требует ресурсов, так что увлекаться свойствами innerHTML и outerHTML не следует. Например, следующий код создает с помощью свойства innerHTML несколько элементов списка:

```
for (var i=0, len=values.length; i < len; i++){
    ul.innerHTML += "<li>" + values[i] + "</li>";    // не делайте так!
}
```

Этот код неэффективен, потому что свойство innerHTML задается при каждой итерации цикла. Хуже того: оно еще считывается при каждой итерации, что удваивает расходы. Лучше создать строку отдельно и назначить ее свойству innerHTML только один раз в конце:

```
var itemsHtml = "";
for (var i=0, len=values.length; i < len; i++){
    itemsHtml += "<li>" + values[i] + "</li>";
}
ul.innerHTML = itemsHtml;
```

Этот код более эффективен, потому что доступ к свойству innerHTML выполняется один раз.

Метод scrollIntoView()

Спецификация DOM не определяет средства прокрутки областей страницы. Чтобы восполнить этот пробел, разработчики браузеров реализовали несколько разных методов прокрутки, из которых в HTML5 был добавлен только scrollIntoView().

Метод scrollIntoView() доступен для всех HTML-элементов; он прокручивает окно браузера или другой контейнер так, чтобы элемент был виден в области просмотра. Если аргумент метода имеет значение true или опущен, элемент прилегает к верхнему краю области просмотра (если это возможно), в противном случае — к ее нижнему краю, например:

```
// вывод элемента в область просмотра
document.forms[0].scrollIntoView();
```

Этот метод наиболее полезен, если нужно привлечь внимание пользователя к каким-либо изменениям на странице. Отметим, что при установке фокуса для элемента браузер также при необходимости прокручивает страницу, чтобы отобразить выделенный элемент в области просмотра.

Метод scrollIntoView() поддерживается в Internet Explorer, Firefox, Safari, Opera и Chrome.

Фирменные расширения

Хотя производители браузеров понимают важность соблюдения стандартов, все они добавляли в DOM собственные расширения, если считали, что в ней чего-то не хватает. На первый взгляд, это может показаться плохой идеей, однако фирменные расширения предоставили сообществу веб-разработчиков много возможностей, которые позднее были формализованы в HTML5 и других стандартах.

В то же время многие фирменные DOM-расширения по-прежнему не отражены в стандартах. Вполне возможно, что какие-то из них будут стандартизированы, но пока они реализованы только в некоторых браузерах.

Режим документа

В Internet Explorer 8 была представлена концепция *режимов документа* (document mode), которые определяют, какие возможности доступны странице. Иначе говоря, на странице можно использовать только ограниченный CSS- и JavaScript-функционал и только некоторые способы обработки типов документов. Internet Explorer 9 поддерживает четыре режима документа.

- `Internet Explorer 5` — страница визуализируется в режиме совместимости (этот режим предлагается по умолчанию в Internet Explorer 5). Новые возможности Internet Explorer 8 и более поздних версий недоступны.
- `Internet Explorer 7` — страница визуализируется в стандартном режиме Internet Explorer 7. Новые возможности Internet Explorer 8 и более поздних версий недоступны.
- `Internet Explorer 8` — страница визуализируется в стандартном режиме Internet Explorer 8. Доступны новые возможности Internet Explorer 8, такие как Selectors API, дополнительные селекторы CSS 2, а также некоторые возможности CSS3 и HTML5. Возможности Internet Explorer 9 недоступны.
- `Internet Explorer 9` — страница визуализируется в стандартном режиме Internet Explorer 9. Доступны новые возможности Internet Explorer 9, такие как ECMAScript 5, полная поддержка CSS3 и расширенные возможности HTML5. Этот режим документа обеспечивает наиболее мощный функционал.

Концепция режима документа очень важна для понимания работы Internet Explorer 8 и более поздних версий.

Вы можете задать режим документа с помощью HTTP-заголовка `X-UA-Compatible` или эквивалентного тега `<meta>`:

```
<meta http-equiv="X-UA-Compatible" content="IE=версия IE">
```

Версия Internet Explorer может иметь одно из указанных далее значений, которые не всегда соответствуют какому-либо из режимов документа.

- `Edge` — документ всегда переводится в наиболее новый режим документа из доступных. Тип документа игнорируется. В Internet Explorer 8 всегда включается стандартный режим Internet Explorer 8, а в Internet Explorer 9 — стандартный режим Internet Explorer 9.
- `EmulateIE9` — если указан тип документа, включается стандартный режим Internet Explorer 9, иначе задается режим Internet Explorer 5.
- `EmulateIE8` — если указан тип документа, включается стандартный режим Internet Explorer 8, иначе задается режим Internet Explorer 5.
- `EmulateIE7` — если указан тип документа, включается стандартный режим Internet Explorer 7, иначе задается режим Internet Explorer 5.
- `9` — включается стандартный режим Internet Explorer 9, тип документа игнорируется.
- `8` — включается стандартный режим Internet Explorer 8, тип документа игнорируется.
- `7` — включается стандартный режим Internet Explorer 7, тип документа игнорируется.
- `5` — включается режим Internet Explorer 5, тип документа игнорируется.

Например, выбрать режим документа Internet Explorer 7 можно следующим образом:

```
<meta http-equiv="X-UA-Compatible" content="IE=EmulateIE7">
```

А чтобы использовать стандартный режим документа Internet Explorer 7 независимо от типа документа, добавьте в страницу такой код:

```
<meta http-equiv="X-UA-Compatible" content="IE=7">
```

Задавать поле `X-UA-Compatible` на страницах необязательно. По умолчанию браузер выбирает режим совместимости или наилучший стандартный режим на основе типа документа.

С помощью свойства `document.documentMode`, добавленного в Internet Explorer 8, можно определить текущий режим документа конкретной страницы (в Internet Explorer 9 могут быть возвращены значения `5`, `7`, `8` и `9`):

```
var mode = document.documentMode;
```

Это свойство позволяет получить некоторое представление о том, как будет работать страница. Оно доступно во всех режимах документа.

Свойство children

Различия в интерпретации свободного пространства в текстовых узлах в Internet Explorer до версии 9 и других браузерах стали поводом для создания свойства

children. Это коллекция HTMLCollection, содержащая только те дочерние узлы элемента, которые сами являются элементами. Если все дочерние узлы элемента представляют собой элементы, содержимое свойств children и childNodes не различается. Доступ к свойству children осуществляется так:

```
var childCount = element.children.length;
var firstChild = element.children[0];
```

Свойство children поддерживается в Internet Explorer 5, Firefox 3.5, Safari 2 (с дефектами), Safari 3 (полностью), Opera 8 и Chrome (во всех версиях). Internet Explorer 8 и более ранних версий (но не Internet Explorer 9+) возвращает в коллекции children еще и комментарии.

Метод contains()

Часто требуется определить, является ли один узел потомком другого. Метод contains() позволяет выяснить это без обхода дерева DOM-документа. Он вызывается для узла, с которого нужно начать поиск, и принимает в качестве аргумента предполагаемый узел-потомок. Если переданный в метод узел входит в число потомков опорного узла, метод возвращает true, иначе — false, например:

```
alert(document.documentElement.contains(document.body));    // true
```

Этот код проверяет, является ли элемент <body> потомком элемента <html>, что действительно имеет место во всех правильных HTML-страницах. Метод contains() поддерживается в Internet Explorer, Firefox 9+, Safari, Opera и Chrome.

Определить отношения между узлами можно также с помощью метода compareDocumentPosition() из DOM Level 3, который поддерживается в Internet Explorer 9+, Firefox, Safari, Opera 9.5+ и Chrome. Сведения об отношениях узлов возвращаются в виде битовой маски, отдельные биты которой описаны в таблице.

Маска	Отношения между узлами
1	Узлы не связаны (переданный в метод узел отсутствует в документе)
2	Отношение «предшествует» (переданный в метод узел располагается в DOM-дереве до опорного узла)
4	Отношение «следует» (переданный в метод узел располагается в DOM-дереве после опорного узла)
8	Отношение «содержит» (переданный в метод узел является предком опорного узла)
16	Отношение «содержится» (переданный в метод узел является потомком опорного узла)

С помощью маски 16 можно имитировать метод `contains()`. Для этого к маске и результату вызова метода `compareDocumentPosition()` нужно применить поразрядный оператор И:

```
var result =
    document.documentElement.compareDocumentPosition(document.body);
alert(!!(result & 16));
```

Переменная `result` в этом примере получает значение 20 (4 за отношение «следует» и 16 за «содержится»). Применение поразрядного оператора И к битовой маске 16 и результату возвращает ненулевое число, которое затем преобразуется в логическое значение `true` с помощью двух операторов НЕ.

Применив распознавание браузера и возможностей, можно создать универсальную функцию `contains`:

Листинг ContainsExample02.htm


```
function contains(refNode, otherNode){
    if (typeof refNode.contains == "function" &&
            (!client.engine.webkit || client.engine.webkit >= 522)){
        return refNode.contains(otherNode);
    } else if (typeof refNode.compareDocumentPosition == "function"){
        return !!(refNode.compareDocumentPosition(otherNode) & 16);
    } else {
        var node = otherNode.parentNode;
        do {
            if (node === refNode){
                return true;
            } else {
                node = node.parentNode;
            }
        } while (node !== null);
        return false;
    }
}
```

Эта функция объединяет три способа определения того, является ли один узел потомком другого. В качестве аргументов она принимает опорный и проверяемый узлы. Первым делом функция выясняет, есть ли у опорного узла `refNode` метод `contains()` (распознавание возможности) и какая версия WebKit используется. Если метод `contains()` доступен, а визуализатор отличается от WebKit (`!client.engine.webkit`), выполняется следующая инструкция. Она выполняется также, если браузер основан на WebKit версии 522 или более поздней (это соответствует Safari 3+). В WebKit до версии 522 метод `contains()` работает неправильно.

Если первый способ не поддерживается, функция пробует воспользоваться методом `compareDocumentPosition()`, а если и он недоступен, выполняется рекурсивный проход к корню дерева от узла `otherNode`, при этом каждый очередной узел `parentNode` сравнивается с узлом `refNode`. В корне дерева документа свойство `parentNode` окажется

равным null, и цикл завершится. Этот страховочный код нужен для поддержки старых версий Safari.

Вставка разметки

Кроме свойств innerHTML и outerHTML, добавленных в HTML5 из Internet Explorer, для вставки разметки можно использовать свойства innerText и outerText, которые не вошли в HTML5.

Свойство innerText

Свойство innerText предназначено для работы со всем текстовым контентом элемента независимо от того, насколько глубоко в поддереве находится этот текст. При чтении свойства innerText значения всех текстовых узлов в поддереве объединяются в строку с использованием поиска в глубину. При записи свойства innerText все дочерние узлы элемента заменяются текстовым узлом, содержащим указанное значение. Рассмотрим следующий HTML-код:

Листинг InnerTextExample01.htm


```
<div id="content">
    <p>This is a <strong>paragraph</strong> with a list following it.</p>
    <ul>
        <li>Item 1</li>
        <li>Item 2</li>
        <li>Item 3</li>
    </ul>
</div>
```

В этом примере свойство innerText возвратило бы для элемента следующую строку:

```
This is a paragraph with a list following it.
Item 1
Item 2
Item 3
```

Имейте в виду, что браузеры обрабатывают свободное пространство по-разному, так что форматирование может не включать отступы, присутствующие в исходном HTML-коде.

Задать содержимое элемента с помощью свойства innerText можно следующим образом:

Листинг InnerTextExample02.htm


```
div.innerText = "Hello world!";
```

При выполнении этой строки HTML-код страницы изменится:

```
<div id="content">Hello world!</div>
```

Задание свойства `innerText` удаляет все существующие дочерние узлы, полностью изменяя DOM-поддерево. Отметим также, что при этом кодируются все знаки HTML-синтаксиса в тексте («меньше», «больше», кавычки и амперсанды), например:

```
div.innerText = "Hello & welcome, <b>\"reader\"!</b>";
```

Листинг InnerTextExample03.htm

В результате получается следующий HTML-код:

```
<div id="content"> Hello &amp;
    welcome, &lt;b&gt;&quot;reader&quot;!&lt;/b&gt;</div>
```

Задание свойства `innerText` всегда дает в результате единственный дочерний текстовый узел, поэтому HTML-кодирование текста необходимо для его правильной обработки. С помощью свойства `innerText` можно также легко удалить HTML-теги, присвоив его самому себе:

```
div.innerText = div.innerText;
```

Если выполнить этот код, в элементе-контейнере останется только текстовый контент.

Свойство `innerText` поддерживается в Internet Explorer 4+, Safari 3+, Opera 8+ и Chrome. Firefox не поддерживает его, но предоставляет эквивалентное свойство `textContent`, которое определено в DOM Level 3 и доступно также в Internet Explorer 9+, Safari 3+, Opera 10+ и Chrome. Чтобы код работал во всех браузерах, можно использовать следующие функции, проверяющие доступность свойств:

Листинг InnerTextExample05.htm

```
function getInnerText(element){
    return (typeof element.textContent == "string") ?
        element.textContent : element.innerText;
}

function setInnerText(element, text){
    if (typeof element.textContent == "string"){
        element.textContent = text;
    } else {
        element.innerText = text;
    }
}
```

Каждая из этих функций проверяет, есть ли у переданного в нее элемента свойство `textContent`. Если да, выражение `typeof element.textContent` возвращает значение `"string"`. Если свойство `textContent` недоступно, в обеих функциях используется свойство `innerText`. Вот примеры их применения:

```
setInnerText(div, "Hello world!");
alert(getInnerText(div));    // "Hello world!"
```

Использование этих функций гарантирует выбор правильного свойства на основе возможностей браузера.

Значения, возвращаемые свойствами innerText и textContent, немного различаются. Свойство innerText пропускает встроенные блоки стилей и сценариев, тогда как textContent возвращает их вместе с другим текстом. Чтобы обойти различия браузеров, можно читать текст только из неглубоких DOM-поддеревьев и DOM-частей, в которых нет встроенных стилей и сценариев.

Свойство outerText

Свойство `outerText` аналогично свойству `innerText`, но в отличие от него включает узел, которому принадлежит. При чтении текстовых значений свойства `outerText` и `innerText` работают, в общем-то, одинаково, а в режиме записи сильно различаются. Вместо того чтобы заменять только дочерние узлы опорного элемента, свойство `outerText` заменяет весь элемент с дочерними узлами, например:

```
div.outerText = "Hello world!";
```

Эта строка кода эквивалентна двум следующим:

```
var text = document.createTextNode("Hello world!");
div.parentNode.replaceChild(text, div);
```

По сути, новый текстовый узел полностью заменяет элемент, для которого было задано свойство `outerText`. После этого исходный элемент в документе недоступен.

Свойство `outerText` поддерживается в Internet Explorer 4+, Safari 3+, Opera 8+ и Chrome. Из-за того что оно изменяет свой элемент, использовать его не рекомендуется.

Прокрутка

Как уже отмечалось, до HTML5 никаких спецификаций прокрутки не было. Метод `scrollIntoView()` был стандартизирован в HTML5, но есть еще несколько фирменных методов, доступных в разных браузерах. Каждый из представленных далее методов расширяет тип `HTMLElement`, а потому доступен для всех элементов.

- `scrollIntoViewIfNeeded(`*выравниваниеПоЦентру*`)` — прокручивает окно браузера или элемент-контейнер, чтобы элемент появился в области просмотра, но только при условии, что он не виден. Если элемент уже отображается в области просмотра, метод ничего не делает. Если необязательный аргумент имеет значение `true`, предпринимается попытка центрировать элемент в области просмотра. Этот метод реализован в Safari и Chrome.
- `scrollByLines(`*количествоСтрок*`)` — прокручивает содержимое элемента по высоте на указанное количество текстовых строк, которое может быть положительным или отрицательным. Этот метод реализован в Safari и Chrome.

- `scrollByPages(`*количествоСтраниц*`)` — прокручивает содержимое элемента по высоте на указанное количество страниц. Этот метод реализован в Safari и Chrome.

Имейте в виду, что методы `scrollIntoView()` и `scrollIntoViewIfNeeded()` выполняются для контейнера элемента, а методы `scrollByLines()` и `scrollByPages()` — для самого элемента. Вот примеры их использования:

```
// прокрутка тела документа на пять строк
document.body.scrollByLines(5);

// вывод элемента в область просмотра, если он не виден
document.images[0].scrollIntoViewIfNeeded();

// прокрутка тела документа на одну страницу в обратную сторону
document.body.scrollByPages(-1);
```

Поскольку `scrollIntoView()` — единственный метод, который поддерживается во всех браузерах, обычно используется только он.

Резюме

Стандартная модель DOM определяет базовый API для взаимодействия с XML- и HTML-документами, но существует несколько спецификаций, расширяющих ее возможности. Многие такие расширения основаны на фирменных разработках, которые по мере реализации аналогичных функций в других браузерах стали стандартами де-факто. В этой главе мы рассмотрели три спецификации.

- **Selectors** определяет методы `querySelector()` и `querySelectorAll()`, служащие для получения DOM-элементов на основе CSS-селекторов.
- **Element Traversal** описывает дополнительные свойства DOM-элементов, позволяющие легко переходить к ближайшим связанным элементам. Эти возможности потребовались из-за различий в обработке свободного пространства между DOM-элементами.
- **HTML5** определяет целый ряд расширений стандартной DOM, таких как свойство `innerHTML`, средства управления фокусом, кодировками, прокруткой и т. д.

В настоящее время DOM-расширений немного, но по мере развития веб-технологий их количество будет расти. Удачные фирменные расширения могут со временем стать стандартами де-факто и войти в будущие версии спецификаций.

12 DOM Level 2 и 3

- Изменения в спецификациях Level 2 и 3
- DOM API для работы со стилями
- Обход и диапазоны DOM

Базовая структура HTML- и XML-документов, определенная в спецификации DOM Level 1, была расширена в DOM Level 2 и 3 интерактивными возможностями и улучшенными XML-механизмами. В результате на данный момент спецификации DOM Level 2 и 3 составляют следующие связанные модули, описывающие конкретные подмножества DOM:

- **DOM Core** — дополняет DOM Level 1 Core, добавляя к узлам методы и свойства;
- **DOM Views** — определяет для документа разные представления на основе стилей;
- **DOM Events** — обеспечивает интерактивность DOM-документов с помощью событий;
- **DOM Style** — описывает программный доступ к CSS-стилям и их изменение;
- **DOM Traversal and Range** — предоставляет новые интерфейсы для обхода DOM-документа и выделения его частей;
- **DOM HTML** — расширяет HTML Level 1 новыми свойствами, методами и интерфейсами.

В данной главе мы обсудим все эти модули, кроме событий DOM, которые подробно описаны в главе 13.

DOM Level 3 содержит также модули XPath и Load and Save, описываемые в главе 18.

Изменения DOM

Спецификации DOM Level 2 и 3 Core были разработаны для того, чтобы реализовать в DOM API все требования языка XML и улучшить обработку ошибок и распознавание функциональных возможностей. В основном это сводится к поддержке XML-пространств имен. DOM Level 2 Core не определяет никаких новых типов, а просто добавляет новые методы и свойства к типам DOM Level 1. DOM Level 3 Core расширяет доступные типы и определяет несколько новых.

Модули DOM Views и DOM HTML, которые также содержат ряд новых свойств и методов, довольно малы, и поэтому мы обсудим их вместе с DOM Core. Узнать, поддерживает ли браузер эти части DOM, можно следующим образом:

```
var supportsDOM2Core = document.implementation.hasFeature("Core", "2.0");
var supportsDOM3Core = document.implementation.hasFeature("Core", "3.0");
var supportsDOM2HTML = document.implementation.hasFeature("HTML", "2.0");
var supportsDOM2Views = document.implementation.hasFeature("Views", "2.0");
var supportsDOM2XML = document.implementation.hasFeature("XML", "2.0");
```

В этой главе описаны только те части DOM, которые уже реализованы в браузерах.

XML-пространства имен

XML-пространства имен позволяют использовать элементы из разных XML-подобных языков в одном документе правильного формата без риска конфликтов имен. Технически XML-пространства имен не поддерживаются в HTML, поэтому примеры в этом разделе написаны на XHTML.

Пространства имен указываются с помощью атрибута `xmlns`. Языку XHTML соответствует пространство имен `http://www.w3.org/1999/xhtml`, которое нужно включать в элемент `<html>` любой XHTML-страницы правильного формата, например:

```
<html xmlns="http://www.w3.org/1999/xhtml">
    <head>
        <title>Example XHTML page</title>
    </head>
    <body>
        Hello world!
    </body>
</html>
```

В этом примере все элементы считаются по умолчанию частью пространства имен XHTML. Можно явно создать префикс для пространства имен XML, указав атрибут `xmlns`, двоеточие и префикс, например:

```
<xhtml:html xmlns:xhtml="http://www.w3.org/1999/xhtml">
    <xhtml:head>
```

```
        <xhtml:title>Example XHTML page</xhtml:title>
    </xhtml:head>
    <xhtml:body>
        Hello world!
    </xhtml:body>
</xhtml:html>
```

Здесь определяется пространство имен XHTML с префиксом `xhtml`, в результате все XHTML-элементы должны начинаться с этого префикса. Пространства имен можно также использовать с атрибутами для предотвращения конфликтов между языками:

```
<xhtml:html xmlns:xhtml="http://www.w3.org/1999/xhtml">
    <xhtml:head>
        <xhtml:title>Example XHTML page</xhtml:title>
    </xhtml:head>
    <xhtml:body xhtml:class="home">
        Hello world!
    </xhtml:body>
</xhtml:html>
```

Атрибуту `class` в этом примере предшествует префикс `xhtml`. Указывать пространства имен не требуется, если в документе используется только один язык, основанный на XML, но это полезно, если языков больше. Например, следующий документ содержит XHTML- и SVG-код:

```
<html xmlns="http://www.w3.org/1999/xhtml">
    <head>
        <title>Example XHTML page</title>
    </head>
    <body>
        <svg xmlns="http://www.w3.org/2000/svg" version="1.1"
             viewBox="0 0 100 100" style="width:100%; height:100%">
            <rect x="0" y="0" width="100" height="100" style="fill:red" />
        </svg>
    </body>
</html>
```

В этом примере для элемента `<svg>` задано отдельное пространство имен `http://www.w3.org/2000/svg`, которое указывает, что он является посторонним для документа. Оно применяется также ко всем дочерним элементам элемента `<svg>` и ко всем их атрибутам. Хотя технически это XHTML-документ, благодаря пространству имен SVG-код обрабатывается правильно.

При вызове методов, работающих с узлами такого документа, возникают интересные проблемы. Например, к какому пространству имен будут относиться создаваемые элементы? Какие элементы будут возвращены, если запросить теги с конкретным именем? К счастью, в DOM Level 2 Core для большинства методов DOM Level 1 определены версии, учитывающие пространства имен.

Изменения типа Node

В DOM Level 2 тип `Node` включает следующие новые свойства, учитывающие пространства имен:

- `localName` — имя узла без префикса пространства имен;
- `namespaceURI` — URI пространства имен узла или `null`, если свойство не задано;
- `prefix` — префикс пространства имен или `null`, если свойство не задано.

Если для узла задан префикс пространства имен, свойство `nodeName` эквивалентно значению `prefix + ":" + localName`. Рассмотрим пример:

Листинг NamespaceExample.xml


```
<html xmlns="http://www.w3.org/1999/xhtml">
    <head>
        <title>Example XHTML page</title>
    </head>
    <body>
        <s:svg xmlns:s="http://www.w3.org/2000/svg" version="1.1"
             viewBox="0 0 100 100" style="width:100%; height:100%">
            <s:rect x="0" y="0" width="100" height="100" style="fill:red" />
        </s:svg>
    </body>
</html>
```

У элемента `<html>` свойства `localName` и `tagName` имеют значение `"html"`, `namespaceURI` — значение `"http://www.w3.org/1999/xhtml"`, а `prefix` — значение `null`. У элемента `<s:svg>` свойство `localName` имеет значение `"svg"`, `tagName` — значение `"s:svg"`, `namespaceURI` — значение `"http://www.w3.org/2000/svg"`, а `prefix` — значение `"s"`.

В DOM Level 3 представлены следующие методы для работы с пространствами имен:

- `isDefaultNamespace(`*URIПространстваИмен*`)` — возвращает `true`, если аргумент представляет пространство имен, предлагаемое по умолчанию для узла;
- `lookupNamespaceURI(`*префикс*`)` — возвращает URI пространства имен для указанного префикса;
- `lookupPrefix(`*URIПространстваИмен*`)` — возвращает префикс для указанного URI пространства имен.

Для предыдущего примера эти методы возвратили бы следующие значения:

```
// возвращает true
alert(document.body.isDefaultNamespace("http://www.w3.org/1999/xhtml"));

// предполагается, что svg содержит ссылку на элемент <s:svg>
alert(svg.lookupPrefix("http://www.w3.org/2000/svg"));    // "s"
alert(svg.lookupNamespaceURI("s"));    // "http://www.w3.org/2000/svg"
```

Эти методы полезны, если имеется ссылка на узел и нужно определить, как он связан с остальным документом.

Изменения типа Document

В DOM Level 2 тип `Document` содержит следующие новые методы, специфичные для пространств имен:

- `createElementNS(`*URIПространстваИмен*, *имяТега*`)` — создает элемент с заданным именем тега в пространстве имен с указанным URI;
- `createAttributeNS(`*URIПространстваИмен*, *имяАтрибута*`)` — создает узел атрибута в пространстве имен с указанным URI;
- `getElementsByTagNameNS(`*URIПространстваИмен*, *имяТега*`)` — возвращает коллекцию `NodeList`, содержащую элементы с заданным именем тега из пространства имен с указанным URI.

Обратите внимание, что в эти методы передается URI нужного пространства имен (а не префикс), например:

```
// создание SVG-элемента
var svg = document.createElementNS("http://www.w3.org/2000/svg", "svg");

// создание атрибута random в пространстве имен
var att = document.createAttributeNS("http://www.somewhere.com", "random");

// получение всех XHTML-элементов
var elems =
    document.getElementsByTagNameNS("http://www.w3.org/1999/xhtml", "*");
```

Методы, специфичные для пространств имен, могут потребоваться, только если документ содержит более одного пространства имен.

Изменения типа Element

Изменения типа `Element` в DOM Level 2 Core связаны в основном с атрибутами. Он содержит следующие новые методы:

- `getAttributeNS(`*URIПространстваИмен*, *локальноеИмя*`)` — получает атрибут с заданным именем из пространства имен с указанным URI;
- `getAttributeNodeNS(`*URIПространстваИмен*, *локальноеИмя*`)` — получает узел атрибута с заданным именем из пространства имен с указанным URI;
- `getElementsByTagNameNS(`*URIПространстваИмен*, *имяТега*`)` — возвращает коллекцию `NodeList`, содержащую элементы из поддерева опорного элемента, которые имеют заданное имя тега и относятся к пространству имен с указанным URI;
- `hasAttributeNS(`*URIПространстваИмен*, *локальноеИмя*`)` — определяет, есть ли у элемента атрибут с заданным именем из пространства имен с указанным URI

(в DOM Level 2 Core есть также метод `hasAttribute()`, не учитывающий пространство имен);

- `removeAttributeNS(`*URIПространстваИмен*, *локальноеИмя*`)` — удаляет атрибут с заданным именем из пространства имен с указанным URI;
- `setAttributeNS(`*URIПространстваИмен*, *квалифицированноеИмя*, *значение*`)` — присваивает указанное значение атрибуту с заданным именем из пространства имен с указанным URI;
- `setAttributeNodeNS(`*узелАтрибута*`)` — задает узел атрибута из пространства имен с указанным URI.

Эти методы работают так же, как их аналоги из DOM Level 1, и отличаются от них только первым аргументом, которым у всех методов, кроме `setAttributeNodeNS()`, является URI пространства имен.

Изменения типа NamedNodeMap

Тип `NamedNodeMap` также содержит несколько новых методов для работы с пространствами имен. Поскольку он используется для представления атрибутов, эти методы в основном применяются к атрибутам:

- `getNamedItemNS(`*URIПространстваИмен*, *локальноеИмя*`)` — получает элемент с заданным именем из пространства имен с указанным URI;
- `removeNamedItemNS(`*URIПространстваИмен*, *локальноеИмя*`)` — удаляет элемент с заданным именем из пространства имен с указанным URI;
- `setNamedItemNS(`*узел*`)` — добавляет узел, которому уже должно быть назначено пространство имен.

Эти методы используются редко, потому что доступ к атрибутам обычно осуществляется через элемент.

Другие изменения

DOM Level 2 Core содержит также ряд других небольших изменений DOM, которые не имеют отношения к XML-пространствам имен и были внесены в основном для обеспечения гибкости и полноты API.

Изменения типа DocumentType

В тип `DocumentType` добавлены свойства `publicId`, `systemId` и `internalSubset`. Первые два из них представляют данные, которые содержатся в объявлении типа документа, но недоступны в DOM Level 1. Рассмотрим следующее объявление HTML-документа:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
    "http://www.w3.org/TR/html4/strict.dtd">
```

Здесь свойство publicId имеет значение "-//W3C//DTD HTML 4.01//EN", а systemId — значение "http://www.w3.org/TR/html4/strict.dtd". Если браузер поддерживает DOM Level 2, он должен быть способен выполнить следующий код:

```
alert(document.doctype.publicId);
alert(document.doctype.systemId);
```

Едва ли эти сведения когда-нибудь могут потребоваться на веб-страницах.

Свойство internalSubset обеспечивает доступ к любым дополнительным определениям в объявлении типа документа, например:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd"
[<!ELEMENT name (#PCDATA)>] >
```

Для этого кода свойство document.doctype.internalSubset возвращает "<!ELEMENT name (#PCDATA)>". Оно редко используется в HTML и чуть чаще в XML.

Изменения типа Document

Единственный новый метод типа Document, не связанный с пространствами имен, — это метод importNode(), который импортирует узел из одного документа в другой, чтобы его можно было добавить в структуру документа. Если помните, у каждого узла есть свойство ownerDocument, которое указывает, к какому документу относится узел. Если в метод вроде appendChild() передается узел, у которого свойство ownerDocument указывает на другой документ, происходит ошибка. При вызове метода importNode() для узла из другого документа возвращается новая версия узла, принадлежащая текущему документу.

Метод importNode() похож на метод cloneNode() элемента. Он принимает узел, который нужно клонировать, и логическое значение, указывающее, нужно ли также скопировать его дочерние узлы, и возвращает копию узла, например:

```
// импорт узла и всех его дочерних узлов
var newNode = document.importNode(oldNode, true);
document.body.appendChild(newNode);
```

Этот метод используется в основном с XML-документами (см. также главу 18).

В DOM Level 2 Views доступно новое свойство defaultView, содержащее указатель на окно (или фрейм), которому принадлежит документ. В спецификации Views не указано, когда могут быть доступны другие представления, так что это единственное добавленное свойство. Оно поддерживается во всех браузерах, кроме Internet Explorer 8 и более ранних версий, где доступно эквивалентное свойство parentWindow (оно доступно также в Opera). Определить окно, которому принадлежит документ, можно следующим образом:

```
var parentWindow = document.defaultView || document.parentWindow;
```

В DOM Level 2 Core к объекту `document.implementation` добавлены методы `createDocumentType()` и `createDocument()`. Первый из них создает узел `DocumentType`, принимая три аргумента: тип документа и свойства `publicId` и `systemId`. Например, следующий код создает тип документа HTML 4.01 Strict:

```
var doctype = document.implementation.createDocumentType("html",
                  "-//W3C//DTD HTML 4.01//EN",
                  "http://www.w3.org/TR/html4/strict.dtd");
```

Тип существующего документа изменить нельзя, поэтому метод `createDocumentType()` полезен только при создании документов, для чего можно использовать метод `createDocument()`. Он принимает три аргумента: URI пространства имен элемента документа, имя тега элемента документа и тип нового документа. Например, так можно создать пустой XML-документ:

```
var doc = document.implementation.createDocument("", "root", null);
```

Этот код создает документ с элементом документа `<root>` без пространства имен и типа документа. Создать XHTML-документ можно следующим образом:

```
var doctype = document.implementation.createDocumentType("html",
                  "-//W3C//DTD XHTML 1.0 Strict//EN",
                  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd");

var doc = document.implementation.createDocument(
    "http://www.w3.org/1999/xhtml", "html", doctype);
```

В этом примере создается XHTML-документ с соответствующими пространством имен и типом документа. Он содержит единственный элемент `<html>`, все остальное содержимое нужно добавлять.

Модуль DOM Level 2 HTML добавляет к объекту `document.implementation` метод `createHTMLDocument()`, который создает полный HTML-документ с элементами `<html>`, `<head>`, `<title>` и `<body>`. Он принимает заголовок нового документа (строковое содержимое элемента `<title>`) и возвращает HTML-документ:

Листинг CreateHTMLDocumentExample.htm


```
var htmldoc = document.implementation.createHTMLDocument("New Doc");
alert(htmldoc.title);              // "New Doc"
alert(typeof htmldoc.body);        // "object"
```

Объект, созданный с помощью метода `createHTMLDocument()`, имеет тип `HTMLDocument` и содержит все его свойства и методы, включая свойства `title` и `body`. Этот метод поддерживается в Internet Explorer 9+, Firefox 4+, Safari, Chrome и Opera.

Изменения типа Node

Единственное изменение типа `Node`, не связанное с пространствами имен, — это новый метод `isSupported()`. Как и метод `hasFeature()` объекта `document.implementation`,

представленный в DOM Level 1, он указывает возможности узла. Этот метод принимает два аргумента: имя компонента и его версию. Если компонент реализован и может использоваться узлом, метод `isSupported()` возвращает `true`, например:

```
if (document.body.isSupported("HTML", "2.0")){
    // какие-то действия с использованием модуля HTML из DOM Level 2
}
```

Этот метод не очень полезен, потому что ему присущ тот же недостаток, что и методу `hasFeature()`: реализующие его разработчики сами решают, возвращать ли `true` или `false` для конкретного компонента. Для определения того, доступна ли какая-то конкретная функциональная возможность, лучше использовать механизм распознавания возможностей.

Для сравнения узлов в DOM Level 3 представлены методы `isSameNode()` и `isEqualNode()`. Они принимают узел и возвращают `true`, если полученный и опорный узлы одинаковы или эквивалентны соответственно. Узлы одинаковы, если они ссылаются на один объект, и эквивалентны, если они имеют один тип, содержат равные свойства `nodeName`, `nodeValue` и т. д., причем их свойства `attributes` и `childNodes` эквивалентны (содержат одинаковые значения в аналогичных позициях), например:

```
var div1 = document.createElement("div");
div1.setAttribute("class", "box");

var div2 = document.createElement("div");
div2.setAttribute("class", "box");

alert(div1.isSameNode(div1));     // true
alert(div1.isEqualNode(div2));    // true
alert(div1.isSameNode(div2));     // false
```

Здесь создаются два элемента `<div>` с одними и теми же атрибутами. Эти элементы эквивалентны, но не одинаковы.

В DOM Level 3 добавлены также методы присоединения дополнительных данных к DOM-узлам. Метод `setUserData()`, который назначает данные узлу, принимает три аргумента: задаваемый ключ, фактические данные (которые могут иметь любой тип) и функцию-обработчик. Назначить данные узлу можно следующим образом:

```
document.body.setUserData("name", "Nicholas", function(){});
```

Получить эти данные можно с помощью метода `getUserData()`, передав в него тот же ключ:

```
var value = document.body.getUserData("name");
```

Функция-обработчик для метода `setUserData()` вызывается при клонировании, удалении, переименовании узла с данными или при его импорте в другой документ, позволяя указать, что нужно сделать с данными пользователя в каждом

из этих случаев. Она принимает пять аргументов: число, задающее тип операции (1 — клонирование; 2 — импорт; 3 — удаление; 4 — переименование), ключ данных, значение данных, исходный узел и целевой узел. Исходный узел равен `null`, если узел удаляется, а целевой узел равен `null` при всех операциях, кроме клонирования. Затем можно указать, как следует сохранить данные. Вот пример:

Листинг UserDataExample.htm


```
var div = document.createElement("div");
div.setUserData("name", "Nicholas",
                function(operation, key, value, src, dest){
    if (operation == 1){
        dest.setUserData(key, value, function(){}); }
});

var newDiv = div.cloneNode(true);
alert(newDiv.getUserData("name"));    // "Nicholas"
```

Созданному элементу `<div>` здесь назначаются некоторые данные, включая пользовательские. Затем этот элемент клонируется методом `cloneNode()`, при этом вызывается функция-обработчик и данные автоматически назначаются клону. В результате метод `getUserData()` возвращает для клона то же значение, которое было назначено оригиналу.

Изменения фреймов

Обычные и встроенные фреймы, представленные типами `HTMLFrameElement` и `HTMLIFrameElement` соответственно, имеют в DOM Level 2 HTML новое свойство `contentDocument`. Оно содержит указатель на объект `document` или содержимое фрейма. Ранее было невозможно получить объект `document` непосредственно через элемент — приходилось использовать коллекцию `frames`. Теперь это можно сделать с помощью нового свойства:

Листинг IFrameElementExample.htm


```
var iframe = document.getElementById("myIframe");
var iframeDoc = iframe.contentDocument;  // не работает в IE до версии 8
```

Свойство `contentDocument` является экземпляром типа `Document` и может быть использовано как любой другой HTML-документ. Оно поддерживается в Opera, Firefox, Safari и Chrome. Internet Explorer до версии 8 не поддерживает свойство `contentDocument` во фреймах, но предоставляет свойство `contentWindow`, которое возвращает для фрейма объект `window` со свойством `document`. Таким образом, для доступа к объекту `document` встроенного фрейма во всех браузерах можно использовать следующий код:

Листинг IFrameElementExample2.htm

```
var iframe = document.getElementById("myIframe");
var iframeDoc = iframe.contentDocument || iframe.contentWindow.document;
```

Свойство `contentWindow` доступно во всех браузерах.

Доступ к объекту document обычного или встроенного фрейма регламентируется междоменными ограничениями. Попытка доступа к объекту document фрейма со страницей, загруженной из другого домена или поддомена либо по другому протоколу, приведет к ошибке.

Стили

Чтобы определить в HTML-коде стили, можно включить в файл внешнюю таблицу стилей с помощью элемента `<link>`, добавить в файл встроенные стили, используя элемент `<style>`, или задать стили для отдельного элемента, указав атрибут `style`. DOM Level 2 Styles предоставляет API для всех этих механизмов. Узнать, поддерживает ли браузер DOM Level 2 CSS, можно следующим образом:

```
var supportsDOM2CSS = document.implementation.hasFeature("CSS", "2.0");
var supportsDOM2CSS2 = document.implementation.hasFeature("CSS2", "2.0");
```

Доступ к стилям элементов

У любого HTML-элемента, поддерживающего атрибут `style`, есть свойство `style`, доступное в JavaScript-коде. Объект `style` является экземпляром типа `CSSStyleDeclaration` и содержит все стили, заданные с помощью HTML-атрибута `style`, но не каскадно применяемые стили из внешних или встроенных таблиц стилей. Каждое CSS-свойство, заданное в атрибуте `style`, представлено свойством объекта `style`. Поскольку в именах CSS-свойств слова разделяются дефисом (например, `background-image`), для использования в JavaScript-коде их нужно преобразовать в верблюжью нотацию. В следующей таблице указаны некоторые часто используемые CSS-свойства и соответствующие им свойства объекта `style`:

CSS-свойство	JavaScript-свойство
background-image	style.backgroundImage
color	style.color
display	style.display
font-family	style.fontFamily

Имена CSS-свойств преобразуются в имена JavaScript-свойств напрямую, исключая свойство `float`. Поскольку в JavaScript это зарезервированное слово, его нельзя использовать как имя свойства. В спецификации DOM Level 2 Style сказано, что соответствующее свойство объекта `style` должно называться `cssFloat`. Оно поддерживается в Internet Explorer 9, Firefox, Safari, Opera и Chrome. В Internet Explorer 8 и более ранних версий вместо этого используется свойство `styleFloat`.

Стили можно задавать с помощью JavaScript-кода в любое время, если доступна действительная ссылка на DOM-элемент, например:

```
var myDiv = document.getElementById("myDiv");

// задание фонового цвета
myDiv.style.backgroundColor = "red";

// изменение размеров
myDiv.style.width = "100px";
myDiv.style.height = "200px";

// настройка границы
myDiv.style.border = "1px solid black";
```

При изменении стилей таким способом вид элемента автоматически обновляется.

В стандартном режиме все размеры должны включать единицу измерения. Если присвоить свойству style.width значение "20" в режиме совместимости, оно будет интерпретировано как "20px", но в стандартном режиме оно игнорируется. На практике лучше всегда указывать единицу измерения.

Стили, заданные в атрибуте `style`, можно также получать с помощью объекта `style`. Рассмотрим следующий HTML-код:

```
<div id="myDiv" style="background-color: blue; width: 10px; height: 25px">
</div>
```

Содержимое атрибута `style` этого элемента можно получить следующим образом:

```
alert(myDiv.style.backgroundColor);    // "blue"
alert(myDiv.style.width);              // "10px"
alert(myDiv.style.height);             // "25px"
```

Если для элемента не указан атрибут `style`, объект `style` может содержать некоторые значения, предлагаемые по умолчанию, но они не отражают стиль элемента.

Свойства и методы Style

Спецификация DOM Level 2 Style определяет также несколько свойств и методов объекта `style`, которые предоставляют информацию о содержимом атрибута `style` и изменяют его:

- `cssText` — возвращает CSS-код атрибута `style`;
- `length` — количество CSS-свойств, примененных к элементу;
- `parentRule` — объект `CSSRule`, представляющий CSS-информацию (тип `CSSRule` обсуждается позже);

- `getPropertyCSSValue(`*имяСвойства*`)` — возвращает объект `CSSValue`, содержащий значение указанного свойства;
- `getPropertyPriority(`*имяСвойства*`)` — возвращает значение `"important"`, если указанное свойство задано с объявлением `!important`, иначе возвращает пустую строку;
- `getPropertyValue(`*имяСвойства*`)` — возвращает строковое значение указанного свойства;
- `item(`*индекс*`)` — возвращает имя CSS-свойства в указанной позиции;
- `removeProperty(`*имяСвойства*`)` — удаляет указанное свойство из стиля;
- `setProperty(`*имяСвойства*, *значение*, *приоритет*`)` — присваивает указанному свойству полученное значение с заданным приоритетом (`"important"` или пустая строка).

Свойство `cssText` обеспечивает доступ к CSS-коду стиля. В режиме чтения оно возвращает внутреннее представление значения атрибута `style` в браузере. При записи свойства `cssText` присваиваемое ему значение перезаписывает все содержимое атрибута `style`, то есть все прежние параметры стиля, заданные с помощью этого атрибута, удаляются. Например, если для элемента в атрибуте `style` задана граница и свойство `cssText` перезаписывается по правилам, которые не включают границу, она удаляется из элемента. Свойство `cssText` используется следующим образом:

```
myDiv.style.cssText = "width: 25px; height: 100px; background-color: green";
alert(myDiv.style.cssText);
```

Задание свойства `cssText` — это самый быстрый способ внесения многих изменений в стиль элемента, потому что все они применяются одновременно.

Свойство `length` используется вместе с методом `item()` для перебора CSS-свойств элемента. По сути, объект `style` при этом обрабатывается как коллекция, так что для получения имени CSS-свойства в конкретной позиции можно задействовать скобочную нотацию вместо метода `item()`:

```
for (var i=0, len=myDiv.style.length; i < len; i++){
    alert(myDiv.style[i]);        // или myDiv.style.item(i)
}
```

Используя скобочную нотацию или метод `item()`, можно получить имя CSS-свойства (`"background-color"`, но не `"backgroundColor"`), а затем передать его в метод `getPropertyValue()` для получения значения свойства:

```
var prop, value, i, len;
for (i=0, len=myDiv.style.length; i < len; i++){
    prop = myDiv.style[i];        // или myDiv.style.item(i)
    value = myDiv.style.getPropertyValue(prop);
    alert(prop + " : " + value);
}
```

Метод `getPropertyValue()` всегда возвращает значение CSS-свойства как строку. Если вам нужны дополнительные сведения, можно использовать метод `getPropertyCSSValue()`, который возвращает объект `CSSValue` со свойствами `cssText` и `cssValueType`. Свойство `cssText` не отличается от значения, возвращаемого методом `getPropertyValue()`, а свойство `cssValueType` содержит числовую константу, указывающую тип значения свойства: 0 — унаследованное значение; 1 — примитивное значение; 2 — список; 3 — пользовательское значение. Следующий код выводит значение CSS-свойства и его тип:

Листинг DOMStyleObjectExample.htm


```
var prop, value, i, len;
for (i=0, len=myDiv.style.length; i < len; i++){
    prop = myDiv.style[i];        //или myDiv.style.item(i)
    value = myDiv.style.getPropertyCSSValue(prop);
    alert(prop + " : " + value.cssText + " (" + value.cssValueType + ")");
}
```

На практике метод `getPropertyCSSValue()` менее полезен, чем `getPropertyValue()`. Он поддерживается в Internet Explorer 9+, Safari 3+ и Chrome. В Firefox до версии 7 включительно он доступен, но всегда возвращает `null`.

Метод `removeProperty()` удаляет CSS-свойство из стиля элемента, при этом к элементу применяется предлагаемый по умолчанию стиль, полученный по каскаду из других таблиц стилей. Например, следующий код удаляет свойство `border`, которое было задано в атрибуте `style`:

```
myDiv.style.removeProperty("border");
```

Этот метод полезен, если вы не знаете, каково предлагаемое по умолчанию значение конкретного CSS-свойства. Для восстановления значения, предлагаемого по умолчанию, можно просто удалить свойство.

Если не указано иное, свойства и методы, описываемые в этом разделе, поддерживаются в Internet Explorer 9+, Firefox, Safari, Opera 9+ и Chrome.

Вычисляемые стили

Объект `style` позволяет получить значение атрибута `style` любого элемента, который его поддерживает, но не содержит сведений о стилях, примененных к элементу по каскаду из таблиц стилей. DOM Level 2 Style расширяет объект `document.defaultView` методом `getComputedStyle()`, принимающим два аргумента: элемент, для которого нужно получить вычисляемый стиль, и строку псевдоэлемента (например, `":after"`). Вторым аргументом может быть значение `null`, если сведения о псевдоэлементе не нужны. Метод `getComputedStyle()` возвращает объект `CSSStyleDeclaration` (как и свойство `style`), содержащий все вычисляемые стили элемента. Рассмотрим следующую HTML-страницу:

Листинг ComputedStylesExample.htm


```
<!DOCTYPE html>
<html>
<head>
    <title>Computed Styles Example</title>
    <style type="text/css">
        #myDiv {
            background-color: blue;
            width: 100px;
            height: 200px;
        }
    </style>
</head>
<body>
    <div id="myDiv" style="background-color: red; border: 1px solid black">
    </div>
</body>
</html>
```

В этом примере к элементу `<div>` применены стили из встроенной таблицы стилей (элемент `<style>`) и из атрибута `style`. Объект `style` содержит значения свойств `backgroundColor` и `border`, но не `width` и `height`, которые берутся из таблицы стилей. Следующий код получает вычисляемый стиль элемента:

Листинг ComputedStylesExample.htm


```
var myDiv = document.getElementById("myDiv");
var computedStyle = document.defaultView.getComputedStyle(myDiv, null);

alert(computedStyle.backgroundColor); // "red"
alert(computedStyle.width);           // "100px"
alert(computedStyle.height);          // "200px"
alert(computedStyle.border);          // "1px solid black" в ряде браузеров
```

Этот код возвращает фоновый цвет `"red"`, ширину `"100px"` и высоту `"200px"`. Обратите внимание, что фоновый цвет отличается от `"blue"`, потому что он переопределен в самом элементе. Свойство `border` может не возвратить точное правило `border` из таблицы стилей (Opera его возвращает, но не остальные браузеры). Это несоответствие связано с тем, как браузеры интерпретируют агрегирующие свойства вроде `border`, которые на самом деле задают ряд других свойств. При установке свойства `border` вы на самом деле задаете правила для ширины, цвета и стиля всех четырех границ (`border-left-width`, `border-top-color`, `border-bottom-style` и т. д.). Таким образом, хотя свойство `computedStyle.border` возвращает значение не во всех браузерах, свойство `computedStyle.borderLeftWidth` работает везде.

В браузерах, которые поддерживают этот функционал, значения стилей представляются по-разному. Firefox и Safari преобразуют все цвета в формат RGB (например, rgb(255,0,0) для красного), а Opera представляет цвета в шестнадцатеричном формате (#ff0000 для красного). При использовании метода getComputedStyle() следует тестировать код в разных браузерах.

Internet Explorer не поддерживает метод `getComputedStyle()`, но в нем доступны схожие механизмы. У каждого элемента со свойством `style` есть также свойство `currentStyle`, которое имеет тип `CSSStyleDeclaration` и содержит все окончательные вычисляемые стили элемента. Их можно получить аналогичным образом:

Листинг IEComputedStylesExample.htm

```
var myDiv = document.getElementById("myDiv");
var computedStyle = myDiv.currentStyle;

alert(computedStyle.backgroundColor);    // "red"
alert(computedStyle.width);              // "100px"
alert(computedStyle.height);             // "200px"
alert(computedStyle.border);             // undefined
```

Как и в версии DOM, Internet Explorer не возвращает стиль `border`, потому что он считается агрегирующим свойством.

Вычисляемые стили во всех браузерах доступны только для чтения — изменить CSS-свойства в объекте вычисляемого стиля нельзя. Кроме того, вычисляемый стиль содержит стили из внутренней таблицы стилей браузера, а потому в нем представлено любое CSS-свойство, у которого есть значение, предлагаемое по умолчанию. Например, у свойства `visibility` есть значение, предлагаемое по умолчанию во всех браузерах, но оно зависит от реализации. В некоторых браузерах это свойство имеет по умолчанию значение `"visible"`, в других — `"inherit"`. Нельзя полагаться на то, что значение CSS-свойства, предлагаемое по умолчанию, будет одинаковым во всех браузерах. Если требуется, чтобы элементы по умолчанию имели определенное значение, следует задать его вручную в таблице стилей.

Работа с таблицами стилей

Тип `CSSStyleSheet` представляет таблицу CSS-стилей, включенную в файл с помощью элемента `<link>` или определенную в элементе `<style>`. Сами элементы при этом имеют типы `HTMLLinkElement` и `HTMLStyleElement` соответственно. Тип `CSSStyleSheet` достаточно универсален, чтобы с его помощью можно было представить таблицу стилей независимо от того, как она определена в HTML. Типы, специфичные для элементов, позволяют изменять HTML-атрибуты, тогда как объект `CSSStyleSheet`, за исключением одного свойства, доступен только для чтения. Узнать, поддерживает ли браузер таблицы стилей DOM Level 2, можно следующим образом:

```
var supportsDOM2StyleSheets =
    document.implementation.hasFeature("StyleSheets", "2.0");
```

Тип `CSSStyleSheet` наследуется от типа `StyleSheet`, который можно использовать как основу для определения таблиц стилей, отличных от CSS. От типа `StyleSheet` наследуются следующие свойства.

- `disabled` — логическое значение, указывающее, отключена ли таблица стилей. Это свойство доступно для чтения и записи, так что можно отключить таблицу стилей, присвоив ему значение `true`.
- `href` — URL-адрес таблицы стилей, если она включается в файл с помощью элемента `<link>`, иначе значение `null`.
- `media` — носители информации, поддерживаемые таблицей стилей. Как и все DOM-коллекции, эта содержит свойство `length` и метод `item()`. Для доступа к элементам коллекции можно задействовать скобочную нотацию. Пустой список указывает, что таблица стилей должна использоваться со всеми носителями. В Internet Explorer 8 и более ранних версий свойство `media` содержит строку, соответствующую атрибуту `media` элемента `<link>` или `<style>`.
- `ownerNode` — указатель на узел, которому принадлежит таблица стилей. В HTML им может быть элемент `<link>` или `<style>`, а в XML — инструкция по обработке. Если таблица стилей включается в другую таблицу стилей с помощью правила `@import`, это свойство имеет значение `null`. В Internet Explorer 8 и более ранних версий оно не поддерживается.
- `parentStyleSheet` — если таблица стилей включается в файл помощью правила `@import`, это свойство содержит указатель на таблицу стилей, из которой импортируется первая таблица.
- `title` — значение атрибута `title` узла `ownerNode`.
- `type` — строка, указывающая тип таблицы стилей (`"text/css"` в случае таблицы CSS-стилей).

За исключением `disabled`, все эти свойства доступны только для чтения. В дополнение к ним тип `CSSStyleSheet` поддерживает свои свойства и методы.

- `cssRules` — набор правил, содержащихся в таблице стилей. В Internet Explorer 8 и более ранних версий это свойство не поддерживается, но доступно похожее свойство `rules`. Internet Explorer 9 поддерживает оба свойства.
- `ownerRule` — если таблица стилей была включена в файл с помощью правила `@import`, это свойство содержит указатель на правило, представляющее импорт, в противном случае оно имеет значение `null`. Internet Explorer не поддерживает это свойство.
- `deleteRule(`*индекс*`)` — удаляет правило в указанной позиции в коллекции `cssRules`. В Internet Explorer 8 и более ранних версий этот метод не поддерживается, но доступен похожий метод `removeRule()`. Internet Explorer 9 поддерживает оба метода.
- `insertRule(`*правило*, *индекс*`)` — вставляет полученное строковое правило в указанной позиции в коллекцию `cssRules`. В Internet Explorer 8 и более ранних версий этот метод не поддерживается, но доступен похожий метод `addRule()`. Internet Explorer 9 поддерживает оба метода.

Таблицы стилей документа содержатся в коллекции `document.styleSheets`. Количество таблиц стилей у документа можно узнать с помощью свойства `length`, а для

доступа к отдельным таблицам стилей можно использовать метод `item()` или скобочную нотацию, например:

Листинг StyleSheetsExample.htm


```
var sheet = null;
for (var i=0, len=document.styleSheets.length; i < len; i++){
    sheet = document.styleSheets[i];
    alert(sheet.href);
}
```

Этот код выводит на экран свойство `href` каждой таблицы стилей в документе (у элементов `<style>` нет свойства `href`).

Таблицы стилей, возвращаемые свойством `document.styleSheets`, зависят от браузера. Все браузеры возвращают элементы `<style>`, а также элементы `<link>`, у которых атрибут `rel` имеет значение `"stylesheet"`, а Internet Explorer и Opera — еще и элементы `<link>`, у которых атрибут `rel` равен `"alternate stylesheet"`.

Можно также получить объект `CSSStyleSheet` непосредственно из элемента `<link>` или `<style>` с помощью свойства `sheet`. Оно содержит объект `CSSStyleSheet` и поддерживается всеми браузерами, кроме Internet Explorer, в котором доступно аналогичное свойство `styleSheet`. Для получения объекта таблицы стилей в любых браузерах можно использовать следующий код:

Листинг StyleSheetsExample2.htm

```
function getStyleSheet(element){
    return element.sheet || element.styleSheet;
}

// получение таблицы стилей для первого элемента <link/>
var link = document.getElementsByTagName("link")[0];
var sheet = getStyleSheet(link);
```

Метод `getStyleSheet()` возвращает тот же объект, что и коллекция `document.styleSheets`.

Правила CSS

Объект `CSSRule` представляет отдельное правило в таблице стилей. От типа `CSSRule` наследуются несколько других типов, из которых чаще всего используется тип `CSSStyleRule`, представляющий данные стиля (среди других правил — `@import`, `@font-face`, `@page` и `@charset`, но в сценариях они требуются редко). Перечислим свойства объекта `CSSStyleRule`.

- `cssText` — возвращает текст всего правила. Он может отличаться от фактического текста в таблице стилей из-за особенностей внутренней обработки таблиц стилей в браузерах (например, Safari всегда преобразует текст в нижний регистр). Это свойство не поддерживается в Internet Explorer.

- `parentRule` — если правило импортируется, это свойство содержит правило-контейнер, иначе оно равно `null`. Это свойство не поддерживается в Internet Explorer.
- `parentStyleSheet` — таблица стилей, в которую входит правило. Это свойство не поддерживается в Internet Explorer.
- `selectorText` — возвращает текст селектора для правила. Он может отличаться от фактического текста в таблице стилей из-за особенностей внутренней обработки таблиц стилей в браузерах. Это свойство доступно только для чтения в Firefox, Safari, Chrome и Internet Explorer (где оно генерирует ошибку). В Opera его также можно изменять.
- `style` — объект `CSSStyleDeclaration`, позволяющий задавать и читать отдельные стили в правиле.
- `type` — константа, указывающая тип правила. У правил стилей она всегда равна 1. Это свойство не поддерживается в Internet Explorer.

Чаще всего используются свойства `cssText`, `selectorText` и `style`. Свойство `cssText` похоже на `style.cssText`, но не идентично ему. Первое включает текст селектора и данные стиля в фигурных скобках, а второе содержит только данные стиля (подобно свойству `style.cssText` элемента). Кроме того, свойство `cssText` доступно только для чтения, а `style.cssText` можно перезаписывать.

В большинстве случаев свойства `style` достаточно для манипулирования правилами стилей. Его можно использовать для чтения и изменения стилей в правиле так же, как и аналогичное свойство элемента. Рассмотрим следующее CSS-правило:

Листинг CSSRulesExample.htm


```
div.box {
    background-color: blue;
    width: 100px;
    height: 200px;
}
```

Если это правило содержится в первой таблице стилей на странице и является в ней единственным, получить все эти данные можно следующим образом:

Листинг CSSRulesExample.htm


```
var sheet = document.styleSheets[0];
var rules = sheet.cssRules || sheet.rules;    // получение списка правил
var rule = rules[0];                          // получение первого правила
alert(rule.selectorText);                     // "div.box"
alert(rule.style.cssText);                    // полный CSS-код
alert(rule.style.backgroundColor);            // "blue"
alert(rule.style.width);                      // "100px"
alert(rule.style.height);                     // "200px"
```

Используя эту методику, можно читать стили, связанные с правилами, так же, как встроенные стили элементов. Изменение стилей также возможно, например:

Листинг CSSRulesExample.htm

```
var sheet = document.styleSheets[0];
var rules = sheet.cssRules || sheet.rules;    // получение списка правил
var rule = rules[0];                          // получение первого правила
rule.style.backgroundColor = "red"
```

Изменение правила таким способом влияет на все элементы, к которым оно применено. Например, если страница содержит два элемента `<div>` класса `box`, будут изменены оба элемента.

Создание правил

Для добавления новых правил в существующие таблицы стилей используется метод `insertRule()`, который принимает два аргумента: текст правила и позицию, где нужно его вставить, например:

```
sheet.insertRule("body { background-color: silver }", 0);  // DOM-метод
```

Этот код вставляет в таблицу стилей правило, изменяющее фоновый цвет документа. Новое правило становится первым (позиция 0), что важно для определения порядка каскадного применения правил в документе. Метод `insertRule()` поддерживается в Internet Explorer 9+ и всех современных версиях Firefox, Safari, Opera и Chrome.

В Internet Explorer 8 и более ранних версий есть похожий метод `addRule()`, который принимает текст селектора и данные CSS-стилей. Необязательный третий аргумент указывает позицию, в которой нужно вставить правило. Вот эквивалент предыдущего примера для Internet Explorer:

```
sheet.addRule("body", "background-color: silver", 0);      // только IE
```

В документации сказано, что с помощью метода `addRule()` можно добавить до 4095 правил стилей; последующие вызовы завершаются ошибками.

Для добавления правила в таблицу стилей независимо от браузера можно использовать приведенный далее метод. Он принимает четыре аргумента: таблицу стилей, в которую нужно добавить правило, и три аргумента метода `addRule()`.

Листинг CSSRulesExample2.htm


```
function insertRule(sheet, selectorText, cssText, position){
    if (sheet.insertRule){
        sheet.insertRule(selectorText + "{" + cssText + "}", position);
    } else if (sheet.addRule){
        sheet.addRule(selectorText, cssText, position);
    }
}
```

Вызывается этот метод следующим образом:

```
insertRule(document.styleSheets[0], "body", "background-color: silver", 0);
```

К сожалению, этот способ слишком громоздок, если нужно добавить много правил. В этом случае лучше использовать динамическую загрузку стилей (см. главу 10).

Удаление правил

Для удаления правил из таблицы стилей в DOM используется метод `deleteRule()`, принимающий индекс правила, которое нужно удалить. Например, удалить первое правило в таблице стилей можно следующим образом:

```
sheet.deleteRule(0);    // DOM-метод
```

В Internet Explorer 8 и более ранних версий доступен метод `removeRule()`, который используется так же:

```
sheet.removeRule(0);    // только IE
```

Для удаления правила независимо от браузера можно задействовать следующую функцию, которая принимает таблицу стилей и индекс удаляемого правила:

Листинг CSSRulesExample2.htm

```
function deleteRule(sheet, index){
    if (sheet.deleteRule){
        sheet.deleteRule(index);
    } else if (sheet.removeRule){
        sheet.removeRule(index);
    }
}
```

Вызывается она так:

```
deleteRule(document.styleSheets[0], 0);
```

Добавлять и удалять правила при разработке веб-приложений требуется редко, при этом нужно соблюдать осторожность, чтобы не нарушить каскадное применение стилей.

Размеры элементов

Описываемые здесь свойства и методы не входят в спецификацию DOM Level 2 Style, но все же связаны со стилями HTML-элементов. DOM не описывает способы определения фактических размеров элементов на странице. Некоторые такие свойства впервые появились в Internet Explorer, а затем были реализованы во всех основных браузерах.

Смещения

Свойства из первой группы представляют *размеры смещений* (offset dimensions), которые охватывают все визуальное пространство, занимаемое элементом на экране.

Оно определяется высотой и шириной элемента и включает все отступы, полосы прокрутки и границы элемента (но не поля). Получить размеры смещений можно с помощью четырех следующих свойств:

- `offsetHeight` — размер элемента по вертикали в пикселях, включающий высоту самого элемента, горизонтальной полосы прокрутки (если она отображается), а также верхней и нижней границ;
- `offsetLeft` — расстояние в пикселях между левой внешней границей элемента и левой внутренней границей элемента-контейнера;
- `offsetTop` — расстояние в пикселях между верхней внешней границей элемента и верхней внутренней границей элемента-контейнера;
- `offsetWidth` — размер элемента по горизонтали в пикселях, включающий ширину самого элемента, вертикальной полосы прокрутки (если она отображается), а также левой и правой границ.

Значения `offsetLeft` и `offsetTop` рассчитываются относительно элемента-контейнера, который хранится в свойстве `offsetParent` и может отличаться от `parentNode`. Например, у элемента `<td>` свойство `offsetParent` указывает на соответствующий элемент `<table>`, потому что это первый элемент в иерархии, для которого определены размеры. Размеры, соответствующие этим свойствам, показаны на рис. 12.1.

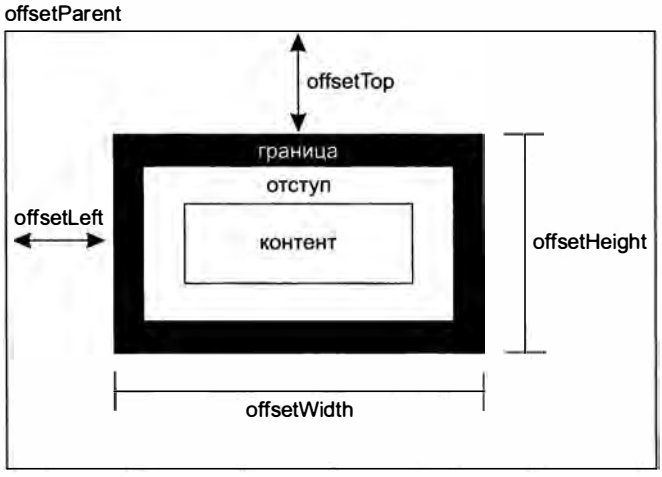

Рис. 12.1

Смещение элемента на странице можно примерно определить, сложив значения свойств `offsetLeft` и `offsetTop` с аналогичными значениями элемента `offsetParent` и более высокоуровневых элементов в иерархии вплоть до корневого элемента, например:

Листинг OffsetDimensionsExample.htm


```
function getElementLeft(element){
    var actualLeft = element.offsetLeft;
    var current = element.offsetParent;

    while (current !== null){
        actualLeft += current.offsetLeft;
        current = current.offsetParent;
    }

    return actualLeft;
}

function getElementTop(element){
    var actualTop = element.offsetTop;
    var current = element.offsetParent;

    while (current !== null){
        actualTop += current.offsetTop;
        current = current.offsetParent;
    }

    return actualTop;
}
```

Эти функции поднимаются по DOM-иерархии с помощью свойства `offsetParent`, суммируя смещения на каждом уровне. Если страницы основаны на простых CSS-макетах, эти функции очень точны. Если страницы содержат таблицы и встроенные фреймы, возвращаемые этими функциями значения могут зависеть от браузера из-за различий реализации. Как правило, у всех элементов, содержащихся в элементах `<div>`, свойство `offsetParent` указывает на элемент `<body>`, так что функции `getElementLeft()` и `getElementTop()` возвращают для них те же значения, что и свойства `offsetLeft` и `offsetTop`.

Все свойства смещений доступны только для чтения и вычисляются при каждом обращении к ним. Чтобы не снижать быстродействие, лучше не вызывать эти свойства повторно, а кэшировать их значения в локальных переменных.

Клиентские размеры

Клиентские размеры (client dimensions) элемента, которые представлены свойствами `clientWidth` и `clientHeight`, определяют пространство, занимаемое содержимым элемента и отступами. Значение `clientWidth` равно ширине области контента с левым и правым отступами, а `clientHeight` — высоте области контента с верхним и нижним отступами. Эти размеры показаны на рис. 12.2.

Клиентские размеры не включают полосы прокрутки и чаще всего используются для определения размеров области просмотра браузера (см. главу 8). Это делается с помощью свойств `clientWidth` и `clientHeight` объекта `document.documentElement` или `document.body` (в Internet Explorer 6 и более ранних версий):

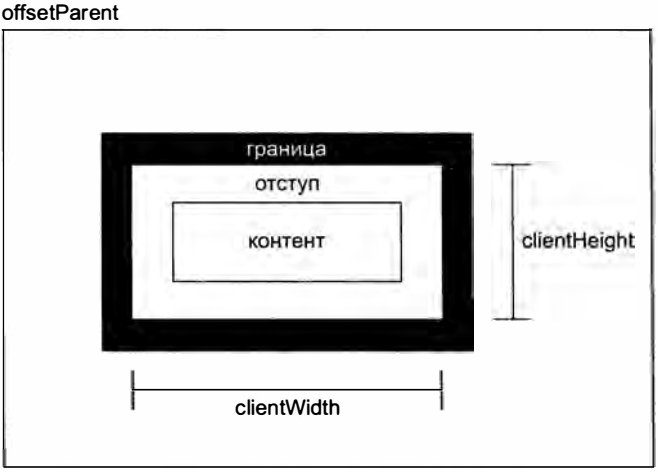

Рис. 12.2

```
function getViewport(){
    if (document.compatMode == "BackCompat"){
        return {
            width: document.body.clientWidth,
            height: document.body.clientHeight
        };
    } else {
        return {
            width: document.documentElement.clientWidth,
            height: document.documentElement.clientHeight
        };
    }
}
```

Прежде всего эта функция проверяет свойство `document.compatMode`, чтобы выяснить, работает ли браузер в режиме совместимости. Браузеры Internet Explorer 8+, Chrome, Safari, Opera и Firefox обычно работают в стандартном режиме, поэтому в них выполняется ветвь `else`. Функция возвращает объект со свойствами `width` и `height`, которые представляют размеры области просмотра (элемент `<html>` или `<body>`).

Как и смещения, клиентские размеры доступны только для чтения и вычисляются при каждом обращении к ним.

Размеры области прокрутки

Последняя группа размеров — это *размеры области прокрутки* (scroll dimensions), предоставляющие сведения об элементе, содержимое которого можно прокручивать. Некоторые элементы, такие как `<html>`, прокручиваются автоматически без

дополнительного кода, тогда как другие можно прокручивать с помощью CSS-свойства `overflow`. Четыре размера прокрутки таковы:

- `scrollHeight` — общая высота контента без полос прокрутки;
- `scrollLeft` — количество скрытых пикселей слева от области контента (с помощью этого свойства можно прокрутить область элемента);
- `scrollTop` — количество скрытых пикселей сверху от области контента (с помощью этого свойства можно прокрутить область элемента);
- `scrollWidth` — общая ширина контента без полос прокрутки.

Эти размеры показаны на рис. 12.3.

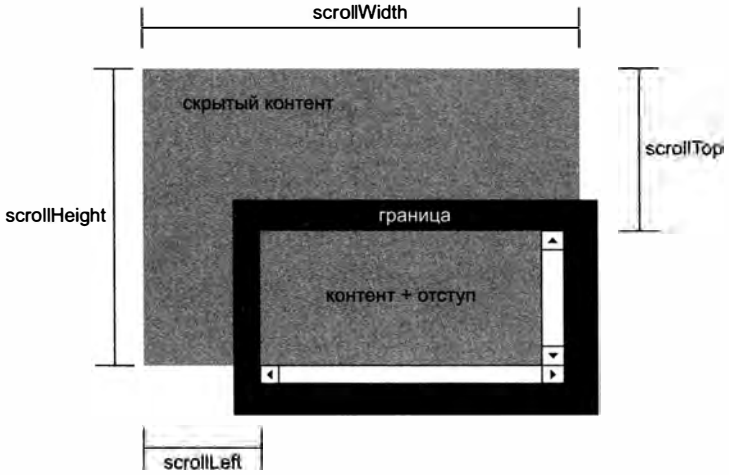

Рис. 12.3

С помощью свойств `scrollWidth` и `scrollHeight` можно узнать фактические размеры содержимого элемента. Например, прокручиваемая область просмотра в веб-браузере представлена элементом `<html>` (в Internet Explorer 5.5 и более ранних версий это элемент `<body>`). Следовательно, высота всей страницы, которую можно прокручивать по вертикали, равна `document.documentElement.scrollHeight`.

Если документ не прокручивается, отношение между значениями `scrollWidth` и `scrollHeight`, с одной стороны, и `clientWidth` и `clientHeight`, с другой, не определено. У объекта `document.documentElement` эти свойства имеют несогласованные значения в разных браузерах:

- в Firefox эти свойства равны, но их значения определяются по фактическому контенту документа, а не по области просмотра;
- в Opera, Safari 3.1 и более поздних версий и Chrome значения `scrollWidth` и `scrollHeight` соответствуют размерам области просмотра, а `clientWidth` и `clientHeight` — размерам контента документа;

- в Internet Explorer (в стандартном режиме) значения `scrollWidth` и `scrollHeight` соответствуют размерам контента документа, а `clientWidth` и `clientHeight` — размерам области просмотра.

При определении общих размеров документа, в том числе минимальных размеров по области просмотра, для получения правильных результатов в разных браузерах нужно брать максимальные значения из пар `scrollWidth`/`clientWidth` и `scrollHeight`/`clientHeight`, например:

```
var docHeight = Math.max(document.documentElement.scrollHeight,
                         document.documentElement.clientHeight);

var docWidth = Math.max(document.documentElement.scrollWidth,
                        document.documentElement.clientWidth);
```

В Internet Explorer в режиме совместимости нужно использовать аналогичные свойства объекта `document.body`, а не `document.documentElement`.

Свойства `scrollLeft` и `scrollTop` можно использовать как для определения текущих параметров прокрутки, так и для их установки. Если элемент не прокручивался, оба свойства равны нулю. Если элемент прокручивался по вертикали, свойство `scrollTop` указывает объем скрытого содержимого сверху от отображаемой области элемента, а свойство `scrollLeft` — слева от нее. Обнулив свойства `scrollLeft` и `scrollTop`, можно показать в элементе начальную область. Следующая функция прокручивает содержимое элемента по вертикали к самому началу, если оно скрыто:

```
function scrollToTop(element){
    if (element.scrollTop != 0){
        element.scrollTop = 0;
    }
}
```

Для прокрутки содержимого функция использует свойство `scrollTop`.

Определение размеров элемента

В Internet Explorer, Firefox 3+, Safari 4+, Opera 9.5+ и Chrome у каждого элемента есть метод `getBoundingClientRect()`. Он возвращает прямоугольник со свойствами `left`, `top`, `right` и `bottom`, которые определяют расположение элемента на странице относительно области просмотра. Реализации браузеров в этом плане слегка различаются. В Internet Explorer 8 и более ранних версий левый верхний угол документа имеет координаты (2,2), а в других браузерах, включая Internet Explorer 9, — традиционные (0,0). Это требует проверки координат элемента в левом верхнем углу, которые равны (2,2) в Internet Explorer 8 и более ранних версий и (0,0) в других браузерах, например:

Листинг GetBoundingClientRectExample.htm


```
function getBoundingClientRect(element){
    if (typeof arguments.callee.offset != "number"){
        var scrollTop = document.documentElement.scrollTop;
```

```
        var temp = document.createElement("div");
        temp.style.cssText = "position:absolute;left:0;top:0;";
        document.body.appendChild(temp);
        arguments.callee.offset = -temp.getBoundingClientRect().top -
                                  scrollTop;
        document.body.removeChild(temp);
        temp = null;
    }

    var rect = element.getBoundingClientRect();
    var offset = arguments.callee.offset;

    return {
        left: rect.left + offset,
        right: rect.right + offset,
        top: rect.top + offset,
        bottom: rect.bottom + offset
    };
}
```

Для коррекции координат эта функция использует собственное свойство `offset`. Первым делом она выясняет, определено ли это свойство, и если нет, определяет его, присваивая ему отрицательное значение верхней координаты нового элемента, то есть –2 в Internet Explorer и –0 в Firefox и Opera. Для получения нужного значения функция создает временный элемент в позиции (0,0), вызывает его метод `getBoundingClientRect()` и вычитает из координаты `top` значение `scrollTop` области просмотра на тот случай, если окно прокручивалось до вызова метода. Использование этой конструкции позволяет не дублировать вызов `getBoundingClientRect()` при каждом вызове функции. Затем вызывается метод `getBoundingClientRect()` элемента и создается объект с исправленными значениями.

В браузерах, которые не поддерживают метод `getBoundingClientRect()`, те же данные можно получить иначе. Как правило, разница значений `right` и `left` равна `offsetWidth`, а разница значений `bottom` и `top` — `offsetHeight`. Кроме того, свойства `left` и `top` примерно соответствуют функциям `getElementLeft()` и `getElementTop()`, определенным ранее в этой главе. Это позволяет реализовать кроссбраузерную версию функции:

Листинг GetBoundingClientRectExample.htm


```
function getBoundingClientRect(element){

    var scrollTop = document.documentElement.scrollTop;
    var scrollLeft = document.documentElement.scrollLeft;

    if (element.getBoundingClientRect){
        if (typeof arguments.callee.offset != "number"){
            var temp = document.createElement("div");
            temp.style.cssText = "position:absolute;left:0;top:0;";
            document.body.appendChild(temp);
            arguments.callee.offset = -temp.getBoundingClientRect().top -
```

```
                                        scrollTop;
            document.body.removeChild(temp);
            temp = null;
        }

        var rect = element.getBoundingClientRect();
        var offset = arguments.callee.offset;

        return {
            left: rect.left + offset,
            right: rect.right + offset,
            top: rect.top + offset,
            bottom: rect.bottom + offset
        };
    } else {

        var actualLeft = getElementLeft(element);
        var actualTop = getElementTop(element);

        return {
            left: actualLeft - scrollLeft,
            right: actualLeft + element.offsetWidth - scrollLeft,
            top: actualTop - scrollTop,
            bottom: actualTop + element.offsetHeight - scrollTop
        }
    }
}
```

Эта функция использует встроенный метод `getBoundingClientRect()`, если он доступен, и вычисляет размеры в противном случае. Иногда эти значения могут различаться, например, если страница содержит таблицы или элементы с прокруткой.

Из-за использования свойства arguments.callee этот метод не будет работать в строгом режиме.

Обход

В DOM Level 2 Traversal and Range определены типы `NodeIterator` и `TreeWalker`, служащие для последовательного обхода DOM-структуры в глубину начиная с указанной точки. Они доступны в браузерах, которые соответствуют спецификации DOM, включая Internet Explorer 9+, Firefox, Safari, Opera и Chrome. В Internet Explorer 8 и более ранних версий обход DOM не поддерживается. Узнать, поддерживает ли браузер модуль DOM Level 2 Traversal, можно следующим образом:

```
var supportsTraversals =
    document.implementation.hasFeature("Traversal", "2.0");
var supportsNodeIterator =
    (typeof document.createNodeIterator == "function");
var supportsTreeWalker =
    (typeof document.createTreeWalker == "function");
```

Обход DOM-структуры возможен как минимум в двух направлениях (в зависимости от используемого типа), при этом подняться по DOM-дереву выше корневого узла нельзя. Рассмотрим следующую HTML-страницу:

```
<!DOCTYPE html>
<html>
    <head>
        <title>Example</title>
    </head>
    <body>
        <p><b>Hello</b> world!</p>
    </body>
</html>
```

Этой странице соответствует DOM-дерево, показанное на рис. 12.4.

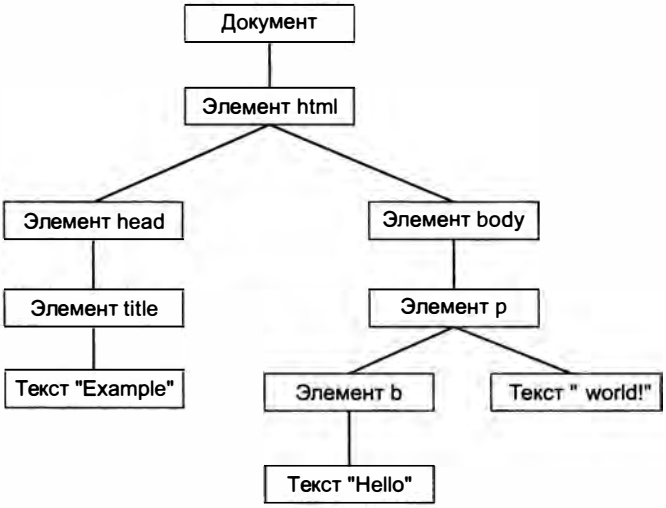

Рис. 12.4

Корневым при обходе может быть любой узел. Предположим, например, что им является элемент `<body>`. Тогда при обходе могут быть посещены элементы `<p>` и `<b>`, а также два текстовых узла, которые являются потомками `<body>`, но не элементы `<html>`, `<head>` или любые другие узлы, не входящие в поддерево элемента `<body>`. В то же время при обходе с корневым узлом `document` доступны все узлы документа. На рис. 12.5 показан обход DOM-дерева с корнем `document` в глубину.

Обход начинается узлом `document` и завершается текстовым узлом `" world!"`, но можно обойти узлы и в обратном направлении. В этом случае текстовый узел `" world!"` будет посещен первым, а узел `document` — последним. Типы `NodeIterator` и `TreeWalker` поддерживают оба способа.

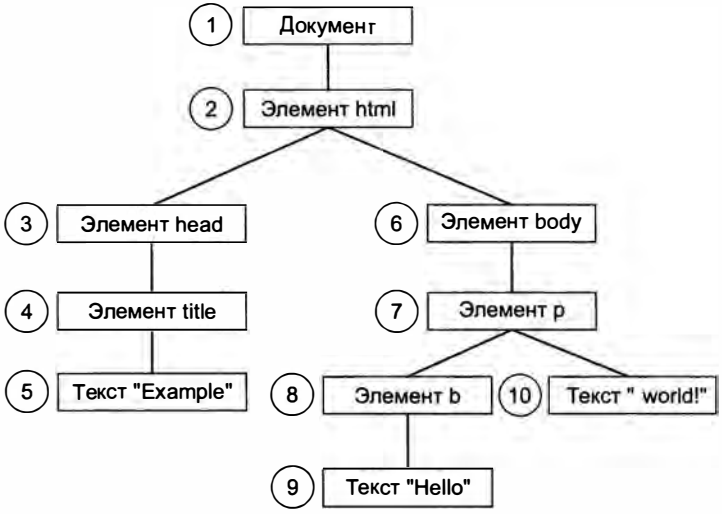

Рис. 12.5

Тип NodeIterator

Тип `NodeIterator` проще, чем `TreeWalker`; создать его экземпляр можно с помощью метода `document.createNodeIterator()`, который принимает четыре аргумента:

- `root` — узел в дереве, с которого нужно начать обход;
- `whatToShow` — числовой код, указывающий, какие узлы нужно посетить;
- `filter` — объект `NodeFilter` или некая функция, указывающие, нужно ли посетить конкретный узел;
- `entityReferenceExpansion` — логическое значение, указывающее, нужно ли развертывать ссылки на сущности (в HTML-страницах ссылки на сущности никогда не развертываются).

Аргумент `whatToShow` — это битовая маска с фильтрами, которые определяют, какие узлы нужно посетить. Значениями этого аргумента могут быть следующие константы типа `NodeFilter`:

- `NodeFilter.SHOW_ALL` — отображать узлы всех типов;
- `NodeFilter.SHOW_ELEMENT` — отображать узлы элементов;
- `NodeFilter.SHOW_ATTRIBUTE` — отображать узлы атрибутов (из-за DOM-структуры использовать это значение невозможно);
- `NodeFilter.SHOW_TEXT` — отображать текстовые узлы;
- `NodeFilter.SHOW_CDATA_SECTION` — отображать узлы CDATA-разделов (в HTML-страницах не используется);

- `NodeFilter.SHOW_ENTITY_REFERENCE` — отображать узлы ссылок на сущности (в HTML-страницах не используется);
- `NodeFilter.SHOW_ENTITY` — отображать узлы сущностей (в HTML-страницах не используется);
- `NodeFilter.SHOW_PROCESSING_INSTRUCTION` — отображать узлы инструкций по обработке (в HTML-страницах не используется);
- `NodeFilter.SHOW_COMMENT` — отображать узлы комментариев;
- `NodeFilter.SHOW_DOCUMENT` — отображать узлы документов;
- `NodeFilter.SHOW_DOCUMENT_TYPE` — отображать узлы типов документов;
- `NodeFilter.SHOW_DOCUMENT_FRAGMENT` — отображать узлы фрагментов документов (в HTML-страницах не используется);
- `NodeFilter.SHOW_NOTATION` — отображать узлы обозначений (в HTML-страницах не используется).

С помощью поразрядного оператора ИЛИ можно комбинировать эти параметры (исключая значение `NodeFilter.SHOW_ALL`), например:

```
var whatToShow = NodeFilter.SHOW_ELEMENT | NodeFilter.SHOW_TEXT;
```

С помощью аргумента `filter` метода `createNodeIterator()` можно задать в качестве фильтра узлов собственный объект `NodeFilter` или функцию, действующую как фильтр узлов. У объекта `NodeFilter` есть единственный метод `acceptNode()`, который возвращает значение `NodeFilter.FILTER_ACCEPT`, если конкретный узел нужно посетить, или `NodeFilter.FILTER_SKIP` в противном случае. Поскольку тип `NodeFilter` является абстрактным, создать его экземпляр невозможно. Вместо этого просто создайте объект с методом `acceptNode()` и передайте его в метод `createNodeIterator()`. Например, следующий код отфильтровывает элементы `<p>`:

```
var filter = {
    acceptNode: function(node){
        return node.tagName.toLowerCase() == "p" ?
               NodeFilter.FILTER_ACCEPT :
               NodeFilter.FILTER_SKIP;
    }
};

var iterator = document.createNodeIterator(root, NodeFilter.SHOW_ELEMENT,
                   filter, false);
```

Третьим аргументом также может быть функция, аналогичная методу `acceptNode()`:

```
var filter = function(node){
         return node.tagName.toLowerCase() == "p" ?
                NodeFilter.FILTER_ACCEPT :
```

```
                NodeFilter.FILTER_SKIP;
};

var iterator = document.createNodeIterator(root, NodeFilter.SHOW_ELEMENT,
                    filter, false);
```

Этот формат используется в JavaScript чаще других, потому что он проще и больше похож на другой JavaScript-код. Если фильтр не требуется, третий аргумент должен иметь значение `null`.

Создать простой объект `NodeIterator`, посещающий узлы всех типов, можно следующим образом:

```
var iterator = document.createNodeIterator(document, NodeFilter.SHOW_ALL,
                    null, false);
```

Два основных метода объекта `NodeIterator` называются `nextNode()` и `previousNode()`. Первый из них делает один шаг вперед при обходе DOM-поддерева в глубину, а второй возвращается на шаг назад. При создании объекта `NodeIterator` его внутренний указатель указывает на корневой узел, который и возвращается при первом вызове `nextNode()`. Когда обход достигает последнего узла в DOM-поддереве, метод `nextNode()` возвращает `null`. Метод `previousNode()` работает подобным образом. При завершении обхода он возвращает корневой узел, а при следующем вызове — значение `null`.

Рассмотрим такой HTML-код:

Листинг NodeIteratorExample1.htm


```
<div id="div1">
    <p><b>Hello</b> world!</p>
    <ul>
        <li>List item 1</li>
        <li>List item 2</li>
        <li>List item 3</li>
    </ul>
</div>
```

Предположим, нам нужно обойти все элементы внутри элемента `<div>`. Это можно сделать следующим образом:

Листинг NodeIteratorExample1.htm

```
var div = document.getElementById("div1");
var iterator = document.createNodeIterator(div, NodeFilter.SHOW_ELEMENT,
                    null, false);

var node = iterator.nextNode();
while (node !== null) {
    alert(node.tagName);    // вывод имени тега
    node = iterator.nextNode();
}
```

Первый вызов метода `nextNode()` в этом примере возвращает элемент `<p>`. Поскольку при достижении конца DOM-поддерева метод `nextNode()` возвращает `null`, мы используем это условие в цикле `while`. Если запустить этот код, появятся оповещения со следующими именами тегов:

```
DIV
P
B
UL
LI
LI
LI
```

Если при обходе нужно возвратить только элементы `<li>`, это можно сделать с помощью фильтра:

Листинг NodeIteratorExample2.htm

```
var div = document.getElementById("div1");
var filter = function(node){
    return node.tagName.toLowerCase() == "li" ?
        NodeFilter.FILTER_ACCEPT :
        NodeFilter.FILTER_SKIP;
};

var iterator = document.createNodeIterator(div, NodeFilter.SHOW_ELEMENT,
                   filter, false);

var node = iterator.nextNode();
while (node !== null) {
    alert(node.tagName);    // вывод имени тега
    node = iterator.nextNode();
}
```

В этом примере итератор возвращает только элементы `<li>`.

При работе с DOM-структурой методы `nextNode()` и `previousNode()` используют внутренний указатель объекта `NodeIterator`, который отражает любые изменения структуры.

В Firefox до версии 3.5 метод createNodeIterator() не реализован, но доступен метод createTreeWalker(), описываемый в следующем разделе.

Тип TreeWalker

`TreeWalker` — это улучшенная версия типа `NodeIterator`, которая поддерживает весь его функционал, включая методы `nextNode()` и `previousNode()`, и расширяет его следующими методами для обхода DOM-структуры в разных направлениях:

- `parentNode()` — переходит к родительскому узлу текущего узла;
- `firstChild()` — переходит к первому дочернему узлу текущего узла;

- `lastChild()` — переходит к последнему дочернему узлу текущего узла;
- `nextSibling()` — переходит к следующему узлу текущего уровня;
- `previousSibling()` — переходит к предыдущему узлу текущего уровня.

Создать объект `TreeWalker` можно методом `document.createTreeWalker()`, который принимает те же аргументы, что и метод `document.createNodeIterator()`: корневой узел обхода, типы отображаемых узлов, фильтр и логическое значение, указывающее, нужно ли развертывать ссылки на сущности. Благодаря такому подобию объект `TreeWalker` всегда можно использовать вместо `NodeIterator`, например:

Листинг TreeWalkerExample1.htm


```
var div = document.getElementById("div1");
var filter = function(node){
    return node.tagName.toLowerCase() == "li" ?
        NodeFilter.FILTER_ACCEPT :
        NodeFilter.FILTER_SKIP;
};

var walker = document.createTreeWalker(div, NodeFilter.SHOW_ELEMENT,
                    filter, false);

var node = iterator.nextNode();
while (node !== null) {
    alert(node.tagName);    // вывод имени тега
    node = iterator.nextNode();
}
```

Одно из различий этих типов заключается в значениях, которые может возвращать фильтр. Кроме `NodeFilter.FILTER_ACCEPT` и `NodeFilter.FILTER_SKIP` доступно также значение `NodeFilter.FILTER_REJECT`. При использовании с объектом `NodeIterator` значения `NodeFilter.FILTER_SKIP` и `NodeFilter.FILTER_REJECT` указывают на необходимость пропустить узел. При использовании с объектом `TreeWalker` значение `NodeFilter.FILTER_SKIP` указывает на необходимость пропустить узел и перейти к следующему узлу в поддереве, тогда как `NodeFilter.FILTER_REJECT` пропускает узел и все его поддерево. Например, если в предыдущем фрагменте кода возвращать из фильтра значение `NodeFilter.FILTER_REJECT`, а не `NodeFilter.FILTER_SKIP`, никакие узлы посещены не будут. Дело в том, что первым возвращаемым элементом является `<div>`, у которого нет тега с именем `"li"`, поэтому для него возвращается значение `NodeFilter.FILTER_REJECT`, указывающее, что все поддерево должно быть пропущено. Поскольку элемент `<div>` является корнем обхода, на этом обход завершается.

Конечно, истинная мощь объекта `TreeWalker` — это возможность перемещения по DOM-структуре. С его помощью можно обойти элементы `<li>` без использования фильтра:

Листинг TreeWalkerExample2.htm


```
var div = document.getElementById("div1");
var walker = document.createTreeWalker(div, NodeFilter.SHOW_ELEMENT, null,
                                       false);
```

```
walker.firstChild();     // переход к элементу <p>
walker.nextSibling();    // переход к элементу <ul>

var node = walker.firstChild();    // переход к первому элементу <li>
while (node !== null) {
    alert(node.tagName);
    node = walker.nextSibling();
}
```

Поскольку нам известно, где в документе находятся элементы `<li>`, можно добраться до них, вызвав метод `firstChild()` для перехода к элементу `<p>`, метод `nextSibling()` для перехода к элементу `<ul>` и метод `firstChild()` для перехода к первому элементу `<li>`. Имейте в виду, что из-за второго аргумента, переданного в метод `createTreeWalker()`, объект `TreeWalker` в этом примере возвращает только элементы. Затем метод `nextSibling()` перебирает в цикле элементы `<li>`, возвращая при их исчерпании значение `null`.

У типа `TreeWalker` также есть свойство `currentNode`, указывающее последний узел, возвращенный любым из методов обхода. Задав это свойство, можно изменить место возобновления обхода, например:

```
var node = walker.nextNode();
alert(node === walker.currentNode);  // true
walker.currentNode = document.body;  // изменение места возобновления обхода
```

В сравнении с типом `NodeIterator` тип `TreeWalker` обеспечивает больше гибкости при обходе DOM. В Internet Explorer 8 и более ранних версий нет эквивалентного типа, поэтому кроссбраузерные решения для обхода редки.

Диапазоны

Еще большего контроля над страницей можно добиться с помощью так называемых диапазонов, определенных в модуле DOM Level 2 Traversal and Range. Диапазон можно использовать для выделения части документа независимо от границ узлов (выделение выполняется неявно и незаметно для пользователя). Диапазоны полезны, если обычные DOM-манипуляции недостаточно конкретны. DOM-диапазоны поддерживаются в Firefox, Opera, Safari и Chrome, а в Internet Explorer они реализованы иначе.

Диапазоны в DOM

В DOM Level 2 для типа `Document` определен метод `createRange()`. В браузерах, соответствующих требованиям DOM, этот метод принадлежит объекту `document`. Чтобы выяснить, поддерживает ли браузер диапазоны, можно проверить наличие этого метода или воспользоваться методом `hasFeature()`:

```
var supportsRange = document.implementation.hasFeature("Range", "2.0");
var alsoSupportsRange = (typeof document.createRange == "function");
```

Если DOM-диапазон поддерживается, его можно создать методом `createRange()`:

```
var range = document.createRange();
```

Подобно узлам, новый диапазон связывается с документом, для которого был создан. Это означает, что его можно использовать для неявного выделения частей только этого, но не других документов. Как только диапазон создан и настроен, можно выполнять различные операции с его содержимым, что делает возможным более тонкое манипулирование базовым DOM-деревом.

Каждый диапазон представляется экземпляром типа `Range`, у которого есть ряд свойств и методов. Указанные далее свойства определяют расположение диапазона в документе.

- `startContainer` — узел, в котором начинается диапазон (родительский узел первого узла в выделенном фрагменте).
- `startOffset` — смещение начала диапазона в узле `startContainer`. Если узел `startContainer` является текстовым, узлом комментария или узлом `CData`, свойство `startOffset` указывает количество знаков, пропускаемых перед началом диапазона, иначе — индекс первого дочернего узла в диапазоне.
- `endContainer` — узел, в котором завершается диапазон (родительский узел последнего узла в выделенном фрагменте).
- `endOffset` — смещение конца диапазона в узле `endContainer` (действуют те же правила, что и для узла `startOffset`).
- `commonAncestorContainer` — наиболее глубокий узел в документе, являющийся предком `startContainer` и `endContainer`.

Эти свойства заполняются при определении диапазона в конкретной позиции в документе.

Простое выделение с помощью DOM-диапазонов

Самый простой способ выделить часть документа с помощью диапазона — это использовать метод `selectNode()` или `selectNodeContents()`. Каждый из них принимает DOM-узел и заполняет диапазон информацией из этого узла. Метод `selectNode()` выделяет весь узел вместе с дочерними узлами, а метод `selectNodeContents()` — только дочерние узлы. Рассмотрим, например, такой HTML-код:

```
<!DOCTYPE html>
<html>
    <body>
        <p id="p1"><b>Hello</b> world!</p>
    </body>
</html>
```

Для доступа к этому фрагменту можно использовать следующий JavaScript-код:

Листинг DOMRangeExample.htm


```
var range1 = document.createRange(),
    range2 = document.createRange(),
    p1 = document.getElementById("p1");
range1.selectNode(p1);
range2.selectNodeContents(p1);
```

Два диапазона в этом примере содержат разные разделы документа: `range1` — элемент `<p>` и все его дочерние элементы, а `range2` — элемент `<b>` и текстовые узлы `"Hello"` и `" world!"` (рис. 12.6).

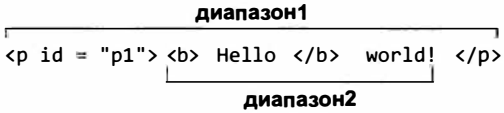

Рис. 12.6

При вызове метода `selectNode()` свойства `startContainer`, `endContainer` и `commonAncestorContainer` содержат родительский узел узла, переданного в метод (`document.body` в данном случае). Значение `startOffset` равно индексу конкретного узла в коллекции `childNodes` родительского узла (1 в данном примере, потому что браузеры, соответствующие требованиям DOM, интерпретируют свободное пространство как текстовый узел), а значение `endOffset` на единицу превышает `startOffset`, потому что выделен только один узел.

При вызове метода `selectNodeContents()` свойства `startContainer`, `endContainer` и `commonAncestorContainer` содержат узел, переданный в метод, или `<p>` в нашем случае. Свойство `startOffset` всегда равно 0, так как диапазон начинается с первого дочернего узла опорного узла, а свойство `endOffset` равно количеству дочерних узлов (`node.childNodes.length`), или 2 в примере.

Для более точного управления выделением узлов можно использовать перечисленные методы.

- `setStartBefore(`*опорныйУзел*`)` — задает начальную точку диапазона перед опорным узлом, который становится первым узлом в диапазоне. Свойству `startContainer` назначается родительский узел опорного узла, а свойству `startOffset` — индекс опорного узла в коллекции `childNodes` его родительского узла.
- `setStartAfter(`*опорныйУзел*`)` — задает начальную точку диапазона после опорного узла, который не включается в диапазон, а первым выделяемым узлом становится следующий узел того же уровня. Свойству `startContainer` назначается родительский узел опорного узла, а свойству `startOffset` — индекс опорного узла в коллекции `childNodes` его родительского узла, увеличенный на единицу.

- `setEndBefore(`*опорныйУзел*`)` — задает конечную точку диапазона перед опорным узлом, который не включается в диапазон. Свойству `endContainer` назначается родительский узел опорного узла, а свойству `endOffset` — индекс опорного узла в коллекции `childNodes` его родительского узла.
- `setEndAfter(`*опорныйУзел*`)` — задает конечную точку диапазона перед опорным узлом, который становится последним узлом в диапазоне. Свойству `endContainer` назначается родительский узел опорного узла, а свойству `endOffset` — индекс опорного узла в коллекции `childNodes` его родительского узла, увеличенный на единицу.

При использовании всех этих методов свойства задаются автоматически, но при желании их можно назначать вручную для более точного выделения диапазонов.

Сложное выделение с помощью DOM-диапазонов

Для создания сложных диапазонов нужно использовать методы `setStart()` и `setEnd()`, которые принимают два аргумента: опорный узел и смещение. В методе `setStart()` опорный узел назначается свойству `startContainer`, а смещение — свойству `startOffset`. В методе `setEnd()` опорный узел назначается свойству `endContainer`, а смещение — свойству `endOffset`.

С помощью этих методов можно имитировать методы `selectNode()` и `selectNodeContents()`, например:

Листинг DOMRangeExample2.htm


```
var range1 = document.createRange(),
    range2 = document.createRange(),
    p1 = document.getElementById("p1"),
    p1Index = -1,
    i, len;
for (i=0, len=p1.parentNode.childNodes.length; i < len; i++) {
    if (p1.parentNode.childNodes[i] == p1) {
        p1Index = i;
        break;
    }
}

range1.setStart(p1.parentNode, p1Index);
range1.setEnd(p1.parentNode, p1Index + 1);
range2.setStart(p1, 0);
range2.setEnd(p1, p1.childNodes.length);
```

Заметьте, что для выделения узла `p1` с помощью диапазона `range1` нужно сначала определить его индекс в коллекции `childNodes` родительского узла. Для выделения содержимого узла с помощью диапазона `range2` вычислять что-либо не требуется — можно использовать в методах `setStart()` и `setEnd()` значения, предлагаемые по умолчанию. Хотя имитировать методы `selectNode()` и `selectNodeContents()` может

быть полезно, реальная мощь методов `setStart()` и `setEnd()` — это возможность частичного выделения узлов.

Предположим, что нам нужно выделить в предыдущем HTML-коде текст с букв `"llo"` в слове `"Hello"` до буквы `"o"` в слове `"world!"`. Сделать это довольно легко. Сначала нужно получить ссылки на соответствующие узлы:

Листинг DOMRangeExample3.htm


```
var p1 = document.getElementById("p1"),
helloNode = p1.firstChild.firstChild,
worldNode = p1.lastChild
```

Текстовый узел `"Hello"` приходится внуком узлу `<p>` и является дочерним по отношению к узлу `<b>`, так что его можно получить с помощью свойства `p1.firstChild.firstChild` (свойство `p1.firstChild` возвращает узел `<b>`). Текстовый узел `"world!"` — это второй (и последний) дочерний узел `<p>`, которому соответствует свойство `p1.lastChild`. Получив узлы, нужно создать диапазон и задать его границы:

Листинг DOMRangeExample3.htm

```
var range = document.createRange();
range.setStart(helloNode, 2);
range.setEnd(worldNode, 3);
```

Поскольку выделение должно начаться после буквы `"e"` в слове `"Hello"`, мы передаем в метод `setStart()` узел `helloNode` и смещение 2 (буква `"H"` находится в позиции 0). Чтобы указать конец диапазона, мы передаем в метод `setEnd()` узел `worldNode` и смещение 3, задающее первый знак, который выделять не следует, то есть `"r"` (позиции 0 соответствует пробел). Содержимое диапазона показано на рис. 12.7.

Рис. 12.7

Узлы `helloNode` и `worldNode` являются текстовыми, поэтому они становятся свойствами `startContainer` и `endContainer`, а свойства `startOffset` и `endOffset` определяют смещения в тексте, а не дочерние узлы (это происходит, если в методы передается элемент). Свойству `commonAncestorContainer` назначается элемент `<p>` — первый общий предок обоих узлов.

Разумеется, выделение фрагментов документа полезно, только если с ними можно что-нибудь сделать. Сейчас мы это обсудим.

Работа с контентом DOM-диапазона

При создании диапазона неявно создается узел фрагмента документа, к которому присоединяются все узлы в выделенной области. Чтобы это было возможно, содержимое диапазона должно быть синтаксически правильным. В предыдущем примере диапазон

определяет недопустимую DOM-структуру, потому что выделение начинается в одном текстовом узле и завершается в другом. Однако диапазоны распознают отсутствие открывающих и закрывающих тегов и могут воссоздавать допустимую DOM-структуру.

В предыдущем примере в выделенном фрагменте отсутствует открывающий тег `<b>`, поэтому диапазон динамически добавляет его вместе с закрывающим тегом `</b>` для строки `"He"`, заменяя DOM-структуру следующим фрагментом:

```
<p><b>He</b><b>llo</b> world!</p>
```

Кроме того, текстовый узел `" world!"` разделяется на два, из которых первый содержит символы `" wo"`, а второй — `"rld!"`. Итоговое DOM-дерево вместе с содержимым фрагмента документа показано на рис. 12.8.

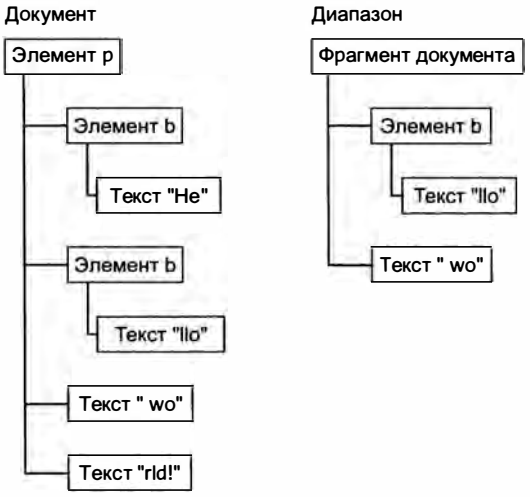

Рис. 12.8

После создания диапазона можно выполнять различные операции над его содержимым (имейте в виду, что все узлы во внутреннем фрагменте документа диапазона просто указывают на узлы в документе).

Первый метод, `deleteContents()`, просто удаляет содержимое диапазона из документа, например:

Листинг DOMRangeExample4.htm


```
var p1 = document.getElementById("p1"),
    helloNode = p1.firstChild.firstChild,
    worldNode = p1.lastChild,
    range = document.createRange();

range.setStart(helloNode, 2);
range.setEnd(worldNode, 3);

range.deleteContents();
```

Этот код дает следующий результат:

```
<p><b>He</b>rld!</p>
```

Поскольку при выделении диапазона DOM-структура была подкорректирована, она остается синтаксически правильной даже после удаления содержимого.

Метод `extractContents()` похож на `deleteContents()` тем, что он также удаляет выделенный диапазон из документа. Разница между ними в том, что `extractContents()` возвращает соответствующий диапазону фрагмент документа, что позволяет вставить его в другое место, например:

Листинг DOMRangeExample5.htm


```
var p1 = document.getElementById("p1"),
    helloNode = p1.firstChild.firstChild,
    worldNode = p1.lastChild,
    range = document.createRange();

range.setStart(helloNode, 2);
range.setEnd(worldNode, 3);

var fragment = range.extractContents();
p1.parentNode.appendChild(fragment);
```

Этот код извлекает фрагмент и добавляет его в конец элемента `<body>` (помните, что при передаче фрагмента документа в метод `appendChild()` добавляются только его дочерние узлы, но не сам фрагмент). В итоге получается такой HTML-код:

```
<p><b>He</b>rld!</p>
<b>llo</b> wo
```

С помощью метода `cloneContents()` можно создать копию диапазона, оставив его на месте. Скопированный фрагмент затем можно вставить в другое место:

Листинг DOMRangeExample6.htm


```
var p1 = document.getElementById("p1"),
    helloNode = p1.firstChild.firstChild,
    worldNode = p1.lastChild,
    range = document.createRange();

range.setStart(helloNode, 2);
range.setEnd(worldNode, 3);

var fragment = range.cloneContents();
p1.parentNode.appendChild(fragment);
```

Этот метод очень похож на `extractContents()`, потому что оба они возвращают фрагмент документа, однако в случае метода `cloneContents()` он содержит копии узлов в диапазоне, а не фактические узлы. Предыдущий фрагмент генерирует следующий HTML-код:

```
<p><b>Hello</b> world!</p>
<b>llo</b> wo
```

Важно отметить, что из-за разделения узлов синтаксис документа не станет правильным, пока не будет вызван один из этих методов. Исходный HTML-документ остается неизменным до изменения DOM-структуры.

Вставка контента DOM-диапазона

С помощью диапазонов можно не только удалять и клонировать контент. Так, метод `insertNode()` позволяет вставить узел в начало выделенного диапазона. Предположим, например, что нам нужно вставить перед HTML-кодом предыдущего примера следующий HTML-код:

```
<span style="color: red">Inserted text</span>
```

Это можно сделать следующим образом:

Листинг DOMRangeExample7.htm


```
var p1 = document.getElementById("p1"),
    helloNode = p1.firstChild.firstChild,
    worldNode = p1.lastChild,
    range = document.createRange();

range.setStart(helloNode, 2);
range.setEnd(worldNode, 3);

var span = document.createElement("span");
span.style.color = "red";
span.appendChild(document.createTextNode("Inserted text"));
range.insertNode(span);
```

В результате будет сгенерирован такой фрагмент:

```
<p id="p1"><b>He<span style="color: red">Inserted text</span>llo</b> world</p>
```

Обратите внимание, что тег `<span>` вставляется непосредственно перед строкой `"llo"` в слове `"Hello"`, которая является первой частью выделенного диапазона. Заметьте также, что элементов `<b>` в исходном HTML-коде не стало больше или меньше, потому что мы не используем ни один из методов, упомянутых в предыдущем разделе. Эту методику можно применять для вставки полезных сведений, например для добавления изображений рядом со ссылками, которые открывают новые окна.

Кроме содержимого диапазона можно вставить в документ и контент, охватывающий диапазон. Для этого используется метод `surroundContents()`, принимающий узел, в который должен быть заключен диапазон. За кулисами при этом происходит следующее.

1. Содержимое диапазона извлекается (как при вызове метода `extractContents()`).
2. Указанный узел вставляется в документ там, где был диапазон.
3. Содержимое фрагмента документа добавляется в новый узел.

Этот метод полезен для выделения слов на веб-странице, например:

Листинг DOMRangeExample8.htm


```
var p1 = document.getElementById("p1"),
    helloNode = p1.firstChild.firstChild,
    worldNode = p1.lastChild,
    range = document.createRange();

range.selectNode(helloNode);

var span = document.createElement("span");
span.style.backgroundColor = "yellow";
range.surroundContents(span);
```

Этот код выделяет диапазон желтым фоновым цветом, при этом генерируется следующий HTML-код:

```
<p><b><span style="background-color:yellow">Hello</span></b> world!</p>
```

Чтобы можно было вставить элемент `<span>`, диапазон должен содержать полноценный DOM-фрагмент, но не частично выделенные узлы.

Свертывание DOM-диапазона

Если диапазон не выделяет никакой части документа, говорят, что он *свернут* (collapsed). Свертывание диапазона напоминает работу текстового поля. Если в нем есть текст, вы может выделить целое слово с помощью мыши. Однако если щелкнуть левой кнопкой мыши еще раз, выделение исчезнет, а курсор установится между двумя буквами. При свертывании диапазона он располагается между частями документа в начале выделенного фрагмента или в его конце. На рис. 12.9 показано, что происходит при свертывании диапазона.

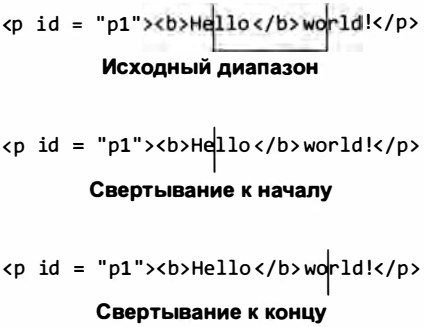

Рис. 12.9

Свернуть диапазон можно методом `collapse()`, который принимает единственный аргумент — логическое значение, указывающее, в какую сторону нужно свернуть

диапазон. Если оно равно `true`, диапазон свертывается к началу, иначе — к концу. Узнать, не свернут ли уже диапазон, можно с помощью свойства `collapsed`:

```
range.collapse(true);          // свертывание к началу
alert(range.collapsed);        // "true"
```

Проверив, свернут ли диапазон, можно узнать, находятся ли два узла рядом. Возьмем для примера следующий HTML-код:

```
<p id="p1">Paragraph 1</p><p id="p2">Paragraph 2</p>
```

Если вы не знаете точную разметку подобного кода (например, если он был сгенерирован автоматически), можно попробовать создать диапазон:

```
var p1 = document.getElementById("p1"),
    p2 = document.getElementById("p2"),
    range = document.createRange();
range.setStartAfter(p1);
range.setStartBefore(p2);
alert(range.collapsed);        // "true"
```

В данном случае созданный диапазон свернут, потому что между узлами `p1` и `p2` ничего нет.

Сравнение DOM-диапазонов

Если диапазонов более одного, с помощью метода `compareBoundaryPoints()` можно определить, есть ли у них общие границы (начальная или конечная). Этот метод принимает два аргумента: сравниваемый диапазон и способ сравнения. Вторым аргументом может быть одна из следующих констант:

- `Range.START_TO_START (0)` — сравнивается начало первого диапазона с началом второго;
- `Range.START_TO_END (1)` — сравнивается начало первого диапазона с концом второго;
- `Range.END_TO_END (2)` — сравнивается конец первого диапазона с концом второго;
- `Range.END_TO_START (3)` — сравнивается конец первого диапазона с началом второго.

Метод `compareBoundaryPoints()` возвращает −1, если граница первого диапазона предшествует границе второго; 0, если границы совпадают, и 1, если граница первого диапазона следует за границей второго, например:

Листинг DOMRangeExample9.htm

```
var range1 = document.createRange();
var range2 = document.createRange();
var p1 = document.getElementById("p1");
```

```
range1.selectNodeContents(p1);
range2.selectNodeContents(p1);
range2.setEndBefore(p1.lastChild);

alert(range1.compareBoundaryPoints(Range.START_TO_START, range2));    // 0
alert(range1.compareBoundaryPoints(Range.END_TO_END, range2));        // 1
```

В этом коде начала двух диапазонов совпадают, потому что мы создаем их одинаковыми вызовами метода `selectNodeContents()`; соответственно, в первый раз метод `compareBoundaryPoints()` возвращает 0. Во второй раз он возвращает 1, потому что после изменения конца второго диапазона методом `setEndBefore()` он предшествует концу первого диапазона (рис. 12.10).

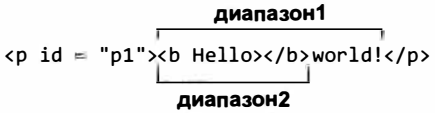

Рис. 12.10

Клонирование DOM-диапазонов

Диапазон можно клонировать методом `cloneRange()`, который создает его точную копию:

```
var newRange = range.cloneRange();
```

Новый диапазон содержит все свойства исходного, а изменения его конечных точек никак не влияют на оригинал.

Очистка

Завершив работу с диапазоном, следует вызвать метод `detach()`, который разрывает его связь с документом, а затем присвоить диапазону значение `null`, чтобы сборщик мусора мог возвратить память системе:

```
range.detach();        // разрыв связи диапазона с документом
range = null;          // уничтожение ссылки
```

Это оптимальный способ завершения работы с диапазоном. Когда его связь с документом разорвана, использовать диапазон нельзя.

Диапазоны в Internet Explorer 8 и более ранних версий

Internet Explorer поддерживает DOM-диапазоны только с девятой версии, но в более ранних версиях Internet Explorer (но не в других браузерах) реализована

похожая концепция — так называемые *текстовые диапазоны* (text ranges). Они предназначены для работы исключительно с текстом (необязательно с DOM-узлами), а создать их можно методом `createTextRange()`, который доступен только для элементов `<body>`, `<button>`, `<input>` и `<textarea>`, например:

```
var range = document.body.createTextRange();
```

Диапазон, созданный таким способом, можно использовать в любом месте страницы (тогда как диапазон, созданный для какого-либо из других упомянутых ранее элементов, ограничен этим элементом). Как и DOM-диапазоны, текстовые диапазоны в Internet Explorer поддерживают несколько операций.

Простое выделение с помощью IE-диапазонов

Самый простой способ выделить область страницы — это воспользоваться методом `findText()` диапазона. Он находит первый экземпляр указанной текстовой строки и заключает ее в диапазон. Если найти текст не удается, метод возвращает `false`, иначе — `true`. Рассмотрим следующий HTML-код:

```
<p id="p1"><b>Hello</b> world!</p>
```

Выделить слово `"Hello"` можно следующим образом:

Листинг IERangeExample1.htm

```
var range = document.body.createTextRange();
var found = range.findText("Hello");
```

Вторая строка кода добавляет в диапазон текст `"Hello"`. Убедиться в этом можно, прочитав свойство `text`, которое возвращает текст диапазона. Также можно проверить значение, возвращенное методом `findText()`, которое равно `true`, если текст был обнаружен, например:

```
alert(found);          // true
alert(range.text);     // "Hello"
```

У метода `findText()` может быть и второй аргумент — число, указывающее направление продолжения поиска. Отрицательное значение показывает, что поиск нужно продолжить назад от текущей позиции, а положительное — вперед. Например, найти первые два экземпляра слова `"Hello"` в документе можно так:

```
var found = range.findText("Hello");
var foundAgain = range.findText("Hello", 1);
```

На DOM-метод `selectNode()` в браузере Internet Explorer больше всего похож метод `moveToElementText()`, который принимает DOM-элемент и выделяет весь его текст, включая HTML-теги, например:

Листинг IERangeExample2.htm

```
var range = document.body.createTextRange();
var p1 = document.getElementById("p1");
range.moveToElementText(p1);
```

Если текстовый диапазон содержит HTML-код, с помощью свойства `htmlText` можно получить все его содержимое, включая HTML-код и текст:

```
alert(range.htmlText);
```

У IE-диапазонов нет других свойств, динамически обновляемых при изменении выделенного фрагмента, хотя метод `parentElement()` работает так же, как DOM-свойство `commonAncestorContainer`:

```
var ancestor = range.parentElement();
```

Родительский элемент всегда соответствует родительскому узлу выделенного текста.

Сложное выделение с помощью IE-диапазонов

Сложные диапазоны в Internet Explorer можно создавать, перемещая границы выделения на конкретные интервалы с помощью методов `move()`, `moveStart()`, `moveEnd()` и `expand()`. Каждый из них принимает два аргумента: единицу измерения и количество интервалов. Единицей измерения может быть одно из следующих строковых значений:

- `"character"` — перемещение границы на один символ;
- `"word"` — перемещение границы на одно слово (слово — это последовательность непробельных символов);
- `"sentence"` — перемещение границы на одно предложение (предложение — это последовательность символов, которая завершается точкой, вопросительным или восклицательным знаком);
- `"textedit"` — перемещение границы к началу или концу текущего выделенного диапазона.

Метод `moveStart()` перемещает на указанное количество интервалов начало диапазона, а метод `moveEnd()` — конец:

```
range.moveStart("word", 2);     // перемещение начала диапазона на два слова
range.moveEnd("character", 1); // перемещение конца диапазона на один символ
```

Для нормализации диапазона можно использовать метод `expand()`, при вызове которого любые блоки, выделенные частично, выделяются полностью. Например, можно выделить два символа внутри слова, а затем вызвать метод `expand("word")`, чтобы заключить в диапазон все слово.

Метод `move()` сначала свертывает диапазон (приравнивая его начало и конец), а затем перемещает его на указанное количество интервалов, например:

```
range.move("character", 5);    // перемещение диапазона на пять символов
```

После вызова метода `move()` начало и конец диапазона совпадают, так что для выделения содержимого нужно использовать метод `moveStart()` или `moveEnd()`.

Работа с контентом IE-диапазона

Для работы с содержимым диапазона в Internet Explorer можно использовать свойство `text` и метод `pasteHTML()`. С помощью свойства `text` можно не только прочитать, но и задать текстовое содержимое диапазона, например:

```
var range = document.body.createTextRange();
range.findText("Hello");
range.text = "Howdy";
```

Если выполнить этот код для прежнего примера со строкой `"Hello world!"`, будет сгенерирован такой HTML-код:

```
<p id="p1"><b>Howdy</b> world!</p>
```

Обратите внимание, что все HTML-теги остались неизменными.

Для вставки HTML-кода в диапазон можно использовать метод `pasteHTML()`:

Листинг IERangeExample3.htm


```
var range = document.body.createTextRange();
range.findText("Hello");
range.pasteHTML("<em>Howdy</em>");
```

В результате будет получен следующий HTML-код:

```
<p id="p1"><b><em>Howdy</em></b> world!</p>
```

Если диапазон содержит HTML-разметку, использовать метод `pasteHTML()` не следует, потому что он может возвратить HTML-код с неправильным синтаксисом.

Свертывание IE-диапазона

У диапазонов в Internet Explorer есть метод `collapse()`, который работает точно так же, как DOM-метод: получив аргумент `true`, он свертывает диапазон к началу, а получив `false` — к концу, например:

```
range.collapse(true);    // свертывание к началу
```

К сожалению, свойство `collapsed`, которое указывало бы, свернут ли уже диапазон, в Internet Explorer отсутствует. Вместо этого нужно использовать свойство `boundingWidth`, которое возвращает ширину диапазона в пикселях. Если оно имеет значение 0, диапазон свернут:

```
var isCollapsed = (range.boundingWidth == 0);
```

Другие сведения о расположении диапазона можно получить с помощью свойств `boundingHeight`, `boundingLeft` и `boundingTop`, но они менее полезны, чем `boundingWidth`.

Сравнение IE-диапазонов

Метод `compareEndPoints()` в Internet Explorer похож на DOM-метод `compareBoundaryPoints()`. Он принимает два аргумента: тип сравнения и сравниваемый диапазон. Тип сравнения задается одним из строковых значений — `"StartToStart"`, `"StartToEnd"`, `"EndToEnd"` и `"EndToStart"`, — которые аналогичны соответствующим значениям для DOM-диапазонов.

Подобно DOM-методу, метод `compareEndPoints()` возвращает –1, если граница первого диапазона предшествует границе второго; 0, если они совпадают, и 1, если граница первого диапазона следует за границей второго. Рассмотрим следующий фрагмент, который на основе уже знакомого нам HTML-кода создает два диапазона — с текстом `"Hello world!"` (включая теги `<b>`) и словом `"Hello"`:

Листинг IERangeExample5.htm

```
var range1 = document.body.createTextRange(),
    range2 = document.body.createTextRange();

range1.findText("Hello world!");
range2.findText("Hello");

alert(range1.compareEndPoints("StartToStart", range2));    // 0
alert(range1.compareEndPoints("EndToEnd", range2));        // 1
```

Оба диапазона начинаются в одном месте, поэтому при их сравнении методом `compareEndPoints()` возвращается значение 0. Конец второго диапазона предшествует концу первого, поэтому второй вызов `compareEndPoints()` возвращает 1.

В Internet Explorer есть также два дополнительных метода сравнения диапазонов: метод `isEqual()` определяет, совпадают ли у двух диапазонов обе границы, а метод `inRange()` показывает, находится ли один диапазон внутри другого. Рассмотрим пример:

Листинг IERangeExample6.htm


```
var range1 = document.body.createTextRange();
var range2 = document.body.createTextRange();
range1.findText("Hello world!");
range2.findText("Hello");
alert("range1.isEqual(range2): " + range1.isEqual(range2));    // false
alert("range1.inRange(range2): " + range1.inRange(range2));    // true
```

Для демонстрации методов здесь используются те же диапазоны, что и в предыдущем примере. Их конечные границы различаются, поэтому вызов `isEqual()` возвращает `false`. В то же время второй диапазон находится внутри первого, так как заканчивается раньше, а начала обоих диапазонов совпадают. Соответственно, метод `inRange()` возвращает `true`.

Клонирование IE-диапазона

Текстовые диапазоны можно клонировать в Internet Explorer с помощью метода `duplicate()`:

```
var newRange = range.duplicate();
```

Все свойства исходного диапазона при этом переносятся в новый.

Резюме

Спецификации DOM Level 2 определяют несколько модулей, расширяющих функционал DOM Level 1. Модуль DOM Level 2 Core добавляет к некоторым DOM-типам методы поддержки пространств имен, но эти изменения задействуются только при работе с XML- и XHTML-документами и не используются в HTML-коде. Новые возможности, не связанные с XML-пространствами имен, включают методы для программного создания экземпляров `Document` и `DocumentType`.

Модуль DOM Level 2 Style описывает, как работать со стилями элементов.

- У каждого элемента есть объект `style`, с помощью которого можно определять и изменять встроенные стили.
- Для определения вычисляемого стиля элемента, в том числе всех применимых к нему CSS-правил, можно использовать метод `getComputedStyle()`.
- Internet Explorer не поддерживает этот метод, но предоставляет свойство `currentStyle`, которое возвращает те же сведения. Оно доступно для всех элементов.
- Для доступа к таблицам стилей можно также использовать коллекцию `document.styleSheets`.
- Интерфейс таблиц стилей поддерживается всеми браузерами, кроме Internet Explorer 8 и более ранних версий, где реализованы другие свойства и методы, сравнимые по функционалу с DOM.

Модуль DOM Level 2 Traversals and Range определяет способы взаимодействия с DOM-структурой.

- С помощью объектов `NodeIterator` и `TreeWalker` можно обходить DOM-дерево в глубину.

- Интерфейс более простого типа `NodeIterator` позволяет перемещаться лишь вперед и назад на один шаг. Интерфейс `TreeWalker` поддерживает перемещение по DOM-структуре в этих и во всех других направлениях, включая переходы к родительским, одноуровневым и дочерним узлам.
- Диапазоны позволяют выделять части DOM-структуры для выполнения каких-либо операций над ними.
- С помощью диапазонов можно удалять выделенные части содержимого с сохранением синтаксически правильной структуры документа и клонировать их.
- В Internet Explorer 8 и более ранних версий модуль DOM Level 2 Traversals and Range не поддерживается, но доступны фирменные текстовые диапазоны, которые можно использовать для простого манипулирования текстом. Internet Explorer 9 полностью поддерживает DOM-обходы.

13 События

- Распространение событий
- Обработчики событий
- Типы событий

Взаимодействие JavaScript с HTML осуществляется посредством *событий* (events), которые сигнализируют, что в документе или окне браузера произошло что-то, что может нас заинтересовать. На события можно подписаться с помощью *слушателей* (listeners), называемых также обработчиками, которые выполняются только при возникновении события. Эта модель, в традиционном программировании реализуемая с помощью *паттерна Наблюдатель* (observer pattern), позволяет ослабить связь между поведением страницы (определенным на JavaScript) и ее видом (описанным с помощью HTML и CSS).

События, впервые реализованные в Internet Explorer 3 и Netscape Navigator 2, позволяли частично обрабатывать в браузере формы еще до отправки данных серверу. Ко времени выпуска Internet Explorer 4 и Netscape 4 каждый браузер предоставлял похожие, но разные API, которые просуществовали еще несколько поколений. В DOM Level 2 была предпринята первая попытка стандартизировать API событий DOM. Основные части спецификации DOM Level 2 Events реализованы в Internet Explorer 9, Firefox, Opera, Safari и Chrome. Internet Explorer 8 стал последним популярным браузером, в котором использовалась исключительно фирменная система событий.

Система событий браузера непроста. Хотя спецификация DOM Level 2 Events реализована во всех основных браузерах, она охватывает не все типы событий. BOM поддерживает собственные события, которые связаны с DOM-событиями запутанными отношениями из-за длительного отсутствия документации (хотя в HTML5 эта ситуация отчасти исправлена). Расширение API событий DOM в DOM Level 3 только добавило разработчикам головной боли. Иногда работать с событиями сравнительно просто, а иногда крайне сложно, но без понимания базовых концепций в любом случае не обойтись.

Распространение событий

Приступив к созданию веб-браузеров четвертого поколения (Internet Explorer 4 и Netscape Communicator 4), разработчики столкнулись с интересным вопросом: как понять, какой части страницы принадлежит то или иное конкретное событие? Чтобы понять проблему, представьте концентрические окружности на листе бумаги. Пометив центр одной из них, вы автоматически пометите центры и всех остальных окружностей. Разработчики из обеих групп смотрели на проблему точно так же. Когда пользователь щелкает на элементе управления, рассудили они, он щелкает также на его контейнере и на странице в целом.

Однако в вопросе *распространения событий* (event flow), то есть порядка, в котором события поступают на страницу, их взгляды разошлись. В Internet Explorer было реализовано так называемое всплытие событий, а в Netscape Communicator — перехват событий.

Всплытие событий

При *всплытии событий* (event bubbling), реализованном в Internet Explorer, событие срабатывает у наиболее конкретного элемента (самого глубокого узла в дереве документа), а затем поднимается по иерархии до наименее конкретного узла (самого документа). Рассмотрим следующую HTML-страницу:

```
<!DOCTYPE html>
<html>
<head>
    <title>Event Bubbling Example</title>
</head>
<body>
    <div id="myDiv">Click Me</div>
</body>
</html>
```

Если щелкнуть на элементе `<div>` этой страницы, событие `click` будет возникать для элементов в следующем порядке:

1. `<div>`.
2. `<body>`.
3. `<html>`.
4. `document`.

Прежде всего, событие `click` возникнет для элемента `<div>`, на котором произошел щелчок, а затем — для каждого последующего узла в DOM-иерархии вплоть до объекта `document`. Этот процесс показан на рис. 13.1.

Всплытие событий поддерживают все современные браузеры, хотя его реализации немного различаются. В Internet Explorer 5.5 и более ранних версий пропускается элемент `<html>` (после элемента `<body>` событие возникает для элемента `document`).

В Internet Explorer 9, Firefox, Chrome и Safari всплытие продолжается до объекта `window`.

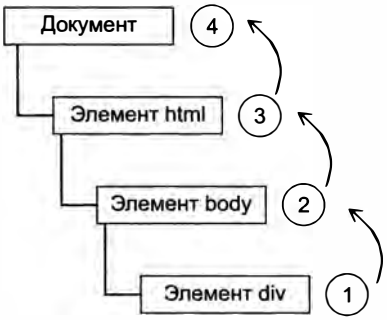

Рис. 13.1

Перехват событий

В Netscape Communicator вместо всплытия был реализован *перехват событий* (event capturing), при котором событие сначала возникает для наименее конкретного узла, а в конце — для наиболее конкретного. На самом деле этот механизм был разработан для того, чтобы событие можно было обработать, пока оно еще не достигло своего целевого элемента. В этом случае для предыдущей страницы при щелчке на элементе `<div>` событие `click` поочередно возникает для следующих элементов:

1. `document`.
2. `<html>`.
3. `<body>`.
4. `<div>`.

При перехвате событие `click` генерируется для документа, а затем продолжает спуск по DOM-дереву до своей фактической цели — элемента `<div>` (рис. 13.2).

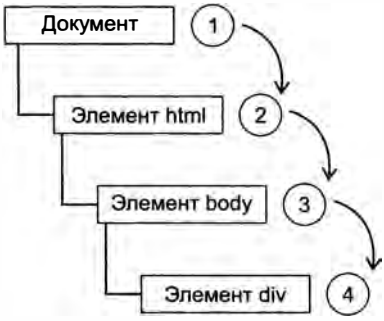

Рис. 13.2

Хотя первоначально эта модель распространения событий использовалась только в Netscape Communicator, теперь ее поддерживают Internet Explorer 9, Safari, Chrome, Opera и Firefox. Все они начинают перехват на уровне объекта `window`, хотя в спецификации DOM Level 2 Events сказано, что первым должен быть объект `document`.

Перехват событий не поддерживается в старых браузерах, поэтому рекомендуется применять его только в специальных ситуациях и использовать в основном всплытие событий.

Распространение DOM-событий

Процесс распространения событий, описанный в DOM Level 2 Events, включает три этапа: фаза перехвата, фаза цели и фаза всплытия. На первом этапе событие можно перехватить, если это требуется. Затем событие обрабатывается целевым элементом, а после этого всплывает, что позволяет выполнить какие-то заключительные действия в ответ на событие. В нашем прежнем примере щелчок на элементе `<div>` инициирует процесс распространения события, показанный на рис. 13.3.

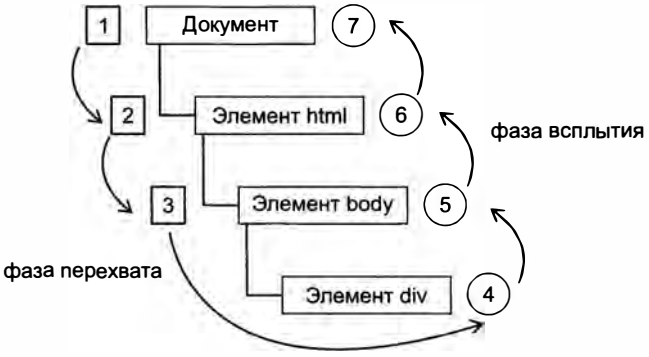

Рис. 13.3

В ходе распространения DOM-событий фактический целевой элемент (`<div>`) не получает его на этапе перехвата. Иначе говоря, событие распространяется от объекта `document` только до элемента `<body>`. После этого начинается следующий этап, когда событие генерируется для целевого элемента. В контексте обработки событий (см. далее) этот этап считается частью фазы всплытия. Затем событие всплывает, возвращаясь к узлу документа.

Большинство браузеров, поддерживающих распространение DOM-событий, имеют особенность. Хотя в спецификации DOM Level 2 Events сказано, что во время перехвата события оно не достигает целевого элемента, в Internet Explorer 9, Safari, Chrome, Firefox и Opera 9.5 и более поздних версий генерируют событие для целевого элемента на этапе перехвата. Это предоставляет еще одну возможность обработки события на уровне его целевого элемента.

Распространение DOM-событий поддерживается в Internet Explorer 9, Opera, Firefox, Chrome и Safari, но не в Internet Explorer 8 и более ранних версий.

Обработчики событий

События соответствуют определенным действиям, которые выполняет пользователь или сам браузер, и имеют имена вроде `click`, `load` и `mouseover`. Функция, выполняемая в ответ на событие, называется *обработчиком события* (event handler), или *слушателем события* (event listener). Имена таких функций начинаются с префикса `"on"`: например, обработчик события `click` имеет имя `onclick`, а обработчик события `load` называется `onload`. Назначать обработчики событиям можно несколькими способами.

HTML-обработчики событий

Каждому событию, поддерживаемому конкретным элементом, можно назначить обработчик, указав одноименный HTML-атрибут. Значением атрибута должен быть JavaScript-код. Например, для обработки щелчка на кнопке можно использовать следующий код:

```
<input type="button" value="Click Me" onclick="alert('Clicked')" />
```

При щелчке на этой кнопке появится оповещение, заданное как значение атрибута `onclick`. Из-за того что значением атрибута является JavaScript-код, в нем нужно экранировать знаки, относящиеся к синтаксису HTML, такие как амперсанд, двойные кавычки, меньше и больше. Именно по этой причине в примере используются одинарные кавычки вместо двойных. Для применения двойных кавычек нужно было бы изменить код:

```
<input type="button" value="Click Me"
       onclick="alert(&quot;Clicked&quot;)" />
```

Обработчик события, определенный в HTML-коде, может также вызывать сценарий, находящийся в другом месте:

Листинг HTMLEventHandlerExample01.htm


```
<script type="text/javascript">
    function showMessage(){
        alert("Hello world!");
    }
</script>
<input type="button" value="Click Me" onclick="showMessage()" />
```

Здесь при щелчке на кнопке вызывается функция `showMessage()`, которая определена в отдельном элементе `<script>` и может даже находиться во внешнем файле. Обработчикам событий доступен весь код в глобальной области видимости.

Обработчики событий, назначенные таким способом, имеют уникальные особенности. Так, для каждого из них создается функция, в которую заключается значение атрибута. У этой функции есть специальная локальная переменная `event`, или объект события (см. далее):

```
<!-- выводит "click" -->
<input type="button" value="Click Me" onclick="alert(event.type)">
```

Это обеспечивает доступ к объекту события, благодаря чему можно не определять его самостоятельно и не извлекать из списка аргументов функции-контейнера.

Значение `this` внутри этой функции указывает на целевой элемент события, например:

```
<!-- выводит "Click Me" -->
<input type="button" value="Click Me" onclick="alert(this.value)">
```

Эта динамически создаваемая функция интересна еще и тем, как она наращивает цепочку областей видимости. В ней доступны члены объекта `document` и самого элемента, как если бы они были ее локальными переменными. Это возможно благодаря приращению цепочки областей видимости с помощью инструкции `with`:

```
function(){
    with(document){
        with(this){
            // значение атрибута
        }
    }
}
```

Это означает, что в обработчике события можно без проблем использовать его собственные свойства. Так, следующий код эквивалентен предыдущему:

```
<!-- выводит "Click Me" -->
<input type="button" value="Click Me" onclick="alert(value)">
```

Если элементом является элемент ввода формы, цепочка областей видимости также содержит запись родительского элемента формы:

```
function(){
    with(document){
        with(this.form){
            with(this){
                // значение атрибута
            }
        }
    }
}
```

По сути, это позволяет обработчику события обращаться к другим членам той же формы, не ссылаясь на сам элемент формы, например:

Листинг HTMLEventHandlerExample04.htm


```
<form method="post">
    <input type="text" name="username" value="">
    <input type="button" value="Echo Username"
           onclick="alert(username.value)">
</form>
```

При щелчке на кнопке в этом примере выводится текст из текстового поля. Обратите внимание, что мы ссылаемся на `username` непосредственно.

Назначение обработчиков событий в HTML-коде имеет несколько недостатков. Первый — проблема времени: возможна ситуация, когда HTML-элемент появится на странице и пользователь начнет взаимодействовать с ним прежде, чем будет готов код обработчика события. Представьте, что в предыдущем примере функция `showMessage()` определена позже кнопки. Если пользователь щелкнет на кнопке до загрузки кода `showMessage()`, произойдет ошибка. По этой причине большинство HTML-обработчиков событий заключают в блоки `try-catch`, например:

```
<input type="button" value="Click Me"
       onclick="try{showMessage();}catch(ex){}">
```

Если щелкнуть на этой кнопке до определения функции `showMessage()`, JavaScript-ошибка останется незамеченной, потому что будет перехвачена прежде, чем достигнет браузера.

Другим недостатком является то, что приращение цепочки областей видимости в функции обработчика событий может давать разные результаты в разных браузерах. Из-за того что правила разрешения идентификаторов в интерпретаторах JavaScript немного различаются, доступ к неквалифицированным членам объектов может приводить к ошибкам.

Наконец, назначение обработчиков событий в HTML-коде усиливает связь HTML- и JavaScript-кода. Если такой обработчик события нужно будет изменить, это может потребовать внесения изменений в двух местах вместо одного. Это главная причина того, что многие разработчики предпочитают реализовывать обработчики событий в JavaScript-коде.

Дополнительные сведения о недостатках HTML-обработчиков событий см. в статье «Event Handler Scope» («Область видимости обработчиков событий») Гаррета Смита (Garrett Smith) по адресу www.jibbering.com/faq/names/event_handler.html.

Обработчики событий DOM Level 0

Традиционный способ обработки событий в JavaScript включает назначение функции свойству обработчика события. Этот способ был представлен еще в браузерах четвертого поколения и до сих пор используется во всех современных браузерах благодаря простоте и широкой поддержке. Чтобы назначить обработчик события в JavaScript, нужно сначала получить ссылку на целевой объект.

У каждого элемента (а также у объектов `window` и `document`) есть свойства обработчиков событий, которые обычно имеют имена в нижнем регистре, например `onclick`. Для обработки события нужно назначить функцию-обработчик такому свойству:

```
var btn = document.getElementById("myBtn");
btn.onclick = function(){
    alert("Clicked");
};
```

Этот код назначает кнопке обработчик события `click`. Имейте в виду, что обработчик события назначается только при выполнении этого кода, и если он располагается на странице после кода кнопки, какое-то время щелчки на кнопке могут срабатывать вхолостую.

Обработчик события, назначенный таким способом, считается методом элемента и выполняется в его области видимости, то есть переменная `this` указывает на элемент:

Листинг DOMLevel0EventHandlerExample01.htm


```
var btn = document.getElementById("myBtn");
btn.onclick = function(){
    alert(this.id);        // "myBtn"
};
```

Этот код при нажатии на кнопку выводит ее идентификатор, для получения которого используется свойство `this.id`. Через переменную `this` в обработчике события доступны любые свойства и методы его элемента. Обработчики событий, добавленные таким способом, используются на этапе всплытия событий.

Чтобы удалить обработчик события, назначенный в стиле DOM Level 0, достаточно присвоить его свойству значение `null`, например:

```
btn.onclick = null;        // удаление обработчика события
```

После удаления обработчика при щелчке на кнопке больше ничего происходить не будет.

Если обработчик события назначается в HTML-коде, свойству обработчика присваивается функция, содержащая код HTML-атрибута. Такой обработчик события также можно удалить, присвоив его свойству значение null.

Обработчики событий DOM Level 2

В DOM Level 2 Events для назначения и удаления обработчиков событий используются методы `addEventListener()` и `removeEventListener()`, которые есть у всех DOM-узлов. Каждый из них принимает три аргумента: имя обрабатываемого события, функцию-обработчик и логическое значение, указывающее, нужно ли вызывать обработчик события на этапе перехвата (`true`) или всплытия (`false`).

Например, добавить обработчик щелчка на кнопке можно следующим образом:

```
var btn = document.getElementById("myBtn");
btn.addEventListener("click", function(){
    alert(this.id);
}, false);
```

Этот код назначает кнопке обработчик события `click`, который будет вызываться на этапе всплытия (поскольку последний аргумент имеет значение `false`). Как и при подходе DOM Level 0, обработчик события выполняется в области видимости элемента, к которому он подключен. Основное преимущество подхода DOM Level 2 в том, что событию можно назначить несколько обработчиков. Рассмотрим пример:

Листинг DOMLevel2EventHandlerExample01.htm


```
var btn = document.getElementById("myBtn");
btn.addEventListener("click", function(){
    alert(this.id);
}, false);
btn.addEventListener("click", function(){
    alert("Hello world!");
}, false);
```

Этот код определяет два обработчика щелчка на кнопке, которые будут срабатывать в порядке добавления, так что первое оповещение будет содержать идентификатор элемента, а второе — сообщение `"Hello world!"`.

Обработчик события, добавленный с помощью метода `addEventListener()`, можно удалить, только вызвав метод `removeEventListener()` с теми же аргументами. Это означает, что анонимные функции, добавляемые методом `addEventListener()`, удалить нельзя:

```
var btn = document.getElementById("myBtn");
btn.addEventListener("click", function(){
    alert(this.id);
}, false);

// другой код

btn.removeEventListener("click", function(){    // не сработает!
    alert(this.id);
}, false);
```

Здесь метод `addEventListener()` добавляет в качестве обработчика события анонимную функцию. На первый взгляд, в вызове `removeEventListener()` используются такие же аргументы, но на самом деле вторым аргументом является совершенно другая функция. Чтобы обработчик события можно было удалить методом `removeEventListener()`, его функция должна быть той же, что была передана в метод `addEventListener()`, например:

Листинг DOMLevel2EventHandlerExample02.htm


```
var btn = document.getElementById("myBtn");
var handler = function(){
    alert(this.id);
};
btn.addEventListener("click", handler, false);

// другой код

btn.removeEventListener("click", handler, false);    // все в порядке!
```

Этот код работает без сюрпризов, потому что в этот раз аргументы методов `addEventListener()` и `removeEventListener()` действительно одинаковы.

В большинстве случаев события обрабатывают на этапе всплытия, потому что этот способ поддерживается шире. Обрабатывать событие на этапе перехвата имеет смысл, только если нужно перехватить его прежде, чем оно достигнет целевого элемента. Если это не требуется, лучше не перехватывать события.

Обработчики событий DOM Level 2 поддерживаются в Internet Explorer 9, Firefox, Safari, Chrome и Opera.

Обработчики событий в Internet Explorer

В Internet Explorer реализованы аналогичные DOM Level 2 методы `attachEvent()` и `detachEvent()`. Они принимают в качестве аргументов имя обработчика события и его функцию. Поскольку в Internet Explorer 8 и более ранних версий поддерживается только всплытие событий, обработчики событий, добавленные методом `attachEvent()`, выполняются на этапе всплытия.

Добавить обработчик щелчка на кнопке можно следующим образом:

Листинг IEEventHandlerExample01.htm


```
var btn = document.getElementById("myBtn");
btn.attachEvent("onclick", function(){
    alert("Clicked");
});
```

Обратите внимание, что первым аргументом метода `attachEvent()` является строка `"onclick"`, а не `"click"`, как в DOM-методе `addEventListener()`.

Этот способ и подход DOM Level 0 в Internet Explorer различаются областью видимости обработчика события. При использовании DOM Level 0 значение `this` в обработчике события указывает на элемент, к которому он подключен, тогда как обработчик, назначенный методом `attachEvent()`, выполняется в глобальном контексте и его переменная `this` указывает на `window`, например:

```
var btn = document.getElementById("myBtn");
btn.attachEvent("onclick", function(){
    alert(this === window);        // true
});
```

Это различие важно иметь в виду при написании кроссбраузерного кода.

Подобно методу `addEventListener()`, метод `attachEvent()` позволяет назначить несколько обработчиков одному событию:

Листинг IEEventHandlerExample01.htm


```
var btn = document.getElementById("myBtn");
btn.attachEvent("onclick", function(){
    alert("Clicked");
});
btn.attachEvent("onclick", function(){
    alert("Hello world!");
});
```

Здесь мы дважды вызываем метод `attachEvent()`, добавляя два обработчика щелчка на кнопке. Однако в отличие от DOM-метода, эти обработчики вызываются в обратном порядке. При щелчке на кнопке первым появится оповещение со строкой `"Hello world!"`, а вторым — со строкой `"Clicked"`.

Обработчики, добавленные методом `attachEvent()`, можно удалить, вызвав метод `detachEvent()` с такими же аргументами. Как и в DOM, это означает, что назначенные событиям анонимные функции удалить невозможно — метод `detachEvent()` должен получить ссылку на ту же функцию:

Листинг IEEventHandlerExample02.htm

```
var btn = document.getElementById("myBtn");
var handler = function(){
    alert("Clicked");
};
btn.attachEvent("onclick", handler);

// другой код

btn.detachEvent("onclick", handler);
```

Этот код назначает событию обработчик, сохраненный в переменной `handler`, а затем удаляет его методом `detachEvent()`.

Обработчики событий Internet Explorer поддерживаются и в Opera.

Кроссбраузерные обработчики событий

Для кроссбраузерной обработки событий многие разработчики либо задействуют JavaScript-библиотеку, которая абстрагирует различия браузеров, либо пишут собственный код, основанный на распознавании возможностей (см. главу 9). Чтобы такой код можно было применять максимально широко, он должен выполняться только на этапе всплытия событий.

Первым делом следует создать метод `addHandler()`, добавляющий обработчик события, используя подход DOM Level 0, DOM Level 2 или Internet Explorer в зависимости от того, какой из них доступен. Мы создадим обработчик в объекте `EventUtil`, который во всей книге будет применяться для сглаживания различий между браузерами. Метод `addHandler()` принимает три аргумента: целевой элемент, имя события и функцию-обработчик.

Нам также потребуется метод `removeHandler()`, который принимает те же три аргумента и удаляет ранее добавленный обработчик события, используя один из специфичных подходов или традиционный подход DOM Level 0, если никакой другой способ недоступен.

Вот полный код объекта `EventUtil`:

Листинг EventUtil.js


```
var EventUtil = {

    addHandler: function(element, type, handler){
        if (element.addEventListener){
            element.addEventListener(type, handler, false);
        } else if (element.attachEvent){
            element.attachEvent("on" + type, handler);
        } else {
            element["on" + type] = handler;
        }
    },

    removeHandler: function(element, type, handler){
        if (element.removeEventListener){
            element.removeEventListener(type, handler, false);
        } else if (element.detachEvent){
            element.detachEvent("on" + type, handler);
        } else {
            element["on" + type] = null;
        }
    }

};
```

Оба метода сначала проверяют, доступен ли для полученного элемента метод DOM Level 2. Если да, мы вызываем его, передавая в качестве аргументов тип события, обработчик и значение `false`, которое указывает, что метод должен быть выполнен на этапе всплытия. Во второй ветви кода мы пытаемся воспользоваться методом Internet Explorer. Заметьте, что перед типом события требуется префикс `"on"`, чтобы код работал в Internet Explorer 8 и более ранних версий. В третьей ветви вызывается метод DOM Level 0 (в современных браузерах эта ветвь выполняться не должна). Обратите внимание, что мы назначаем свойству обработчик события или значение `null` с помощью скобочной нотации.

Этот вспомогательный объект применяется следующим образом:

Листинг CrossBrowserEventHandlerExample01.htm


```
var btn = document.getElementById("myBtn");
var handler = function(){
    alert("Clicked");
};
EventUtil.addHandler(btn, "click", handler);

// другой код

EventUtil.removeHandler(btn, "click", handler);
```

Конечно, методы `addHandler()` и `removeHandler()` не в состояния сгладить различия в функционале всех браузеров (например, проблема с областью видимости в Internet Explorer остается), но все же они упрощают добавление и удаление обработчиков событий. Помните также, что DOM Level 0 поддерживает только один обработчик для каждого события. К счастью, браузеры DOM Level 0 уже почти вышли из употребления, так что это не должно быть проблемой.

Объект event

Когда генерируется DOM-событие, все релевантные данные сохраняются в объекте `event`. Они включают базовые сведения, такие как целевой элемент и тип события, а также любые другие данные о конкретном событии. Например, для события мыши сохраняются сведения о позиции мыши, а для события клавиатуры — сведения о нажатых клавишах. Объект `event` поддерживают все браузеры, но по-разному.

Объект event в DOM

В браузерах, соответствующих требованиям DOM, объект `event` является единственным аргументом обработчика события независимо от того, был ли тот назначен в стиле DOM Level 0 или DOM Level 2, например:

```
var btn = document.getElementById("myBtn");
btn.onclick = function(event){
    alert(event.type);    // "click"
};

btn.addEventListener("click", function(event){
   alert(event.type);     // "click"
}, false);
```

Оба обработчика в этом примере выводят в оповещении значение свойства `event.type`, которое всегда содержит тип произошедшего события, например `"click"` (это то же значение, которое передается в методы `addEventListener()` и `removeEventListener()`).

При назначении обработчика события с помощью HTML-атрибута объект `event` доступен как переменная `event`:

```
<input type="button" value="Click Me" onclick="alert(event.type)">
```

Благодаря этому HTML-обработчики событий можно использовать аналогично JavaScript-функциям.

Объект `event` содержит свойства и методы, связанные с конкретным событием, приведшим к его созданию. Доступные свойства и методы различаются в зависимости от типа события, но члены из приведенной таблицы есть у всех событий.

Свойство/метод	Тип	Чтение/запись	Описание
bubbles	Boolean	Только чтение	Указывает, всплывает ли событие
cancelable	Boolean	Только чтение	Указывает, можно ли отменить поведение события, предлагаемое по умолчанию
currentTarget	Element	Только чтение	Элемент, чей обработчик обрабатывает событие в текущий момент
defaultPrevented	Boolean	Только чтение	Значение true указывает, что был вызван метод preventDefault() (свойство было добавлено в DOM Level 3 Events)
detail	Integer	Только чтение	Дополнительные сведения о событии
eventPhase	Integer	Только чтение	Этап, на котором вызывается обработчик события (1 — перехват; 2 — обработка в целевом элементе; 3 — всплытие)
preventDefault()	Function	Только чтение	Отменяет поведение по умолчанию. Этот метод можно использовать, если свойство cancelable имеет значение true
stopImmediate-Propagation()	Function	Только чтение	Останавливает перехват или всплытие события и предотвращает вызов других обработчиков события (метод был добавлен в DOM Level 3 Events)
stopPropagation()	Function	Только чтение	Отменяет перехват или всплытие события. Этот метод можно использовать, если свойство bubbles имеет значение true
target	Element	Только чтение	Целевой элемент события
trusted	Boolean	Только чтение	Значение true, указывает, что событие было сгенерировано браузером. Значение false указывает, что событие было создано разработчиком с помощью JavaScript (свойство было добавлено в DOM Level 3 Events)

Свойство/метод	Тип	Чтение/запись	Описание
type	String	Только чтение	Тип события
view	AbstractView	Только чтение	Абстрактное представление, связанное с событием (объект window, в котором произошло событие)

Внутри обработчика события объект `this` всегда имеет значение `currentTarget`, тогда как свойство `target` содержит фактический целевой элемент события. Если обработчик события назначен самому целевому элементу, значения `this`, `currentTarget` и `target` совпадают, например:

Листинг DOMEventObjectExample01.htm


```
var btn = document.getElementById("myBtn");
btn.onclick = function(event){
    alert(event.currentTarget === this);     // true
    alert(event.target === this);            // true
};
```

Этот код сравнивает свойства `currentTarget` и `target` со значением `this`. Поскольку событие `click` было сгенерировано кнопкой, все они равны. В обработчике события, назначенном родительскому узлу кнопки, такому как `document.body`, значения были бы разными. Рассмотрим пример:

Листинг DOMEventObjectExample02.htm

```
document.body.onclick = function(event){
    alert(event.currentTarget === document.body);                // true
    alert(this === document.body);                               // true
    alert(event.target === document.getElementById("myBtn"));    // true
};
```

Здесь при щелчке на кнопке свойства `this` и `currentTarget` равны `document.body`, потому что именно этому узлу назначен обработчик события. Однако генерирует событие кнопка, поэтому свойство `target` указывает на нее. Поскольку самой кнопке обработчик события не назначен, событие всплывает к узлу `document.body`, где и обрабатывается.

Свойство `type` полезно, если нужно обработать несколько событий с помощью одной функции, например:

Листинг DOMEventObjectExample03.htm


```
var btn = document.getElementById("myBtn");
var handler = function(event){
    switch(event.type){
        case "click":
            alert("Clicked");
```

```
            break;

        case "mouseover":
            event.target.style.backgroundColor = "red";
            break;

        case "mouseout":
            event.target.style.backgroundColor = "";
            break;
    }
};

btn.onclick = handler;
btn.onmouseover = handler;
btn.onmouseout = handler;
```

В этом примере функция `handler` обрабатывает три события: `click`, `mouseover` и `mouseout`. При щелчке на кнопке функция выводит оповещение, при наведении указателя мыши на кнопку изменяет ее цвет на красный, а при смещении указателя с кнопки возвращает ей цвет, предлагаемый по умолчанию. Используя свойство `event.type`, функция определяет, какое событие произошло, и реагирует надлежащим образом.

Метод `preventDefault()` отменяет для конкретного события выполнение действия, предлагаемого по умолчанию. Например, при щелчке на ссылке таким действием является переход по URL-адресу, указанному в ее атрибуте `href`. При желании это действие можно отменить в обработчике события следующим образом:

Листинг DOMEventObjectExample04.htm

```
var link = document.getElementById("myLink");
link.onclick = function(event){
    event.preventDefault();
};
```

У любого события, которое можно отменить методом `preventDefault()`, свойство `cancelable` имеет значение `true`.

Метод `stopPropagation()` немедленно останавливает распространение события по DOM-структуре, блокируя последующий перехват или всплытие события. Например, в обработчике, назначенном кнопке, с помощью метода `stopPropagation()` можно предотвратить выполнение обработчика, назначенного узлу `document.body`:

Листинг DOMEventObjectExample05.htm


```
var btn = document.getElementById("myBtn");
btn.onclick = function(event){
    alert("Clicked");
    event.stopPropagation();
};

document.body.onclick = function(event){
    alert("Body clicked");
};
```

Без вызова метода `stopPropagation()` этот код при щелчке на кнопке вывел бы на экран два оповещения, но в приведенной версии событие `click` не достигает узла `document.body` и его обработчик `onclick` не выполняется.

Свойство `eventPhase` позволяет определить текущий этап распространения события. При вызове обработчика события на этапе перехвата значение `eventPhase` равно 1, при обработке события в целевом элементе — 2, а во время всплытия — 3. Имейте в виду, что хотя обработка события в целевом элементе относится к этапу всплытия, значение `eventPhase` в этом случае всегда равно 2. Рассмотрим пример:

Листинг DOMEventObjectExample06.htm


```
var btn = document.getElementById("myBtn");
btn.onclick = function(event){
    alert(event.eventPhase);    // 2
};

document.body.addEventListener("click", function(event){
    alert(event.eventPhase);    // 1
}, true);

document.body.onclick = function(event){
    alert(event.eventPhase);    // 3
};
```

В этом примере при щелчке на кнопке первым вызывается один из обработчиков, назначенных узлу `document.body`. Он срабатывает на этапе перехвата и отображает оповещение со значением 1. Вторым вызывается обработчик самой кнопки, в котором свойство `eventPhase` имеет значение 2. Наконец, на этапе всплытия вызывается другой обработчик, назначенный узлу `document.body`, при этом значение `eventPhase` равно 3. Если это свойство равно 2, значения `this`, `target` и `currentTarget` всегда одинаковы.

Объект event существует только во время выполнения обработчиков событий, а после выполнения всех обработчиков он уничтожается.

Объект event в Internet Explorer

В отличие от DOM, в Internet Explorer способ доступа к объекту `event` зависит от того, как был назначен обработчик события. Если использовался подход DOM Level 0, объект `event` доступен только как свойство объекта `window`, например:

```
var btn = document.getElementById("myBtn");
btn.onclick = function(){
    var event = window.event;
    alert(event.type);        // "click"
};
```

Здесь объект `event` используется для определения типа произошедшего события (свойство `type` в Internet Explorer идентично одноименному DOM-свойству). Если

же обработчик события был назначен с помощью метода `attachEvent()`, объект `event` передается в обработчик как его единственный аргумент:

```
var btn = document.getElementById("myBtn");
btn.attachEvent("onclick", function(event){
    alert(event.type);        // "click"
});
```

При использовании метода `attachEvent()` объект `event` также доступен в объекте `window`, как и при подходе DOM Level 0. В обработчик события он передается ради удобства.

Если обработчик события назначается с помощью HTML-атрибута, объект `event` доступен как переменная `event` (как и в модели DOM), например:

```
<input type="button" value="Click Me" onclick="alert(event.type)">
```

В Internet Explorer объект `event` также содержит свойства и методы, связанные с событием, приведшим к его созданию. Многие из них соответствуют свойствам и методам DOM или связаны с ними. Как и в DOM, свойства и методы объекта `event` зависят от типа произошедшего события, при этом члены, указанные в таблице, доступны для всех событий.

Свойство/метод	Тип	Чтение/запись	Описание
cancelBubble	Boolean	Чтение и запись	По умолчанию это свойство имеет значение false, но ему можно присвоить значение true, чтобы отменить всплытие события (это аналог DOM-метода stopPropagation())
returnValue	Boolean	Чтение и запись	По умолчанию это свойство имеет значение true, но ему можно присвоить значение false, чтобы отменить поведение события, предлагаемое по умолчанию (это аналог DOM-метода preventDefault())
srcElement	Element	Только чтение	Целевой элемент события (аналог DOM-свойства target)
type	String	Только чтение	Тип события

Поскольку область видимости обработчика события зависит от того, как он был назначен, объект `this` не всегда указывает на целевой элемент события. Поэтому используйте вместо этого свойство `event.srcElement`, например:

Листинг IEEventObjectExample01.htm


```
var btn = document.getElementById("myBtn");
btn.onclick = function(){
    alert(window.event.srcElement === this);     // true
};
```

```
btn.attachEvent("onclick", function(event){
    alert(event.srcElement === this);           // false
});
```

В первом обработчике события, назначенном с использованием подхода DOM Level 0, свойство `srcElement` равно `this`, но во втором эти значения различаются.

Свойство `returnValue` аналогично DOM-методу `preventDefault()` — оно тоже отменяет поведение, предлагаемое по умолчанию конкретного события. Чтобы предотвратить действие, предлагаемое по умолчанию, нужно лишь присвоить ему значение `false`, например:

Листинг IEEventObjectExample02.htm

```
var link = document.getElementById("myLink");
link.onclick = function(){
    window.event.returnValue = false;
};
```

В этом примере свойство `returnValue` отменяет действие для щелчка не ссылке, предлагаемое по умолчанию. В отличие от DOM, при этом никак нельзя узнать, возможна ли отмена события.

Свойство `cancelBubble` делает то же самое, что и DOM-метод `stopPropagation()`, — останавливает всплытие события. Поскольку в Internet Explorer 8 и более ранних версий этап перехвата не поддерживается, оно отменяет только всплытие события, тогда как метод `stopPropagation()` отменяет и перехват, и всплытие. Вот пример:

Листинг IEEventObjectExample03.htm

```
var btn = document.getElementById("myBtn");
btn.onclick = function(){
    alert("Clicked");
    window.event.cancelBubble = true;
};

document.body.onclick = function(){
    alert("Body clicked ");
};
```

Присвоение значения `true` свойству `cancelBubble` в обработчике щелчка на кнопке предотвращает всплытие события до обработчика, назначенного узлу `document.body`. В результате при щелчке на кнопке отображается только одно оповещение.

Кроссбраузерный объект event

Хотя объекты `event` в DOM и Internet Explorer различаются, они достаточно похожи, чтобы можно было создавать кроссбраузерные решения для обработки событий. Все данные и возможности IE-объекта `event` доступны в несколько иной форме в DOM-объекте, что позволяет легко провести параллели между двумя моделями

событий. Созданный ранее объект `EventUtil` можно расширить методами, компенсирующими их различия:

Листинг EventUtil.js


```
var EventUtil = {

    addHandler: function(element, type, handler){
        // код опущен с целью сокращения объема листинга
    },

    getEvent: function(event){
        return event ? event : window.event;
    },

    getTarget: function(event){
        return event.target || event.srcElement;
    },

    preventDefault: function(event){
        if (event.preventDefault){
            event.preventDefault();
        } else {
            event.returnValue = false;
        }
    },

    removeHandler: function(element, type, handler){
        // код опущен с целью сокращения объема листинга
    },

    stopPropagation: function(event){
        if (event.stopPropagation){
            event.stopPropagation();
        } else {
            event.cancelBubble = true;
        }
    }

};
```

Этот код добавляет к объекту `EventUtil` четыре новых метода. Первый, `getEvent()`, возвращает ссылку на объект `event` независимо от того, как был назначен обработчик события (напомним, что в Internet Explorer от этого зависит способ доступа к событию). В качестве аргумента этот метод принимает объект `event`, переданный в обработчик события:

Листинг CrossBrowserEventObjectExample01.htm


```
btn.onclick = function(event){
    event = EventUtil.getEvent(event);
};
```

Если браузер соответствует требованиям DOM, метод `getEvent()` просто возвращает полученную переменную `event`. В Internet Explorer аргумент `event` не определен,

поэтому метод возвратит свойство `window.event`. Добавление этой строки в начало обработчиков событий гарантирует, что объект `event` будет доступен всегда, независимо от браузера.

Метод `getTarget()` возвращает целевой элемент события. В нем мы проверяем, есть ли у объекта `event` свойство `target`, и если да, возвращаем его, иначе вместо него используется свойство `srcElement`. Этот метод применяется следующим образом:

Листинг CrossBrowserEventObjectExample01.htm

```
btn.onclick = function(event){
    event = EventUtil.getEvent(event);
    var target = EventUtil.getTarget(event);
};
```

Метод `preventDefault()` отменяет для события поведение, предлагаемое по умолчанию. Получив объект `event`, он проверяет, есть ли у события метод `preventDefault()`, и если да, вызывает его, в противном случае присваивает свойству `returnValue` значение `false`. Вот пример его вызова:

Листинг CrossBrowserEventObjectExample02.htm

```
var link = document.getElementById("myLink");
link.onclick = function(event){
    event = EventUtil.getEvent(event);
    EventUtil.preventDefault(event);
};
```

Этот код предотвращает переход на другую страницу при щелчке на ссылке. Здесь мы вызываем метод `EventUtil.getEvent()` для получения объекта `event`, который затем передается в метод `EventUtil.preventDefault()` для отмены действия, предлагаемого по умолчанию.

Последний метод, `stopPropagation()`, работает похожим образом. Сначала он пытается остановить распространение события с помощью DOM-метода, а если тот недоступен, использует свойство `cancelBubble`. Применяется он следующим образом:

Листинг CrossBrowserEventObjectExample03.htm


```
var btn = document.getElementById("myBtn");
btn.onclick = function(event){
    alert("Clicked");
    event = EventUtil.getEvent(event);
    EventUtil.stopPropagation(event);
};

document.body.onclick = function(event){
    alert("Body clicked ");
};
```

В этом фрагменте мы получаем объект `event` с помощью метода `EventUtil.getEvent()`, а затем передаем его в метод `EventUtil.stopPropagation()`. Помните, что в зависимости от браузера он останавливает и перехват, и всплытие события либо только всплытие.

Типы событий

В спецификации DOM Level 3 Events определены категории событий, указанные в приведенном списке. Как уже отмечалось, от типа события зависят доступные сведения о нем.

- **События пользовательского интерфейса** — это общие события браузера, которые иногда связаны с BOM.
- **События изменения фокуса** генерируются, когда элемент получает или теряет фокус.
- **События мыши** генерируются при выполнении каких-либо действий на странице с помощью мыши.
- **События колесика** генерируются при использовании колесика мыши (или аналогичного устройства).
- **События редактирования текста** генерируются при вводе текста в документ.
- **События клавиатуры** генерируются при выполнении каких-либо действий на странице с помощью клавиатуры.
- **События композиции** генерируются при вводе символов в редакторе метода ввода (Input Method Editor, IME).
- **События изменения DOM-структуры** генерируются при изменении базовой DOM-структуры.
- **События изменения имен** генерируются при изменении имен элементов или атрибутов. Эти события устарели и реализованы не во всех браузерах, поэтому мы их рассматривать не будем.

В дополнение к этим категориям доступны HTML5-события, а также фирменные DOM- и BOM-события. Фирменные события обычно определяют исходя из требований разработчиков, а не спецификаций, и их реализации могут зависеть от браузера.

DOM Level 3 Events переопределяет группы событий DOM Level 2 Events и содержит дополнительные события. Спецификацию DOM Level 2 Events поддерживают все основные браузеры, в том числе Internet Explorer 9, который также поддерживает DOM Level 3 Events.

События пользовательского интерфейса

События пользовательского интерфейса (User Interface, UI) не всегда связаны с действиями пользователя. Они существовали в той или иной форме еще до спецификации DOM и были оставлены для обеспечения обратной совместимости.

- `DOMActivate` — генерируется при активации элемента с помощью мыши или клавиатуры (оно более общее, чем события `click` и `keydown`). Это событие, поддерживаемое в Firefox 2+ и Chrome, объявлено устаревшим в DOM Level 3 Events. Из-за различий реализации рекомендуется не использовать его.
- `load` — генерируется для объекта `window` при завершении загрузки страницы, для набора фреймов при завершении загрузки всех фреймов, а также для элементов `<img>` и `<object>` при завершении их загрузки.
- `unload` — генерируется для объекта `window` при завершении выгрузки страницы, для набора фреймов при завершении выгрузки всех фреймов и для элемента `<object>` при завершении его выгрузки.
- `abort` — генерируется для элемента `<object>`, если пользователь останавливает загрузку, а элемент загружен не полностью.
- `error` — генерируется для объекта `window`, если возникает JavaScript-ошибка; для элемента `<img>`, если невозможно загрузить указанное изображение; для элемента `<object>`, если невозможно загрузить его, или для набора фреймов, если невозможно загрузить один или несколько фреймов. Это событие обсуждается в главе 17.
- `select` — генерируется, когда пользователь выделяет один или несколько символов в текстовом поле (`<input>` или `<textarea>`). Это событие обсуждается в главе 14.
- `resize` — генерируется для объекта `window` или фрейма при изменении его размеров.
- `scroll` — генерируется для любого элемента с полосой прокрутки, когда пользователь его прокручивает. Полоса прокрутки загруженной страницы принадлежит элементу `<body>`.

Большинство HTML-событий связаны с объектом `window` или с элементами управления форм.

За исключением `DOMActivate`, в спецификации DOM Level 2 Events эти события входили в группу HTML Events (событие `DOMActivate` входило в группу UI Events). Узнать, поддерживает ли браузер HTML-события, как описано в спецификации DOM Level 2 Events, можно следующим образом:

```
var isSupported = document.implementation.hasFeature("HTMLEvents", "2.0");
```

Имейте в виду, что этот вызов должен возвращать `true`, только если HTML-события реализованы в браузере согласно спецификации DOM Level 2 Events. Браузеры могут поддерживать эти события нестандартным образом и возвращать `false`. Чтобы определить, поддерживает ли браузер эти события согласно спецификации DOM Level 3 Events, используйте следующий код:

```
var isSupported = document.implementation.hasFeature("UIEvent", "3.0");
```

Событие load

Наверное, событие `load` используется в JavaScript чаще любых других событий. Для объекта `window` оно возникает, когда загружена вся страница, включая все внешние ресурсы, такие как изображения, JavaScript- и CSS-файлы. Определить обработчик `onload` можно двумя способами. Первый — использовать JavaScript:

Листинг LoadEventExample01.htm


```
EventUtil.addHandler(window, "load", function(event){
    alert("Loaded!");
});
```

Этот код основан на использовании кроссбраузерного объекта `EventUtil`, который был рассмотрен ранее. Как и при других событиях, в обработчик передается объект `event`. Он не предоставляет никаких дополнительных сведений об этом событии, хотя интересно отметить, что в браузерах, соответствующих DOM, свойство `event.target` указывает на объект `document`, тогда как Internet Explorer до версии 8 не задает свойство `srcElement` для этого события.

Второй способ назначить обработчик события `load` — это добавить атрибут `onload` в элемент `<body>`, например:

Листинг LoadEventExample02.htm

```
<!DOCTYPE html>
<html>
<head>
    <title>Load Event Example</title>
</head>
<body onload="alert('Loaded!')">

</body>
</html>
```

Вообще говоря, обработчики любых событий объекта `window` назначаются в HTML с помощью атрибутов элемента `<body>`, потому что в HTML-коде элемент `window` недоступен. Этот трюк применяется для обеспечения обратной совместимости и все еще хорошо поддерживается во всех браузерах, но по возможности рекомендуется использовать подход JavaScript.

Согласно спецификации DOM Level 2 Events, событие load должно генерироваться для объекта document, а не window, однако из соображений обратной совместимости во всех браузерах оно генерируется для объекта window.

Событие `load` также генерируется для изображений, причем как соответствующих, так и не соответствующих модели DOM. В HTML-коде можно назначить обработчик события `load` для любых изображений в документе следующим образом:

Листинг LoadEventExample03.htm

```
<img src="smile.gif" onload="alert('Image loaded.')">
```

По окончании загрузки указанного изображения этот код выводит оповещение "Image loaded.". То же самое можно сделать с помощью JavaScript:

Листинг LoadEventExample04.htm


```
var image = document.getElementById("myImage");
EventUtil.addHandler(image, "load", function(event){
    event = EventUtil.getEvent(event);
    alert(EventUtil.getTarget(event).src);
});
```

Этот обработчик события load принимает объект event, но интересного в нем мало. Целевым элементом события является элемент <img>, так что мы можем отобразить его свойство src.

При создании элемента <img> можно назначить ему обработчик события, вызываемый при завершении загрузки изображения. В этом случае важно назначить обработчик до установки свойства src, например:

Листинг LoadEventExample05.htm

```
EventUtil.addHandler(window, "load", function(){
    var image = document.createElement("img");
    EventUtil.addHandler(image, "load", function(event){
        event = EventUtil.getEvent(event);
        alert(EventUtil.getTarget(event).src);
    });
    document.body.appendChild(image);
    image.src = "smile.gif";
});
```

В первой части примера мы назначаем обработчик события загрузки окна. Это важно потому, что мы собираемся добавить новый DOM-элемент, а попытка использовать элемент document.body до полной загрузки страницы может привести к ошибке. Далее мы создаем элемент image и устанавливаем обработчик его события load, а затем добавляем изображение на страницу и задаем его атрибут src. Чтобы началась загрузка изображения, элемент не нужно добавлять к документу — оно начинает загружаться, как только задано свойство src.

Эту методику можно также использовать с объектом Image из DOM Level 0, который еще до внедрения DOM применялся на клиентских устройствах для предварительной загрузки изображений. Он работает так же, как элемент <img>, но его нельзя добавить в DOM-дерево. Рассмотрим пример:

Листинг LoadEventExample06.htm

```
EventUtil.addHandler(window, "load", function(){
    var image = new Image();
```

```
    EventUtil.addHandler(image, "load", function(event){
        alert("Image loaded!");
    });
    image.src = "smile.gif";
});
```

Этот код обрабатывает загрузку изображения, созданного с помощью конструктора `Image`. Некоторые браузеры реализуют объект `Image` как элемент `<img>`, но не все, так что лучше считать, что они различаются.

Internet Explorer 8 и более ранних версий не создает объект event, если событие load генерируется для изображения, которое не является частью DOM-документа. Это относится и к элементам <img>, которые не были добавлены к документу, и к объектам Image. В Internet Explorer 9 этот дефект исправлен.

Есть и другие элементы, которые поддерживают событие `load` нестандартным образом. Элемент `<script>` генерирует событие `load` в Internet Explorer 9+, Firefox, Opera, Chrome и Safari 3+, уведомляя о завершении загрузки динамически загружаемого JavaScript-файла. В отличие от изображений, загрузка JavaScript-файлов с сервера начинается только после задания свойства `src` и добавления элемента в документ, так что порядок назначения обработчика события и установки свойства `src` не важен. Следующий пример поясняет назначение обработчика события элементу `<script>`:

Листинг LoadEventExample07.htm


```
EventUtil.addHandler(window, "load", function(){
    var script = document.createElement("script");
    script.type = "text/javascript";
    EventUtil.addHandler(script, "load", function(event){
        alert("Loaded");
    });
    script.src = "example.js";
    document.body.appendChild(script);
});
```

Здесь обработчик события `load` назначается элементу `<script>` с помощью кроссбраузерного объекта `EventUtil`. В большинстве браузеров целевым элементом события будет узел `<script>`. В Internet Explorer 8 и более ранних версий элементы `<script>` не поддерживают событие `load`.

В Internet Explorer и Opera событие `load` генерируется также для элементов `<link>`, что позволяет узнать о завершении загрузки таблицы стилей, например:

Листинг LoadEventExample07.htm

```
EventUtil.addHandler(window, "load", function(){
    var link = document.createElement("link");
    link.type = "text/css";
    link.rel= "stylesheet";
    EventUtil.addHandler(link, "load", function(event){
        alert("css loaded");
    });
```

```
    link.href = "example.css";
    document.getElementsByTagName("head")[0].appendChild(link);
});
```

Подобно сценариям, загрузка таблицы стилей начинается только после задания свойства `href` и добавления элемента `<link>` в документ.

Событие unload

Событие `unload` сигнализирует о завершении выгрузки документа. Оно обычно генерируется при переходе с одной страницы на другую и чаще всего используется для очистки ссылок с целью предотвращения утечки памяти. Подобно событию `load`, обработчик события `unload` можно назначить двумя способами. Первый — с помощью JavaScript:

```
EventUtil.addHandler(window, "unload", function(event){
    alert("Unloaded!");
});
```

В браузерах, соответствующих DOM, объект `event` этого события содержит только свойство `target` (со значением `document`). Internet Explorer 8 и более ранних версий не предоставляет для этого события свойство `srcElement`.

Второй способ задать обработчик `onunload` — добавить атрибут в элемент `<body>`:

Листинг UnloadEventExample01.htm


```
<!DOCTYPE html>
<html>
<head>
    <title>Пример события unload</title>
</head>
<body onunload="alert('Unloaded!')">

</body>
</html>
```

Какой бы подход вы ни выбрали, будьте осторожны с кодом внутри обработчика события `unload`. Поскольку оно генерируется после выгрузки всего контента, в обработчике доступны не все объекты, которые были доступны при загрузке страницы. Попытка изменить расположение или вид DOM-узла может привести к ошибке.

Согласно спецификации DOM Level 2 Events, событие unload должно генерироваться для элемента <body>, а не для объекта window, однако из соображений обратной совместимости во всех браузерах оно генерируется для объекта window.

Событие resize

Событие `resize` происходит при изменении высоты или ширины окна браузера. Оно генерируется для объекта `window`, так что его обработчик можно назначить

с помощью либо JavaScript-кода, либо атрибута `onresize` элемента `<body>`. Как уже отмечалось, первый подход предпочтительнее:

```
EventUtil.addHandler(window, "resize", function(event){
    alert("Resized");
});
```

Подобно другим событиям объекта `window`, в браузерах, соответствующих модели DOM, для события `resize` создается объект `event`, у которого свойство `target` имеет значение `document`. В Internet Explorer 8 и более ранних версий у него нет полезных свойств.

Способ генерирования события `resize` зависит от браузера. Internet Explorer, Safari, Chrome и Opera генерируют его, когда размеры окна браузера меняются на один пиксель, и продолжают генерировать, пока пользователь не прекратит изменять размеры окна. В Firefox это событие генерируется, когда пользователь прекращает изменять размеры окна браузера. Из-за повторения события не следует выполнять в его обработчике ресурсоемкие операции, потому что это может ощутимо замедлить работу браузера.

Событие resize также возникает при свертывании и развертывании окна браузера.

Событие scroll

Хотя событие `scroll` генерируется для объекта `window`, на самом деле оно сигнализирует об изменениях элемента уровня страницы. В режиме совместимости изменения можно отслеживать с помощью свойств `scrollLeft` и `scrollTop` элемента `<body>`, а в стандартном режиме изменения применяются на уровне элемента `<html>` во всех браузерах, за исключением Safari (где используется элемент `<body>`). Вот пример:

Листинг ScrollEventExample01.htm


```
EventUtil.addHandler(window, "scroll", function(event){
    if (document.compatMode == "CSS1Compat"){
        alert(document.documentElement.scrollTop);
    } else {
        alert(document.body.scrollTop);
    }
});
```

Этот код назначает обработчик события, который выводит позицию прокрутки по вертикали с учетом режима визуализации. Safari до версии 3.1 не поддерживает свойство `document.compatMode`, поэтому в старых версиях этого браузера выполняется вторая ветвь кода.

Подобно событию `resize` событие `scroll` генерируется во время прокрутки многократно, так что обрабатывать его следует как можно проще.

События изменения фокуса

События изменения фокуса генерируются, когда элементы страницы получают или теряют фокус. В сочетании со свойствами `document.hasFocus()` и `document.activeElement` они предоставляют сведения о том, что пользователь делает на странице. К этой категории относится шесть событий.

- `blur` — генерируется, когда элемент теряет фокус. Это событие не всплывает и поддерживается во всех браузерах.
- `DOMFocusIn` — генерируется, когда элемент получает фокус. Это всплывающая версия HTML-события `focus`, которая поддерживается только в Opera. В DOM Level 3 Events событие `DOMFocusIn` объявлено устаревшим, а вместо него предложено использовать событие `focusin`.
- `DOMFocusOut` — генерируется, когда элемент теряет фокус. Это универсальная версия HTML-события `blur`, которая поддерживается только в Opera. В DOM Level 3 Events событие `DOMFocusOut` объявлено устаревшим, а вместо него предложено использовать событие `focusout`.
- `focus` — генерируется, когда элемент получает фокус. Это событие не всплывает и поддерживается во всех браузерах.
- `focusin` — генерируется, когда элемент получает фокус. Это всплывающая версия HTML-события `focus`, которая поддерживается в Internet Explorer 5.5+, Safari 5.1+, Opera 11.5+ и Chrome.
- `focusout` — генерируется, когда элемент теряет фокус. Это универсальная версия HTML-события `blur`, которая поддерживается в Internet Explorer 5.5+, Safari 5.1+, Opera 11.5+ и Chrome.

Основными в этой группе являются события `focus` и `blur`, которые поддерживаются в браузерах с ранних дней JavaScript. Проблема в том, что они не всплывают, поэтому в Internet Explorer были добавлены события `focusin` и `focusout`, а в Opera — события `DOMFocusIn` и `DOMFocusOut`. IE-события были стандартизированы в DOM Level 3 Events.

При перемещении фокуса от одного элемента страницы к другому события генерируются в следующем порядке.

1. Для элемента, теряющего фокус, генерируется событие `focusout`.
2. Для элемента, получающего фокус, генерируется событие `focusin`.
3. Для элемента, теряющего фокус, генерируется событие `blur`.
4. Для элемента, теряющего фокус, генерируется событие `DOMFocusOut`.
5. Для элемента, получающего фокус, генерируется событие `focus`.
6. Для элемента, получающего фокус, генерируется событие `DOMFocusIn`.

Целевым элементом событий `blur`, `DOMFocusOut` и `focusout` является элемент, теряющий фокус, а целевым элементом событий `focus`, `DOMFocusIn` и `focusin` — элемент, получающий фокус.

Определить, поддерживает ли браузер эти события, можно следующим образом:

```
var isSupported = document.implementation.hasFeature("FocusEvent", "3.0");
```

События focus и blur не всплывают, но их можно прослушивать на этапе перехвата. Питер-Пол Кох (Peter-Paul Koch) прекрасно раскрыл эту тему в статье по адресу www.quirksmode.org/blog/archives/2008/04/delegating_the.html.

События мыши и колесика мыши

События мыши (mouse events) используются в веб-программировании чаще любых других, потому что большинство действий в браузерах выполняются с помощью мыши. В DOM Level 3 Events эта группа включает девять событий.

- `click` — генерируется, когда пользователь щелкает основной кнопкой мыши (обычно левой) или нажимает клавишу Enter. То, что обработчики `onclick` могут запускаться и клавиатурой, и мышью, помогает упростить работу со страницей для людей с ограниченными возможностями.
- `dblclick` — генерируется, когда пользователь дважды щелкает основной кнопкой мыши (обычно левой). Это событие отсутствует в DOM Level 2 Events, но благодаря широкой поддержке оно было стандартизировано в DOM Level 3 Events.
- `mousedown` — генерируется, когда пользователь нажимает любую кнопку мыши. Это событие не может генерироваться с клавиатуры.
- `mouseenter` — генерируется при наведении указателя мыши на элемент. Это событие не всплывает и не генерируется при пересечении границ потомков элемента. Оно было добавлено в DOM Level 3 Events и поддерживается в Internet Explorer, Firefox 9+ и Opera.
- `mouseleave` — генерируется при смещении указателя мыши, находящегося на элементе, за его пределы. Это событие не всплывает и не генерируется при пересечении границ потомков элемента. Оно было добавлено в DOM Level 3 Events и поддерживается в Internet Explorer, Firefox 9+ и Opera.
- `mousemove` — многократно генерируется при перемещении указателя мыши на элементе. Это событие не может генерироваться с клавиатуры.
- `mouseout` — генерируется при перемещении указателя мыши, находящегося на элементе, в область другого элемента. Новый элемент может находиться вне исходного или быть его дочерним элементом. Это событие не может генерироваться с клавиатуры.
- `mouseover` — генерируется при наведении указателя мыши на элемент. Это событие не может генерироваться с клавиатуры.

- `mouseup` — генерируется, когда пользователь отпускает кнопку мыши. Это событие не может генерироваться с клавиатуры.

События мыши поддерживаются всеми элементами страницы. Все события мыши, кроме `mouseenter` и `mouseleave`, всплывают и могут быть отменены, что может влиять на другие события из-за связей между ними.

Событие `click` может возникнуть, только если вслед за событием `mousedown` для того же элемента генерируется событие `mouseup`; если отменить одно из них, событие `click` не возникнет. Аналогичным образом событие `dbclick` требует двух событий `click`. Если отменить одно из них (`mousedown` или `mouseup`), событие `dblclick` не возникнет. Эти четыре события мыши всегда генерируются в следующем порядке:

1. `mousedown`.
2. `mouseup`.
3. `click`.
4. `mousedown`.
5. `mouseup`.
6. `click`.
7. `dblclick`.

Таким образом, события `click` и `dblclick` зависят от других событий, а `mousedown` и `mouseup` — нет.

Internet Explorer до версии 8 включительно пропускает вторые события `mousedown` и `click` при двойном щелчке:

1. `mousedown`.
2. `mouseup`.
3. `click`.
4. `mouseup`.
5. `dblclick`.

В Internet Explorer 9 этот дефект устранен, и события генерируются правильно.

DOM Level 2 Events включает все указанные события, кроме `dblclick`, `mouseenter` и `mouseleave`. Узнать, поддерживается ли эта спецификация, можно следующим образом:

```
var isSupported = document.implementation.hasFeature("MouseEvents", "2.0");
```

А так можно выяснить, поддерживает ли браузер все события, описываемые в этом разделе:

```
var isSupported = document.implementation.hasFeature("MouseEvent", "3.0")
```

Заметьте, что компонент DOM Level 3 называется `"MouseEvent"`, а не `"MouseEvents"`.

Группа событий мыши включает также *событие колесика* (wheel event). Оно называется `mousewheel` и используется для работы не только с колесиком мыши, но и с похожими устройствами, такими как сенсорная панель в Mac.

Клиентские координаты

Каждое событие мыши происходит в конкретном месте области просмотра, при этом координаты указателя мыши сохраняются в свойствах `clientX` и `clientY` объекта `event`, которые поддерживаются во всех браузерах. Клиентские координаты (client coordinates) области просмотра показаны на рис. 13.4.

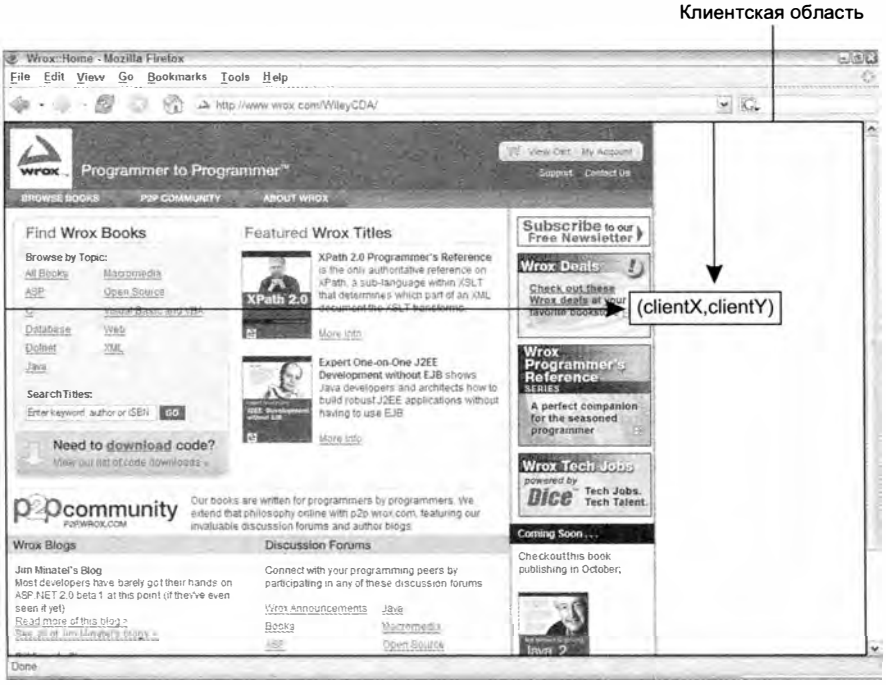

Рис. 13.4

Получить клиентские координаты события мыши можно следующим образом:

Листинг ClientCoordinatesExample01.htm


```
var div = document.getElementById("myDiv");
EventUtil.addHandler(div, "click", function(event){
    event = EventUtil.getEvent(event);
    alert("Client coordinates: " + event.clientX + "," + event.clientY);
});
```

Этот обработчик события `click` элемента `<div>` выводит на экран клиентские координаты щелчка. Имейте в виду, что они не отражают состояние прокрутки, а потому не показывают расположение курсора на странице.

Страничные координаты

В то время как клиентские координаты указывают место возникновения события в области просмотра, *страничные координаты* (page coordinates) определяют это место на странице. Они представлены свойствами `pageX` и `pageY` объекта `event` и рассчитываются относительно левого и верхнего краев самой страницы, а не области просмотра.

Получить страничные координаты события мыши можно следующим образом:

Листинг PageCoordinatesExample01.htm


```
var div = document.getElementById("myDiv");
EventUtil.addHandler(div, "click", function(event){
    event = EventUtil.getEvent(event);
    alert("Page coordinates: " + event.pageX + "," + event.pageY);
});
```

Если страница не прокручивалась, страничные координаты совпадают с клиентскими.

В Internet Explorer 8 и более ранних версий объект `event` не поддерживает страничные координаты, но их можно вычислить по клиентским координатам и свойствам прокрутки. Для этого нужно использовать свойства `scrollLeft` и `scrollTop` объекта `document.body` (в режиме совместимости) или `document.documentElement` (в стандартном режиме):

Листинг PageCoordinatesExample01.htm

```
var div = document.getElementById("myDiv");
EventUtil.addHandler(div, "click", function(event){
    event = EventUtil.getEvent(event);
    var pageX = event.pageX,
        pageY = event.pageY;

    if (pageX === undefined){
        pageX = event.clientX + (document.body.scrollLeft ||
                document.documentElement.scrollLeft);
    }

    if (pageY === undefined){
        pageY = event.clientY + (document.body.scrollTop ||
                document.documentElement.scrollTop);
    }

    alert("Page coordinates: " + pageX + "," + pageY);
});
```

Экранные координаты

С помощью свойств `screenX` и `screenY` можно также определить координаты события мыши относительно всего экрана. *Экранные координаты* (screen coordinates) в браузере показаны на рис. 13.5.

Рис. 13.5

Получить экранные координаты события мыши можно следующим образом:

Листинг ScreenCoordinatesExample01.htm


```
var div = document.getElementById("myDiv");
EventUtil.addHandler(div, "click", function(event){
    event = EventUtil.getEvent(event);
    alert("Screen coordinates: " + event.screenX + "," + event.screenY);
});
```

Как и в предыдущих примерах, здесь обрабатывается событие `click` элемента `<div>`. Его обработчик отображает экранные координаты события.

Клавиши-модификаторы

Хотя события мыши инициируются преимущественно мышью, иногда для выбора нужного действия важно учесть состояние некоторых клавиш. Для изменения поведения событий мыши часто используются *клавиши-модификаторы* (modifier

keys) Shift, Ctrl, Alt и Meta, которым в DOM соответствуют свойства `shiftKey`, `ctrlKey`, `altKey` и `metaKey`. Каждое из них содержит логическое значение `true`, если клавиша нажата, или `false` в противном случае. С помощью этих свойств можно определить состояние клавиш-модификаторов при возникновении события мыши:

Листинг ModifierKeysExample01.htm

```
var div = document.getElementById("myDiv");
EventUtil.addHandler(div, "click", function(event){
    event = EventUtil.getEvent(event);
    var keys = new Array();

    if (event.shiftKey){
        keys.push("shift");
    }

    if (event.ctrlKey){
        keys.push("ctrl");
    }

    if (event.altKey){
        keys.push("alt");
    }

    if (event.metaKey){
        keys.push("meta");
    }

    alert("Keys: " + keys.join(","));

});
```

Этот обработчик `onclick` добавляет названия нажатых клавиш-модификаторов в массив `keys`, а затем выводит его содержимое в оповещении.

Internet Explorer 9, Firefox, Safari, Chrome и Opera поддерживают все четыре клавиши, а более ранние версии Internet Explorer не поддерживают свойство metaKey.

Связанные элементы

События `mouseover` и `mouseout` генерируются при перемещении указателя мыши из одного элемента на другой, а потому имеют отношение сразу к двум элементам. Целевым элементом события `mouseover` является элемент, на который наводится указатель, а связанным элементом — элемент, где указатель находился ранее, тогда как у события `mouseout` все наоборот. Рассмотрим следующую HTML-страницу:

Листинг RelatedElementsExample01.htm


```
<!DOCTYPE html>
<html>
<head>
    <title>Related Elements Example</title>
```

```
</head>
<body>
    <div id="myDiv" style="background-color:red;height:100px;width:100px;">
    </div>
</body>
</html>
```

На этой странице отображается единственный элемент `<div>`. Если сначала указатель мыши находится на нем, а затем перемещается за его пределы, для элемента `<div>` генерируется событие `mouseout`, при этом связанным элементом является `<body>`. Одновременно для элемента `<body>` генерируется событие `mouseover`, при этом связанным считается элемент `<div>`.

При возникновении событий `mouseover` и `mouseout` связанный элемент сохраняется в свойстве `relatedTarget` объекта `event`, которое при любых других событиях равно `null`. Браузеры Internet Explorer 8 и более ранних версий не поддерживают свойство `relatedTarget`, но предоставляют доступ к связанному элементу с помощью других свойств. При возникновении события `mouseover` связанный элемент доступен в Internet Explorer как свойство `fromElement`, а при возникновении события `mouseout` — как свойство `toElement` (Internet Explorer 9 поддерживает все свойства). Теперь мы можем добавить в объект `EventUtil` кроссбраузерный метод получения связанного элемента:

Листинг EventUtil.js


```
var EventUtil = {

    // другой код

    getRelatedTarget: function(event){
        if (event.relatedTarget){
            return event.relatedTarget;
        } else if (event.toElement){
            return event.toElement;
        } else if (event.fromElement){
            return event.fromElement;
        } else {
            return null;
        }
    },

    // другой код

};
```

Как и в предыдущих кроссбраузерных методах, для выбора возвращаемого значения здесь применяется механизм распознавания возможностей. Метод `EventUtil.getRelatedTarget()` можно использовать следующим образом:

Листинг RelatedElementsExample01.htm

```
var div = document.getElementById("myDiv");
EventUtil.addHandler(div, "mouseout", function(event){
    event = EventUtil.getEvent(event);
```

```
    var target = EventUtil.getTarget(event);
    var relatedTarget = EventUtil.getRelatedTarget(event);
    alert("Moused out of " + target.tagName +
          " to " + relatedTarget.tagName);
});
```

Этот код обрабатывает событие `mouseout` элемента `<div>`, выводя оповещение со сведениями о том, откуда и куда был перемещен указатель мыши.

Кнопки мыши

Событие `click` генерируется только при щелчке на основной кнопке мыши (или при нажатии клавиши Enter на клавиатуре), так что для его обработки сведения о кнопке не нужны. Для событий `mousedown` и `mouseup` в объекте `event` доступно свойство `button`, указывающее кнопку, которая была нажата или отпущена. В DOM оно может иметь одно из трех значений: 0 соответствует основной кнопке мыши, 1 — средней (обычно это кнопка-колесико), а 2 — дополнительной. В традиционной конфигурации основной кнопкой мыши является левая, а дополнительной правая.

В Internet Explorer до версии 8 свойство `button` тоже доступно, но имеет совершенно другие значения:

- 0 — никакая кнопка не нажата;
- 1 — нажата основная кнопка мыши;
- 2 — нажата дополнительная кнопка мыши;
- 3 — нажаты основная и дополнительная кнопки мыши;
- 4 — нажата средняя кнопка мыши;
- 5 — нажаты основная и средняя кнопки мыши;
- 6 — нажаты дополнительная и средняя кнопки мыши;
- 7 — нажаты все три кнопки мыши.

Как видите, в модели DOM свойство `button` гораздо проще, чем в Internet Explorer, но при этом столь же полезно, потому что сочетания кнопок мыши почти не используются. На практике все модели работы с кнопками мыши обычно преобразуют в формат DOM, потому что он встроен во все браузеры, кроме Internet Explorer 8 и более ранних версий. Преобразование кодов основной, средней и дополнительной кнопок мыши из Internet Explorer тривиально, а сочетания кнопок транслируются в нажатие одной кнопки, при этом приоритет всегда отдается основной кнопке. Например, коды 5 и 7 в Internet Explorer соответствуют значению 0 в DOM.

Так как свойство `button` доступно во всех браузерах, узнать схему работы с кнопками мыши путем распознавания возможностей не получится. Вместо этого мы можем распознать браузеры, поддерживающие DOM-события мыши, с помощью метода `hasFeature()`. Вот кроссбраузерный метод `getButton()`:

Листинг EventUtil.js


```
var EventUtil = {

    // другой код

    getButton: function(event){
        if (document.implementation.hasFeature("MouseEvents", "2.0")){
            return event.button;
        } else {
            switch(event.button){
                case 0:
                case 1:
                case 3:
                case 5:
                case 7:
                    return 0;
                case 2:
                case 6:
                    return 2;
                case 4:
                    return 1;
            }
        }
    },

    // другой код

};
```

Если компонент `"MouseEvents"` доступен, свойство `button` объекта `event` уже содержит правильные значения, в противном случае мы, вероятно, имеем дело с Internet Explorer, поэтому значения следует нормализовать. Этот метод можно использовать следующим образом:

Листинг ButtonExample01.htm


```
var div = document.getElementById("myDiv");
EventUtil.addHandler(div, "mousedown", function(event){
    event = EventUtil.getEvent(event);
    alert(EventUtil.getButton(event));
});
```

Данный обработчик события `mousedown` выводит на экран оповещение с кодом нажатой кнопки мыши.

При обработке события mouseup значением button будет код отпущенной кнопки.

Дополнительные сведения о событиях

В спецификации DOM Level 2 Events для объекта `event` определено свойство `detail`, которое предоставляет дополнительные сведения о событии. У событий мыши оно

содержит число, указывающее, сколько раз был выполнен щелчок в указанном месте. Щелчком считается последовательность событий `mousedown` и `mouseup` в одной точке. Значение `detail` начинается с единицы и увеличивается при каждом щелчке. Если между событиями `mousedown` и `mouseup` указатель мыши перемещается, свойство `detail` обнуляется.

Internet Explorer предоставляет также следующие сведения о каждом событии мыши:

- `altLeft` — логическое значение, указывающее, нажата ли левая клавиша Alt (если это свойство равно `true`, то и свойство `altKey` равно `true`);
- `ctrlLeft` — логическое значение, указывающее, нажата ли левая клавиша Ctrl (если это свойство равно `true`, то и свойство `ctrlKey` равно `true`);
- `offsetX` — координата *x* указателя мыши относительно границ целевого элемента;
- `offsetY` — координата *y* указателя мыши относительно границ целевого элемента;
- `shiftLeft` — логическое значение, указывающее, нажата ли левая клавиша Shift (если это свойство равно `true`, то и свойство `shiftKey` равно `true`).

Польза от этих свойств невелика, потому что они доступны только в Internet Explorer и предоставляют сведения, которые можно получить иными способами.

Событие mousewheel

Событие `mousewheel` появилось в Internet Explorer 6 и позднее было добавлено в Opera, Chrome и Safari. Оно генерируется для любых элементов при вращении колесика мыши и всплывает к объекту `document` (в Internet Explorer 8) или `window` (в Internet Explorer 9+, Opera, Chrome и Safari). Его объект `event` содержит всю стандартную информацию события мыши, а также дополнительное свойство `wheelDelta`, значением которого является число, кратное 120, — положительное при вращении колесика вперед и отрицательное при вращении назад (рис. 13.6).

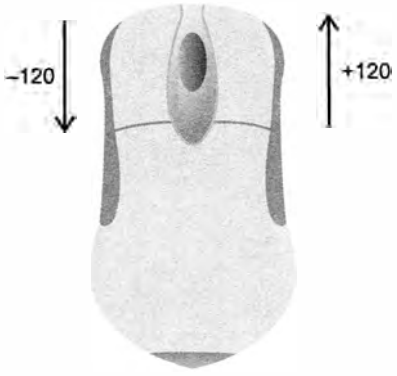

Рис. 13.6

Обработчик события mousewheel можно назначить любому элементу на странице или объекту document для обработки всех взаимодействий с колесиком мыши, например:

```
EventUtil.addHandler(document, "mousewheel", function(event){
    event = EventUtil.getEvent(event);
    alert(event.wheelDelta);
});
```

Это обработчик просто отображает значение wheelDelta. В большинстве случаев нужно узнать лишь направление вращения колесика, что можно легко определить по знаку этого значения.

Имейте в виду, что в Opera до версии 9.5 знак wheelDelta был обратным. Если вам нужно поддерживать ранние версии Opera, для определения значения wheelDelta потребуется распознать браузер:

Листинг MouseWheelEventExample01.htm


```
EventUtil.addHandler(document, "mousewheel", function(event){
    event = EventUtil.getEvent(event);
    var delta = (client.engine.opera && client.engine.opera < 9.5 ?
                -event.wheelDelta : event.wheelDelta);
    alert(delta);
});
```

Для распознавания ранних версий Opera здесь используется созданный ранее объект client (см. главу 9).

Благодаря популярности и широкой поддержке события mousewheel оно было добавлено в HTML5.

В Firefox при вращении колесика мыши генерируется похожее событие DOMMouseScroll. Как и mousewheel, оно считается событием мыши и имеет все соответствующие свойства. Данные колесика предоставляет свойство detail, которое содержит число, кратное 3, — отрицательное при вращении колесика вперед и положительное при вращении назад (рис. 13.7).

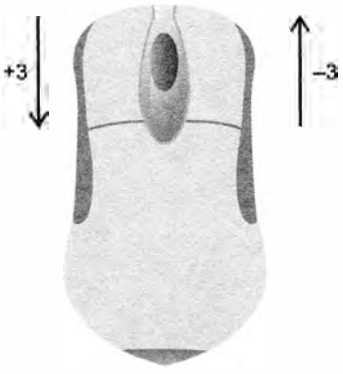

Рис. 13.7

Событие DOMMouseScroll может быть обработано у любого элемента страницы и всплывает к объекту window. Назначить его обработчик можно следующим образом:

Листинг DOMMouseScrollEventExample01.htm

```
EventUtil.addHandler(window, "DOMMouseScroll", function(event){
    event = EventUtil.getEvent(event);
    alert(event.detail);
});
```

При вращении колесика этот код выводит значение свойства detail.

Для кроссбраузерной обработки вращения колесика прежде всего нужно добавить в объект EventUtil метод, возвращающий нормализованное значение «дельта»:

Листинг EventUtil.js

```
var EventUtil = {

    // другой код

    getWheelDelta: function(event){
        if (event.wheelDelta){
            return (client.engine.opera && client.engine.opera < 9.5 ?
                    -event.wheelDelta : event.wheelDelta);
        } else {
            return -event.detail * 40;
        }
    },

    // другой код

};
```

Метод getWheelDelta() проверяет, есть ли у объекта event свойство wheelDelta, и если да, распознает браузер для определения правильного значения. Если свойства wheelDelta нет, то нужное значение содержится в свойстве detail. Чтобы преобразовать его в формат, принятый в других браузерах, метод меняет знак значения и умножает его на 40. Используя этот метод, можно обрабатывать события mousewheel и DOMMouseScroll согласованным образом:

Листинг CrossBrowserMouseWheelExample01.htm


```
(function(){

    function handleMouseWheel(event){
        event = EventUtil.getEvent(event);
        var delta = EventUtil.getWheelDelta(event);
        alert(delta);
    }

    EventUtil.addHandler(document, "mousewheel", handleMouseWheel);
    EventUtil.addHandler(document, "DOMMouseScroll", handleMouseWheel);

})();
```

Чтобы не засорять глобальную область видимости дополнительными функциями, этот код выполняется в закрытой области видимости. Оба события обрабатываются функцией `handleMouseWheel()`, назначение которой несуществующему событию просто игнорируется. Использование метода `EventUtil.getWheelDelta()` обеспечивает нормальную обработку события в обоих случаях.

Поддержка устройств с сенсорным вводом

Устройства с сенсорным вводом с системой iOS или Android не поддерживают мышь, поэтому написание кода для них имеет ряд особенностей.

- Событие `dblclick` не поддерживается. При двойном щелчке в окне браузера увеличивается масштаб страницы, и переопределить это поведение нельзя.
- Касание элемента, поддерживающего щелчок, инициирует событие `mousemove`. Если это приводит к изменению контента, другие события не генерируются; если контент не изменяется, по очереди генерируются события `mousedown`, `mouseup` и `click`. При касании элемента, не поддерживающего щелчок, никакие события не возникают. Элемент, поддерживающий щелчок, — это элемент, при щелчке на котором выполняется какое-то действие, предлагаемое по умолчанию (такой как ссылка), или вызывается обработчик события `click`.
- Событие `mousemove` также инициирует события `mouseover` и `mouseout`.
- Если пользователь касается экрана двумя пальцами и в результате их перемещения страница прокручивается, возникают события `mousewheel` и `scroll`.

Специальные возможности

Если нужно обеспечить доступ к веб-приложению или веб-сайту для людей, использующих программы чтения с экрана, требуется повышенное внимание к событиям мыши. Как уже отмечалось, событие `click` может быть сгенерировано с помощью клавиши Enter, но другие события нельзя инициировать с клавиатуры. Не рекомендуется использовать их для отображения функциональных элементов или выполнения кода, так как это серьезно ограничивает возможности людей с плохим зрением. Перечислим советы по обработке событий мыши в контексте специальных возможностей.

- Используйте для выполнения кода событие `click`. Многие люди с нормальным зрением утверждают, что приложение воспринимается как более быстрое, если код запускается по событию `mousedown`, но программам чтения с экрана такой код недоступен.
- Не используйте событие `mouseover` для показа новых пунктов меню и подобных элементов, потому что программы чтения с экрана не могут инициировать его. Если другие варианты не подходят, попробуйте добавить сочетания клавиш для вывода тех же элементов.
- Не используйте событие `dblclick` для выполнения важных действий. Его невозможно сгенерировать с клавиатуры.

Следование этим простым правилам значительно упростит работу с приложением или сайтом для людей с ограниченными возможностями.

Дополнительные сведения о доступности веб-страниц для людей с ограниченными возможностями см. на сайтах www.webaim.org и http://accessibility.yahoo.com.

События клавиатуры и редактирования текста

События клавиатуры (keyboard events) первоначально входили в спецификацию DOM Level 2 Events, но соответствующий раздел был удален еще до принятия окончательной версии спецификации. В результате поддержка событий клавиатуры в браузерах основана преимущественно на реализациях DOM Level 0.

События клавиатуры из DOM Level 3 Events впервые были полностью реализованы в Internet Explorer 9. Разработчики других браузеров также начали внедрять этот стандарт, но и унаследованных реализаций все еще много.

Опишем три доступных события клавиатуры.

- `keydown` — генерируется при нажатии клавиши и повторяется, пока клавиша нажата.
- `keypress` — генерируется при нажатии символьной клавиши и повторяется, пока она нажата. Это событие также возникает для клавиши Esc. В DOM Level 3 Events оно объявлено устаревшим, а вместо него предложено использовать событие `textInput`.
- `keyup` — генерируется при отпускании клавиши.

Эти события наиболее наглядны при вводе данных в текстовом поле, но вообще их поддерживают все элементы.

Единственное событие редактирования текста называется `textInput`. Оно дополняет событие `keypress`, упрощая перехват вводимого текста до его отображения. Это событие генерируется непосредственно перед вставкой текста в текстовое поле.

Когда пользователь нажимает символьную клавишу, сначала генерируется событие `keydown`, за которым следует событие `keypress`, а за ним — `keyup`. События `keydown` и `keypress` предшествуют изменению содержимого текстового поля, а событие `keyup` возникает после изменения. Если пользователь удерживает символьную клавишу нажатой, события `keydown` и `keypress` генерируются многократно, пока клавиша не будет отпущена.

При нажатии несимвольной клавиши генерируется событие `keydown`, а вслед за ним — `keyup`. Если такая клавиша удерживается нажатой, событие `keydown` повторяется, а при отпускании клавиши возникает событие `keyup`.

События клавиатуры поддерживают те же клавиши-модификаторы, что и события мыши, и имеют аналогичные свойства shiftKey, ctrlKey, altKey и metaKey. В Internet Explorer 8 и более ранних версий свойство metaKey не поддерживается.

Коды клавиш

Для событий keydown и keyup свойству keyCode объекта event назначается код соответствующей клавиши. У алфавитно-цифровых клавиш он равен ASCII-коду буквы в нижнем регистре или числа: например, для клавиши 7 свойству keyCode присваивается значение 55, а для клавиши A — 65 (независимо от состояния клавиши Shift). Свойство keyCode есть у объекта event и в DOM, и в Internet Explorer. Вот пример кода с ним:

Листинг KeyUpEventExample01.htm


```
var textbox = document.getElementById("myText");
EventUtil.addHandler(textbox, "keyup", function(event){
    event = EventUtil.getEvent(event);
    alert(event.keyCode);
});
```

Этот обработчик выводит на экран значение keyCode при каждом событии keyup. Коды несимвольных клавиш приведены в таблице.

Клавиша	Код клавиши	Клавиша	Код клавиши
Backspace	8	8 на цифровой клавиатуре	104
Tab	9	9 на цифровой клавиатуре	105
Enter	13	Плюс на цифровой клавиатуре	107
Shift	16	Минус на цифровой и обычной клавиатурах	109
Ctrl	17	Точка на цифровой клавиатуре	110
Alt	18	Косая черта на цифровой клавиатуре	111
Pause/Break	19	F1	112
Caps Lock	20	F2	113
Esc	27	F3	114
Page Up	33	F4	115
Page Down	34	F5	116
End	35	F6	117
Home	36	F7	118
Стрелка влево	37	F8	119
Стрелка вверх	38	F9	120
Стрелка вправо	39	F10	121
Стрелка вниз	40	F11	122

Клавиша	Код клавиши	Клавиша	Код клавиши
Ins	45	F12	123
Del	46	Num Lock	144
Левая клавиша Windows	91	Scroll Lock	145
Правая клавиша Windows	92	Точка с запятой (IE/Safari/Chrome)	186
Клавиша контекстного меню	93	Точка с запятой (Opera/FF)	59
0 на цифровой клавиатуре	96	Меньше	188
1 на цифровой клавиатуре	97	Больше	190
2 на цифровой клавиатуре	98	Косая черта	191
3 на цифровой клавиатуре	99	Гравис (`)	192
4 на цифровой клавиатуре	100	Равно	61
5 на цифровой клавиатуре	101	Левая квадратная скобка	219
6 на цифровой клавиатуре	102	Обратная косая черта (\)	220
7 на цифровой клавиатуре	103	Правая квадратная скобка	221
		Одинарная кавычка	222

У событий `keydown` и `keyup` есть одна странность. Firefox и Opera возвращают для точки с запятой код 59, который совпадает с ASCII-кодом этого знака, а Internet Explorer, Chrome и Safari возвращают 186, то есть код клавиши на клавиатуре.

Коды символов

Событие `keypress` сигнализирует о том, что нажатие клавиши изменило текст на экране. Для клавиш, которые добавляют или удаляют символ, оно генерируется во всех браузерах, а способ обработки нажатий других клавиш зависит от браузера. Поскольку реализация спецификации DOM Level 3 Events началась сравнительно недавно, в браузерах существуют заметные различия.

В Internet Explorer 9+, Firefox, Chrome и Safari у объекта `event` есть свойство `charCode`, которое задается только для события `keypress`, при этом ему назначается ASCII-код символа нажатой клавиши. Свойство `keyCode` в этом случае обычно имеет значение 0 или содержит код нажатой клавиши. В Internet Explorer 8 и более ранних версий и Opera ASCII-код символа передается в свойстве `keyCode`. Таким образом, чтобы получить код символа кроссбраузерным способом, нужно сначала проверить, используется ли свойство `charCode`, и если нет, возвратить вместо него значение `keyCode`:

Листинг EventUtil.js


```
var EventUtil = {

    // другой код
```

```
    getCharCode: function(event){
        if (typeof event.charCode == "number"){
            return event.charCode;
        } else {
            return event.keyCode;
        }
    },

    // другой код
};
```

Этот метод проверяет, содержит ли свойство charCode число (если свойство не поддерживается, оно не определено). Если да, метод возвращает его, в противном случае возвращается значение keyCode. Использовать метод можно следующим образом:

Листинг KeyPressEventExample01.htm

```
var textbox = document.getElementById("myText");
EventUtil.addHandler(textbox, "keypress", function(event){
    event = EventUtil.getEvent(event);
    alert(EventUtil.getCharCode(event));
});
```

Когда код знака получен, можно преобразовать его в фактический символ методом String.fromCharCode().

Изменения в DOM Level 3

В спецификации DOM Level 3 Events события клавиатуры претерпели ряд изменений. Например, свойство charCode заменено свойствами key и char.

Свойство key предназначено для замены свойства keyCode и содержит строку. При нажатии символьной клавиши ему присваивается символ (например, "k" или "M"), а при нажатии несимвольной клавиши — название этой клавиши (например, "Shift" или "Down"). Свойство char при нажатии символьной клавиши работает так же, а если нажимается несимвольная клавиша, получает значение null.

Internet Explorer 9 поддерживает свойство key, но не char. Safari 5 и Chrome поддерживают свойство keyIdentifier, которое возвращает то же значение, что и key, если нажимается несимвольная клавиша (например, Shift). Для символьных клавиш свойство keyIdentifier возвращает строку формата «U+0000», содержащую символ в кодировке Юникод.

Листинг DOMLevel3KeyPropertyExample01.htm


```
var textbox = document.getElementById("myText");
EventUtil.addHandler(textbox, "keypress", function(event){
    event = EventUtil.getEvent(event);
    var identifier = event.key || event.keyIdentifier;
    if (identifier){
        alert(identifier);
    }
});
```

Поскольку свойства `key`, `keyIdentifier` и `char` доступны не во всех браузерах, использовать их не рекомендуется.

В DOM Level 3 Events также добавлено числовое свойство `location`, указывающее, где была нажата клавиша. Его возможные значения таковы: 0 — клавиатура, предлагаемая по умолчанию; 1 — левая часть клавиатуры (например, левая клавиша Alt); 2 — правая часть клавиатуры (например, правая клавиша Shift); 3 — числовая клавиатура; 4 — клавиатура мобильного устройства (виртуальная); 5 — джойстик (например, контроллер Nintendo Wii). Это свойство доступно в Internet Explorer 9. В Safari 5 и Chrome есть идентичное свойство `keyLocation`, но из-за ошибки оно может содержать только число 3, если нажата клавиша на цифровой клавиатуре, и 0 в остальных случаях.

Листинг DOMLevel3LocationPropertyExample01.htm

```
var textbox = document.getElementById("myText");
EventUtil.addHandler(textbox, "keypress", function(event){
    event = EventUtil.getEvent(event);
    var loc = event.location || event.keyLocation;
    if (loc){
        alert(loc);
    }
});
```

Из-за ограниченной поддержки использовать свойство `location` в кроссбраузерном коде не следует.

Наконец, к объекту `event` добавлен также метод `getModifierState()`. Он принимает строковое значение `"Shift"`, `"Control"`, `"Alt"`, `"AltGraph"` или `"Meta"`, указывающее клавишу-модификатор, которую нужно проверить. Если указанный модификатор активен (клавиша нажата), метод возвращает `true`, иначе — `false`:

Листинг DOMLevel3LocationGetModifierStateExample01.htm


```
var textbox = document.getElementById("myText");
EventUtil.addHandler(textbox, "keypress", function(event){
    event = EventUtil.getEvent(event);
    if (event.getModifierState){
        alert(event.getModifierState("Shift"));
    }
});
```

Некоторые из этих сведений можно также получить с помощью свойств `shiftKey`, `altKey`, `ctrlKey` и `metaKey` объекта `event`. Метод `getModifierState()` реализован только в Internet Explorer 9.

Событие textInput

В DOM Level 3 Events определено событие `textInput`, которое генерируется при вводе символа в редактируемой области. Оно предназначено для замены события `keypress` и работает немного иначе. Во-первых, событие `keypress` генерируется для

любого элемента, который может получить фокус ввода, а `textInput` — только для областей с возможностью редактирования. Во-вторых, событие `textInput` генерируется только для символьных клавиш, а событие `keypress` — для клавиш, способных изменять текст как угодно (включая Backspace).

Для события `textInput` в объекте `event` доступно свойство `data`, которое содержит введенный символ (а не код символа). Символ представляется точно, например в результате нажатия клавиши S без клавиши Shift свойство `data` получает значение `"s"`, а при нажатой клавише Shift — значение `"S"`.

Событие `textInput` можно обработать следующим образом:

Листинг TextInputEventExample01.htm

```
var textbox = document.getElementById("myText");
EventUtil.addHandler(textbox, "textInput", function(event){
    event = EventUtil.getEvent(event);
    alert(event.data);
});
```

В этом примере символ, введенный в текстовом поле, отображается в оповещении.

У объекта `event` есть также свойство `inputMethod`, которое показывает способ ввода текста в элемент управления:

- 0 — браузер не смог определить способ ввода;
- 1 — текст был введен с клавиатуры;
- 2 — текст был вставлен;
- 3 — текст был добавлен путем перетаскивания;
- 4 — текст был введен с помощью редактора метода ввода (IME);
- 5 — текст был добавлен путем выбора элемента формы;
- 6 — текст был введен от руки (например, с помощью стилуса);
- 7 — текст был введен голосом;
- 8 — текст был введен несколькими способами;
- 9 — текст был добавлен программно.

С помощью этого свойства можно узнать, как текст был введен в элемент управления, чтобы проверить, допустим ли он.

Событие `textInput` поддерживается в Internet Explorer 9+, Safari и Chrome. Свойство `inputMethod` доступно только в Internet Explorer.

События клавиатуры на других устройствах

Игровая приставка Nintendo Wii генерирует события клавиатуры при нажатии некоторых кнопок на пульте дистанционного управления Wii. Коды клавиш показаны на рис. 13.8.

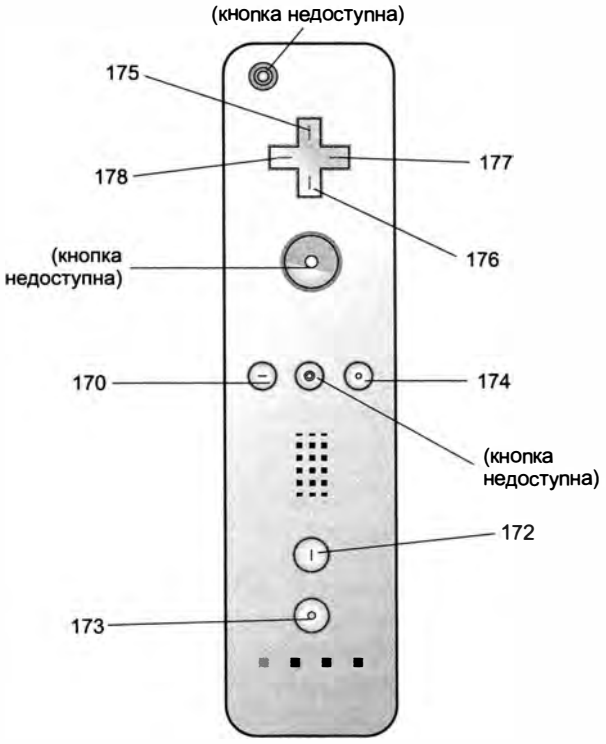

Рис. 13.8

События клавиатуры генерируются при нажатии крестообразной кнопки (коды клавиш 175–178), кнопок с минусом (170), плюсом (174), а также кнопок 1 (172) и 2 (173). Обработать нажатия кнопок A, B и Home, а также кнопки питания невозможно.

Safari в системе iOS и WebKit в Android генерируют события клавиатуры при использовании экранной клавиатуры.

События композиции

События композиции (composition events) представлены в DOM Level 3 Events для обработки сложных сочетаний клавиш, которые обычно используются в редакторах метода ввода (Input Method Editor, IME) для ввода символов, отсутствующих на физической клавиатуре. Например, с помощью IME на клавиатуре с латиницей можно вводить символы японского алфавита. Для ввода единственного символа в IME часто требуется нажать несколько клавиш. Три события композиции помогают распознавать такой ввод и работать с ним:

- `compositionstart` — генерируется при открытии IME-системы композиции текста, отмечая начало ввода символов;

- `compositionupdate` — генерируется при вставке нового символа в поле ввода;
- `compositionend` — генерируется при закрытии системы композиции текста, информируя о возврате к обычному вводу с клавиатуры.

События композиции во многом похожи на события редактирования текста. Целевым элементом события композиции является поле ввода, принимающее текст, а единственное дополнительное свойство `data` содержит одно из следующих значений:

- при обработке события `compositionstart` — редактируемый текст (например, если текст был выделен и сейчас будет заменен);
- при обработке события `compositionupdate` — вставляемый символ;
- при обработке события `compositionend` — весь текст, введенный в течение сеанса композиции.

Как и события редактирования текста, события композиции можно использовать для фильтрации ввода. Назначить им обработчики можно следующим образом:

Листинг CompositionEventsExample01.htm


```
var textbox = document.getElementById("myText");
EventUtil.addHandler(textbox, "compositionstart", function(event){
    event = EventUtil.getEvent(event);
    alert(event.data);
});

EventUtil.addHandler(textbox, "compositionupdate", function(event){
    event = EventUtil.getEvent(event);
    alert(event.data);
});

EventUtil.addHandler(textbox, "compositionend", function(event){
    event = EventUtil.getEvent(event);
    alert(event.data);
});
```

На 2011 год события композиции поддерживаются только в Internet Explorer 9+ и почти бесполезны при написании кроссбраузерного кода. Определить, поддерживает ли их браузер, можно так:

```
var isSupported =
    document.implementation.hasFeature("CompositionEvent", "3.0");
```

События изменения DOM-структуры

События изменения DOM-структуры (mutation events) уведомляют об изменениях частей DOM-структуры. Они не зависят от языка и должны работать в любом DOM-документе с XML- или HTML-кодом. В DOM Level 2 эта группа включает следующие события.

- `DOMSubtreeModified` — универсальное событие, которое генерируется при любом изменении DOM-структуры после всех остальных событий.
- `DOMNodeInserted` — генерируется после вставки дочернего узла.
- `DOMNodeRemoved` — генерируется перед удалением дочернего узла.
- `DOMNodeInsertedIntoDocument` — генерируется после вставки узла или поддерева, в котором он содержится (после события `DOMNodeInserted`). Это событие объявлено устаревшим в DOM Level 3 Events, и использовать его не следует.
- `DOMNodeRemovedFromDocument` — генерируется перед удалением узла или поддерева, в котором он содержится (после события `DOMNodeRemoved`). Это событие объявлено устаревшим в DOM Level 3 Events, и использовать его не следует.
- `DOMAttrModified` — генерируется при изменении атрибута. Это событие объявлено устаревшим в DOM Level 3 Events, и использовать его не следует.
- `DOMCharacterDataModified` — генерируется при изменении значения текстового узла. Это событие объявлено устаревшим в DOM Level 3 Events, и использовать его не следует.

Определить, поддерживает ли браузер события изменения DOM Level 2, можно следующим образом:

```
var isSupported =
    document.implementation.hasFeature("MutationEvents", "2.0");
```

В Internet Explorer 8 и более ранних версий никакие события изменения DOM-структуры не поддерживаются. Сведения о поддержке событий, которые не устарели в других браузерах, приведены в таблице.

Событие	Opera 9+	Firefox 3+	Safari 3+ и Chrome	Internet Explorer 9+
DOMSubtreeModified	—	Да	Да	Да
DOMNodeInserted	Да	Да	Да	Да
DOMNodeRemoved	Да	Да	Да	Да

В этом разделе мы рассмотрим только те события, которые сохранили актуальность.

Удаление узла

При удалении узла из DOM-структуры с помощью метода `removeChild()` или `replaceChild()` сначала генерируется событие `DOMNodeRemoved`. Его целевым элементом является удаляемый узел, а свойство `event.relatedNode` содержит ссылку на родительский узел. В момент события узел еще не удален, так что его свойство `parentNode` все еще указывает на родительский узел (как и свойство `event.relatedNode`). Это событие всплывает, так что его можно обработать на любом уровне DOM.

Если у удаляемого узла есть дочерние узлы, для каждого из них, а затем и для самого удаляемого узла генерируется устаревшее событие DOMNodeRemovedFromDocument. Оно не всплывает, так что его обработчик вызывается только в том случае, если он подключен непосредственно к одному из дочерних узлов. Целевым элементом этого события является дочерний или удаляемый узел, а объект event не содержит никаких дополнительных сведений о событии.

После этого для родительского узла удаленного узла генерируется событие DOMSubtreeModified. Объект event не предоставляет дополнительных сведений о нем.

Чтобы понять, как все это работает на практике, рассмотрим следующую HTML-страницу:

```
<!DOCTYPE html>
<html>
<head>
    <title>Node Removal Events Example</title>
</head>
<body>
    <ul id="myList">
        <li>Item 1</li>
        <li>Item 2</li>
        <li>Item 3</li>
    </ul>
</body>
</html>
```

При удалении этого элемента <ul> события генерируются в таком порядке:

1. Для элемента <ul> генерируется событие DOMNodeRemoved, у которого свойство relatedNode указывает на узел document.body.
2. Для элемента <ul> генерируется событие DOMNodeRemovedFromDocument.
3. Для каждого элемента <li> и каждого дочернего текстового узла элемента <ul> генерируется событие DOMNodeRemovedFromDocument.
4. Для узла document.body генерируется событие DOMSubtreeModified, потому что элемент <ul> был его непосредственным дочерним элементом.

Протестировать этот процесс можно с помощью следующего JavaScript-кода:

```
EventUtil.addHandler(window, "load", function(event){
    var list = document.getElementById("myList");

    EventUtil.addHandler(document, "DOMSubtreeModified", function(event){
        alert(event.type);
        alert(event.target);
    });
    EventUtil.addHandler(document, "DOMNodeRemoved", function(event){
        alert(event.type);
        alert(event.target);
        alert(event.relatedNode);
```

```
    });
    EventUtil.addHandler(list.firstChild, "DOMNodeRemovedFromDocument",
      function(event){
        alert(event.type);
        alert(event.target);
    });

    list.parentNode.removeChild(list);
});
```

Этот код регистрирует обработчики событий `DOMSubtreeModified` и `DOMNodeRemoved` для объекта `document`, чтобы они обрабатывали все такие события. Событие `DOMNodeRemovedFromDocument` не всплывает, поэтому его обработчик добавляется непосредственно к первому дочернему узлу элемента `<ul>`, которым в браузерах, соответствующих DOM, является текстовый узел. Как только обработчики событий зарегистрированы, элемент `<ul>` удаляется из документа.

Вставка узла

При вставке узла в DOM с помощью метода `appendChild()`, `replaceChild()` или `insertBefore()` сначала генерируется событие `DOMNodeInserted`. Его целевым элементом является вставленный узел, а свойство `event.relatedNode` содержит ссылку на родительский узел. В момент события узел уже добавлен к новому родительскому узлу. Это событие всплывает, и его можно обработать на любом уровне DOM.

Затем для вставленного узла генерируется устаревшее событие `DOMNodeInsertedIntoDocument`, которое не всплывает, так что его обработчик необходимо зарегистрировать до вставки узла. Целевым элементом этого события является вставленный узел, а объект `event` не предоставляет никаких дополнительных сведений о событии.

Наконец, для родительского (по отношению к вставленному) узла генерируется событие `DOMSubtreeModified`.

Чтобы протестировать эти события, можно запустить для HTML-документа из предыдущего примера следующий JavaScript-код:

```
EventUtil.addHandler(window, "load", function(event){
    var list = document.getElementById("myList");
    var item = document.createElement("li");
    item.appendChild(document.createTextNode("Item 4"));

    EventUtil.addHandler(document, "DOMSubtreeModified", function(event){
        alert(event.type);
        alert(event.target);
    });
    EventUtil.addHandler(document, "DOMNodeInserted", function(event){
        alert(event.type);
        alert(event.target);
        alert(event.relatedNode);
    });
    EventUtil.addHandler(item, "DOMNodeInsertedIntoDocument",
```

```
    function(event){
      alert(event.type);
      alert(event.target);
  });

  list.appendChild(item);
});
```

Сначала этот код создает элемент `<li>` с текстом `"Item 4"`, после чего назначает обработчики событий `DOMSubtreeModified` и `DOMNodeInserted`. Они подключаются к документу, потому что эти события всплывают. Перед добавлением нового элемента к списку (родительскому узлу) также регистрируется обработчик события `DOMNodeInsertedIntoDocument`. Наконец, новый элемент добавляется в список с методом `appendChild()`, при этом начинают генерироваться события. Первым генерируется событие `DOMNodeInserted` для нового элемента `<li>`, при этом свойство `relatedNode` указывает на элемент `<ul>`. Затем для нового элемента `<li>` возникает событие `DOMNodeInsertedIntoDocument` и, наконец, для элемента `<ul>` генерируется событие `DOMSubtreeModified`.

События HTML5

В спецификации DOM определены не все события, поддерживаемые браузерами. Многие браузеры реализуют уникальные события для решения специфических задач. HTML5 определяет целый ряд дополнительных событий, которые в идеале должны поддерживаться браузерами, и в этом разделе мы обсудим некоторые из них. Имейте в виду, что это не исчерпывающий перечень всех событий.

Событие contextmenu

Контекстные меню, которые появляются при щелчке правой кнопкой мыши, появились в Windows 95 и вскоре были взяты на вооружение веб-разработчиками, но сразу возникли проблемы. Для использования контекстного меню требовался какой-то механизм, уведомляющий о том, что меню нужно вывести на экран (в Windows оно вызывается правой кнопкой мыши, а в Mac — нажатием клавиши Ctrl с одновременным щелчком мыши), и позволяющий отменить вывод контекстного меню, предлагаемого по умолчанию. Для решения этих задач было реализовано событие `contextmenu`, генерируемое перед выводом контекстного меню.

Событие `contextmenu` всплывает, так что его можно обработать для всей страницы, назначив обработчик объекту `document`. Целевым элементом события является элемент, меню которого вызывается. Чтобы отменить событие `contextmenu`, нужно использовать метод `event.preventDefault()` в браузерах, соответствующих DOM, или присвоить свойству `event.returnValue` значение `false` в Internet Explorer 8 и более ранних версий. Событие `contextmenu` считается событием мыши, а потому имеет все свойства, связанные с положением указателя. Как правило, разработчики выводят на экран собственные контекстные меню в обработчике `oncontextmenu`

и скрывают их в обработчике onclick. Возьмем для примера следующую HTML-страницу:

Листинг ContextMenuEventExample01.htm


```
<!DOCTYPE html>
<html>
<head>
    <title>ContextMenu Event Example</title>
</head>
<body>
    <div id=" myDiv">Right click or Ctrl+click me to get a custom
        context menu. Click anywhere else to get the default context
        menu.</div>
    <ul id="myMenu" style="position:absolute;visibility:hidden;
        background-color:silver">
        <li><a href="http://www.nczonline.net">Сайт Николаса</a></li>
        <li><a href="http://www.wrox.com">Сайт Wrox</a></li>
        <li><a href="http://www.yahoo.com">Yahoo!</a></li>
    </ul>
</body>
</html>
```

Этот код создает элемент с элементом <ul> в качестве пользовательского контекстного меню, которое сначала скрыто. Чтобы этот пример заработал, нужен следующий JavaScript-код:

Листинг ContextMenuEventExample01.htm


```
EventUtil.addHandler(window, "load", function(event){
    var div = document.getElementById("myDiv");

    EventUtil.addHandler(div, "contextmenu", function(event){
        event = EventUtil.getEvent(event);
        EventUtil.preventDefault(event);

        var menu = document.getElementById("myMenu");
        menu.style.left = event.clientX + "px";
        menu.style.top = event.clientY + "px";
        menu.style.visibility = "visible";
    });

    EventUtil.addHandler(document, "click", function(event){
        document.getElementById("myMenu").style.visibility = "hidden";
    });
});
```

Здесь элементу назначается обработчик события contextmenu. Сначала он отменяет действие, предлагаемое по умолчанию, блокируя вывод контекстного меню браузера, а затем помещает элемент <ul> в позиции, заданной свойствами clientX и clientY объекта event. После этого для вывода меню его свойству visibility присваивается значение "visible". Кроме того, объекту document назначается обработчик события click, который скрывает меню при щелчке (так работают системные контекстные меню).

Хотя этот пример очень прост, подобный код лежит в основе всех пользовательских контекстных меню на веб-сайтах. Применив к меню CSS-стили, можно добиться еще лучшего результата.

Событие `contextmenu` поддерживается в Internet Explorer, Firefox, Safari, Chrome и Opera 11+.

Событие beforeunload

Событие `beforeunload` генерируется для объекта `window` перед началом выгрузки страницы из браузера, чтобы можно было отменить выгрузку и продолжить работу со страницей. Просто отменить это событие нельзя, потому что в результате пользователь будет заблокирован на странице. Вместо этого в обработчике события следует вывести сообщение, предлагающее пользователю закрыть страницу или остаться на ней (рис. 13.9).

Рис. 13.9

Чтобы показать такое диалоговое окно, назначьте свойству `event.returnValue` строку, которую нужно вывести в окне (для Internet Explorer и Firefox), и возвратите ее как значение функции (для Safari и Chrome):

Листинг BeforeUnloadEventExample01.htm


```
EventUtil.addHandler(window, "beforeunload", function(event){
    event = EventUtil.getEvent(event);
    var message = "I'm really going to miss you if you go.";
    event.returnValue = message;
    return message;
});
```

Событие `beforeunload` поддерживается в браузерах Internet Explorer, Firefox, Safari и Chrome, которые при его возникновении запрашивают подтверждение закрытия страницы. Opera до версии 11 включительно не поддерживает событие `beforeunload`.

Событие DOMContentLoaded

Событие `load` объекта `window` генерируется при завершении загрузки страницы, на что может потребоваться некоторое время, если страница содержит много внешних ресурсов. Событие `DOMContentLoaded` генерируется после завершения формирования

DOM-дерева независимо от изображений, JavaScript- и CSS-файлов, а также других подобных ресурсов. В сравнении с событием `load` событие `DOMContentLoaded` возникает на более раннем этапе загрузки страницы, позволяя пользователям быстрее приступить к работе с ней.

Обработчик события `DOMContentLoaded` можно подключить к объекту `document` или `window` (целевым элементом события на самом деле является `document`, хотя оно всплывает к `window`). Вот пример:

Листинг DOMContentLoadedEventExample01.htm

```
EventUtil.addHandler(document, "DOMContentLoaded", function(event){
    alert("Content loaded");
});
```

Объект `event` не предоставляет дополнительных сведений о событии `DOMContentLoaded`.

Событие `DOMContentLoaded` поддерживается в Internet Explorer 9+, Firefox, Chrome, Safari 3.1+, Opera 9+ и обычно используется для подключения обработчиков событий или выполнения других манипуляций с DOM-структурой. Оно всегда генерируется до события `load`.

В браузерах, которые не поддерживают событие `DOMContentLoaded`, можно попытаться имитировать его, задав во время загрузки страницы тайм-аут с нулевой задержкой:

```
setTimeout(function(){
    // подключите здесь обработчики событий
}, 0);
```

По сути, этот код заставляет вызвать функцию сразу же по завершении текущего JavaScript-процесса, который загружает и конструирует страницу. Совпадет ли это по времени с событием `DOMContentLoaded`, зависит от браузера и другого кода на странице. Чтобы тайм-аут сработал правильно, он должен быть первым на странице, но даже это не гарантирует, что он всегда будет предшествовать событию `load`.

Событие readystatechange

Событие `readystatechange` было впервые определено в Internet Explorer для нескольких объектов DOM-документа. Оно предоставляет сведения о состоянии загрузки документа или элемента, но часто работает с ошибками. У каждого объекта, поддерживающего событие `readystatechange`, есть свойство `readyState`, которое может иметь одно из пяти строковых значений:

- `uninitialized` — объект существует, но не инициализирован;
- `loading` — объект загружает данные;
- `loaded` — объект завершил загрузку данных;
- `interactive` — объект загружен не полностью, но с ним можно взаимодействовать;
- `complete` — объект полностью загружен.

Казалось бы, все просто, но проблема в том, что не все объекты проходят все эти этапы. В документации сказано, что объекты могут пропускать неважные этапы, но не указано, какие этапы считаются важными для тех или иных объектов. Это означает, что событие `readystatechange` часто генерируется менее четырех раз и что последовательности значений `readyState` могут быть разными.

Для объекта `document`, у которого свойство `readyState` изменяет значение на `"interactive"`, событие `readystatechange` генерируется примерно в то же время, что и событие `DOMContentLoaded`. Интерактивный этап наступает, когда все DOM-дерево загружено и с ним можно безопасно взаимодействовать. Доступность изображений и других внешних ресурсов в это время не гарантируется. Событие `readystatechange` можно обработать так:

```
EventUtil.addHandler(document, "readystatechange", function(event){
    if (document.readyState == "interactive"){
        alert("Content loaded");
    }
});
```

Объект `event` не предоставляет дополнительных сведений об этом событии, а его свойство `target` не задано.

Порядок генерирования этого события относительно события `load` не определен. На страницах с многочисленными или крупными внешними ресурсами интерактивный этап наступает задолго до события `load`, а если внешних ресурсов мало или они не слишком объемны, событие `load` может возникнуть раньше, чем `readystatechange`.

Более того, интерактивный этап может наступить до или после полной загрузки страницы (значение `"complete"`). Чем больше внешних ресурсов содержит страница, тем выше вероятность того, что интерактивный этап будет достигнут до ее полной загрузки. Следовательно, чтобы пользователи могли как можно раньше приступить к работе со страницей, нужно проверить оба условия:

```
EventUtil.addHandler(document, "readystatechange", function(event){
    if (document.readyState == "interactive" ||
            document.readyState == "complete"){
        EventUtil.removeHandler(document, "readystatechange",
                                arguments.callee);
        alert("Content loaded");
    }
});
```

При вызове этого обработчика мы прежде всего хотим убедиться, что документ уже загружен или достиг интерактивного этапа. Если это так, мы удаляем обработчик события, чтобы исключить его запуск для других этапов. Поскольку обработчик является анонимной функцией, для его удаления используется свойство `arguments.callee`. В конце выводится оповещение о том, что контент загружен. Такой код позволяет максимально приблизиться по времени к событию `DOMContentLoaded`.

Событие `readystatechange` генерируется для документа в Internet Explorer, Firefox 4+ и Opera.

Хотя с помощью события readystatechange можно имитировать событие DOMContentLoaded, они не идентичны. Порядок генерирования событий load и readystatechange зависит от страницы.

Событие `readystatechange` генерируется также для элементов `<script>` (в Internet Explorer и Opera) и `<link>` (только в Internet Explorer), что позволяет получать уведомления о загрузке внешних JavaScript- и CSS-файлов. Это полезно потому, что динамически создаваемые элементы начинают загружать внешние ресурсы только после добавления на страницу. К сожалению, здесь также не все просто, потому что когда ресурс становится доступен, свойство `readyState` может иметь значение `"loaded"` или `"complete"`. Иногда оно останавливается на значении `"loaded"` и не принимает значение `"complete"`, а иногда этап `"loaded"` пропускается и сразу устанавливается значение `"complete"`. Из-за этого в обработчике необходимо использовать конструкцию из предыдущего фрагмента. Например, следующий код проверяет, загружен ли внешний JavaScript-файл:

Листинг ReadyStateChangeEventExample01.htm


```
EventUtil.addHandler(window, "load", function(){
    var script = document.createElement("script");

    EventUtil.addHandler(script, "readystatechange", function(event){
        event = EventUtil.getEvent(event);
        var target = EventUtil.getTarget(event);

        if (target.readyState == "loaded" ||
                target.readyState == "complete"){
            EventUtil.removeHandler(target, "readystatechange",
                                    arguments.callee);
            alert("Loaded");
        }
    });
    script.src = "example.js";
    document.body.appendChild(script);
});
```

Здесь мы назначаем обработчик события новому узлу `<script>`. Целевым элементом события является сам узел, так что при возникновении события `readystatechange` нужно проверить свойство `readyState` объекта `target`. Если оно имеет значение `"loaded"` или `"complete"`, мы удаляем обработчик события, чтобы он не мог быть выполнен дважды, и выводим оповещение. После этого можно вызывать функции, загруженные из внешнего файла.

Аналогичный код можно использовать для загрузки CSS-файлов с помощью элемента `<link>`:

Листинг ReadyStateChangeEventExample02.htm


```
EventUtil.addHandler(window, "load", function(){
    var link = document.createElement("link");
    link.type = "text/css";
    link.rel= "stylesheet";

    EventUtil.addHandler(script, "readystatechange", function(event){
        event = EventUtil.getEvent(event);
        var target = EventUtil.getTarget(event);

        if (target.readyState == "loaded" ||
                target.readyState == "complete"){
            EventUtil.removeHandler(target, "readystatechange",
                                    arguments.callee);
            alert("CSS-файл загружен");
        }
    });

    link.href = "example.css";
    document.getElementsByTagName("head")[0].appendChild(link);
});
```

Как и прежде, здесь важно проверить оба значения `readyState` и удалить обработчик события при его первом вызове.

События pageshow и pagehide

Firefox и Opera поддерживают так называемый *кэш состояния страниц* (back-forward cache, bfcache), который ускоряет смену страниц при щелчках на кнопках Назад и Вперед. Этот кэш хранит не только данные страницы, но и ее DOM- и JavaScript-состояния — по сути, она полностью удерживается в памяти. Если страница находится в этом кэше, событие `load` при переходе к ней не генерируется. Обычно это не проблема, потому что доступно все состояние страницы, но разработчики Firefox все же решили добавить некоторые события, предоставляющие больший контроль над кэшем состояния страниц.

Первое из них, `pageshow`, генерируется при выводе страницы на экран из кэша состояния страниц или иным образом. Для страницы, загружаемой из Интернета, событие `pageshow` генерируется после события `load`, а для страницы из кэша — сразу после полного восстановления ее состояния. Хотя целевым элементом этого события является объект `document`, обрабатывать его следует у объекта `window`, например:

```
(function(){
    var showCount = 0;

    EventUtil.addHandler(window, "load", function(){
        alert("Load fired");
    });

    EventUtil.addHandler(window, "pageshow", function(){
        showCount++;
```

```
        alert("Show has been fired " + showCount + " times.");
    });
})();
```

В этом примере используется закрытая область видимости, чтобы переменная showCount не была доступна глобально. Если страница загружается впервые, переменная showCount имеет значение 0, которое затем увеличивается при каждом событии pageshow. Каждый раз вы покидаете страницу с этим кодом и возвращаетесь к ней с помощью кнопки Назад, значение showCount увеличивается, потому что оно кэшируется в памяти вместе с остальным состоянием страницы. Если щелкнуть в браузере на кнопке обновления страницы, значение showCount обнулится, потому что страница будет полностью перезагружена.

Кроме обычных свойств, объект event события pageshow содержит свойство persisted, которое равно true, если страница сохранена в кэше состояния страниц, и false в противном случае, например:

Листинг PageShowEventExample01.htm


```
(function(){
    var showCount = 0;

    EventUtil.addHandler(window, "load", function(){
        alert("Load fired");
    });

    EventUtil.addHandler(window, "pageshow", function(){
        showCount++;
        alert("Show has been fired " + showCount +
              " times. Persisted? " + event.persisted);
    });
})();
```

Свойство persisted можно использовать для выбора того или иного действия в зависимости от того, кэширована ли страница.

Событие pagehide возникает непосредственно перед событием unload при выгрузке страницы из браузера. Как и событие pageshow, оно генерируется для объекта document, но его обработчик следует подключать к объекту window. Его объект event также содержит свойство persisted, хотя используется оно немного иначе, например:

Листинг PageShowEventExample01.htm


```
EventUtil.addHandler(window, "pagehide", function(event){
    alert("Hiding. Persisted? " + event.persisted);
});
```

При обработке события pagehide можно использовать свойство persisted для изменения логики сценария. У события pageshow свойство persisted равно true, если страница была загружена из кэша состояния страниц, а у события pagehide — если

страница будет сохранена в кэше после выгрузки. Таким образом, при первом событии pageshow свойство persisted всегда равно false, а при первом событии pagehide оно всегда равно true (конечно, при условии, что страница может быть сохранена в кэше).

События pageshow и pagehide поддерживаются в Firefox, Safari 5+, Chrome и Opera, но не в Internet Explorer до версии 9 включительно.

Страницы с обработчиком события unload автоматически исключаются из кэша состояния страниц, даже если этот обработчик пуст. Дело в том, что событие unload обычно используется для отмены действий, выполненных в обработчике события load, поэтому пропуск события load при следующем выводе страницы на экран может привести к сбою.

Событие hashchange

В HTML5 представлено событие hashchange, уведомляющее об изменении хэша URL-адреса (часть после знака #). Разработчики часто используют хэш URL-адреса для хранения данных состояния или навигации в Ajax-приложениях.

Обработчик события hashchange подключается к объекту window и вызывается при любом изменении хэша URL-адреса. Объект event этого события имеет дополнительные свойства oldURL и newURL, которые содержат полный URL-адрес (включая хэш) до и после изменения, например:

Листинг HashChangeEventExample01.htm

```
EventUtil.addHandler(window, "hashchange", function(event){
    alert("Old URL: " + event.oldURL + "\nNew URL: " + event.newURL);
});
```

Событие hashchange поддерживается в браузерах Internet Explorer 8+, Firefox 3.6+, Safari 5+, Chrome и Opera 10.6+, но из них только Firefox 6+, Chrome и Opera поддерживают свойства oldURL и newURL. По этой причине для определения текущего хэша лучше использовать объект location:

```
EventUtil.addHandler(window, "hashchange", function(event){
    alert("Current hash: " + location.hash);
});
```

Узнать, поддерживается ли событие hashchange, можно следующим образом:

```
var isSupported = ("onhashchange" in window);    // код с дефектом
```

В Internet Explorer 8 этот код ошибочно возвращает true даже в режиме документа Internet Explorer 7, поэтому лучше использовать следующий вариант:

```
var isSupported = ("onhashchange" in window) && (document.documentMode ===
                  undefined || document.documentMode > 7);
```

События устройств

С появлением смартфонов и планшетных компьютеров сфера применения браузеров значительно расширилась. Чтобы разработчики могли узнать, как используется то или иное устройство, были определены новые события — так называемые *события устройств* (device events). В 2011 году консорциум W3C приступил к работе над новой спецификацией этих событий под названием DeviceOrientation Event (http://dev.w3.org/geo/api/spec-source-orientation.html), призванной охватить постоянно растущее количество устройств. В этом разделе описываются как сам API из этого проекта спецификации, так и фирменные события.

Событие orientationchange

Событие `orientationchange` было реализовано в мобильной версии Safari, чтобы разработчики могли определять, когда пользователь изменяет альбомную ориентацию экрана на книжную и наоборот. В мобильной версии Safari доступно свойство `window.orientation`, которое может иметь одно из трех значений: 0 для книжного режима, 90 для альбомной ориентации с поворотом влево (кнопка Home находится справа) и –90 для альбомной ориентации с поворотом вправо (кнопка Home находится слева). В документации также упоминается значение 180, которое указывает, что устройство перевернуто, но это значение пока не поддерживается. Разные варианты ориентации устройства показаны на рис. 13.10.

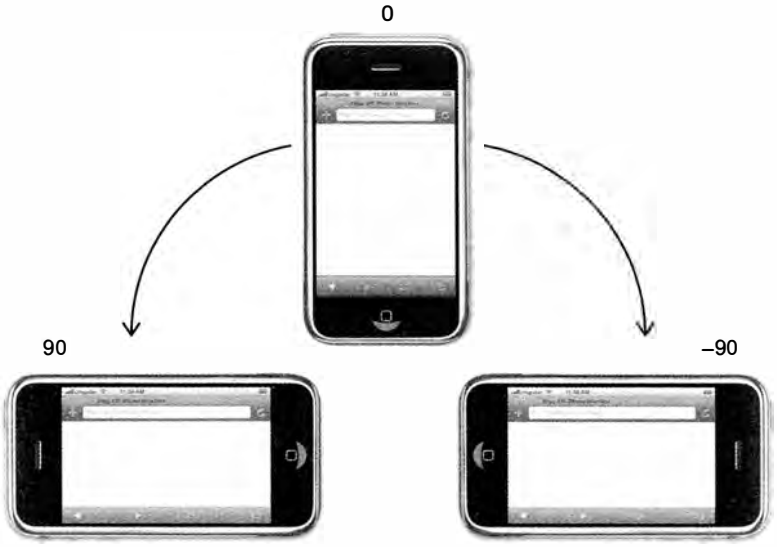

Рис. 13.10

При изменении ориентации генерируется событие `orientationchange`. Его объект `event` не содержит никакой полезной информации, поскольку все, что нужно знать,

это значение `window.orientation`. Обычно это событие используется следующим образом:

Листинг OrientationChangeEventExample01.htm


```
EventUtil.addHandler(window, "load", function(event){
    var div = document.getElementById("myDiv");
    div.innerHTML = "Current orientation is " + window.orientation;

    EventUtil.addHandler(window, "orientationchange", function(event){
        div.innerHTML = "Current orientation is: " + window.orientation;
    });
});
```

Этот код сначала выводит значение первоначальной ориентации в обработчике события `load`, а затем назначает обработчик события `orientationchange`, при возникновении которого на странице выводится новое значение ориентации.

Событие `orientationchange` и свойство `window.orientation` поддерживают все устройства с системой iOS.

Событие orientationchange относится к объекту window, поэтому для его обработки можно также добавить атрибут onorientationchange к элементу <body>.

Событие MozOrientation

В Firefox 3.6 для определения ориентации устройства было добавлено событие `MozOrientation` (префикс `Moz` указывает, что это фирменное событие, а не стандартное). Оно периодически генерируется, когда акселерометр регистрирует изменения ориентации устройства. Событие `MozOrientation` отличается от события `orientationchange` в iOS, которое представляет движение только в одной плоскости. Событие `MozOrientation` генерируется для объекта `window` и может быть обработано следующим образом:

```
EventUtil.addHandler(window, "MozOrientation", function(event){
    // реакция на событие
});
```

Объект `event` имеет три свойства с данными акселерометра: `x`, `y` и `z`. Каждое из них соответствует отдельной оси и содержит число в интервале от 1 и –1. В покое значения `x` и `y` равны 0, а `z` равно 1 (устройство расположено вертикально). При наклоне устройства вправо значение `x` уменьшается, а при наклоне влево увеличивается. Значение `y` уменьшается при наклоне устройства от себя и наоборот. Значение `z` представляет ускорение по вертикали: оно равно 1 в покое и уменьшается при перемещении устройства (без гравитации оно было бы равно 0). Вот простой пример, в котором на экран выводятся все три значения:

Листинг MozOrientationEventExample01.htm

```
EventUtil.addHandler(window, "MozOrientation", function(event){
    var output = document.getElementById("output");
    output.innerHTML = "X=" + event.x + ", Y=" + event.y + ", Z=" +
                       event.z + "<br>";
});
```

Событие `MozOrientation` поддерживают только устройства с акселерометрами, в том числе ноутбуки Macbook и Lenovo Thinkpad, а также устройства с системами Windows Mobile и Android. Имейте в виду, что это экспериментальное событие, которое, вероятно, будет заменено другим событием.

Событие deviceorientation

Событие `deviceorientation` определено в спецификации DeviceOrientation Event и похоже на событие `MozOrientation`. Оно тоже генерируется для объекта `window` при изменении показаний акселерометра и имеет те же ограничения. Помните, что оно предназначено для описания ориентации устройства в пространстве, а не его движения.

Устройство располагается в трехмерном пространстве с осями x, y и z. Ось x пересекает устройство слева направо, ось y — снизу вверх, а ось z — сзади вперед (рис. 13.11). Когда устройство покоится на горизонтальной поверхности, все три параметра ориентации равны нулю.

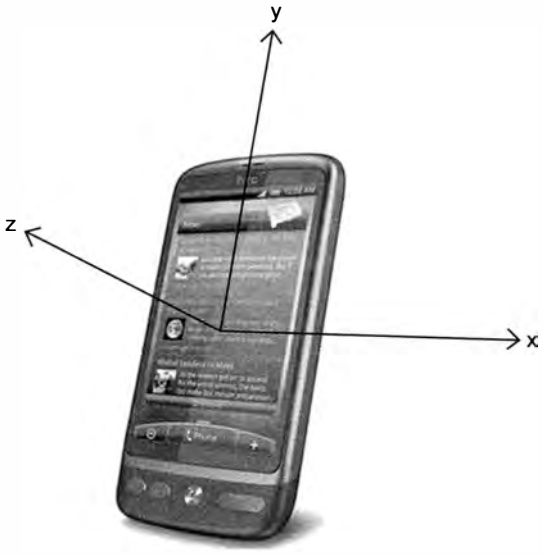

Рис. 13.11

Событие `deviceorientation` возвращает сведения о том, насколько углы наклона каждой оси отличаются от состояния покоя. В объекте `event` содержит пять свойств:

- `alpha` — угол поворота оси y вокруг оси *z* в градусах (число с плавающей точкой в интервале от 0 до 360);
- `beta` — угол поворота оси *z* вокруг оси *x* в градусах (число с плавающей точкой в интервале от –180 до 180);
- `gamma` — угол поворота оси *z* вокруг оси *y* в градусах (число с плавающей точкой в интервале от –90 до 90);
- `absolute` — логическое значение, указывающее, возвращает ли устройство абсолютные значения;
- `compassCalibrated` — логическое значение, указывающее, правильно ли откалиброван компас устройства.

На рис. 13.12 показано, как вычисляются значения `alpha`, `beta` и `gamma`.

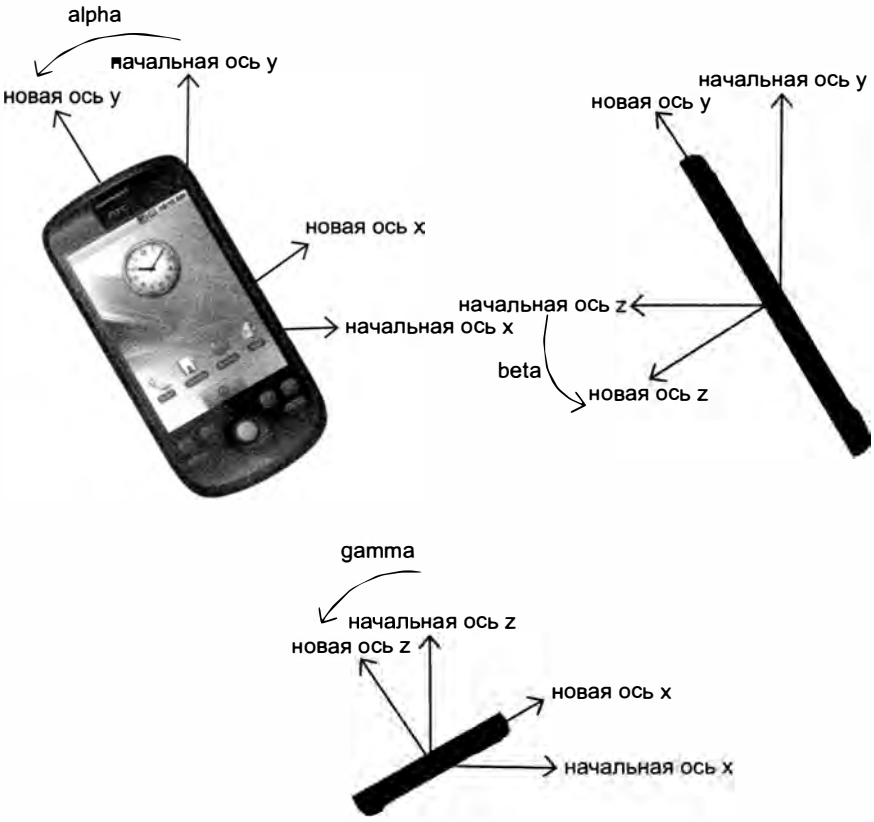

Рис. 13.12

В следующем примере на экран просто выводятся значения свойств `alpha`, `beta` и `gamma`:

Листинг DeviceOrientationEventExample01.htm


```
EventUtil.addHandler(window, "deviceorientation", function(event){
    var output = document.getElementById("output");
    output.innerHTML = "alpha =" + event.alpha + ", beta =" + event.beta +
                       ", gamma =" + event.gamma + "<br>";
});
```

Эти данные можно использовать для переупорядочения или перемещения элементов на экране при изменении ориентации устройства. Например, следующий код поворачивает элемент:

Листинг DeviceOrientationEventExample01.htm

```
EventUtil.addHandler(window, "deviceorientation", function(event){
    var arrow = document.getElementById("arrow");
    arrow.style.webkitTransform = "rotate(" + Math.round(event.alpha) +
                                  "deg)";
});
```

Этот пример работает только в мобильных браузерах WebKit, потому что в нем используется фирменное свойство `webkitTransform` (временная версия стандартного CSS-свойства `transform`). В этом коде элемент `"arrow"` поворачивается в соответствии со значением `event.alpha`, что делает его похожим на компас. Для плавного поворота элемента выполняется CSS3-преобразование с округленным значением свойства.

В 2011 году событие `deviceorientation` было доступно только в Safari для iOS 4.2+, Chrome и WebKit для Android.

Событие devicemotion

Спецификация DeviceOrientation Event включает также событие `devicemotion`, которое описывает перемещение устройства, а не только изменение его ориентации. Например, с его помощью можно определить, что устройство падает или используется во время ходьбы.

Объект `event` этого события содержит следующие дополнительные свойства:

- `acceleration` — объект со свойствами `x`, `y` и `z`, которые показывают ускорение в каждом направлении без учета гравитации;
- `accelerationIncludingGravity` — объект со свойствами `x`, `y` и `z`, которые показывают ускорение в каждом направлении с учетом естественной гравитации по оси *z*;
- `interval` — интервал между событиями `devicemotion` в миллисекундах (это значение должно быть постоянным);
- `rotationRate` — объект со свойствами `alpha`, `beta` и `gamma`, показывающими ориентацию устройства.

Если определить значение `acceleration`, `accelerationIncludingGravity` или `rotationRate` невозможно, оно считается равным `null`, поэтому перед использованием любого из этих свойств нужно проверять, не равно ли оно `null`, например:

Листинг DeviceMotionEventExample01.htm


```
EventUtil.addHandler(window, "devicemotion", function(event){
    var output = document.getElementById("output");
    if (event.rotationRate !== null){
        output.innerHTML += "alpha =" + event.rotationRate.alpha +
                            ", beta =" + event.rotationRate.beta +
                            ", gamma =" + event.rotationRate.gamma;
    }
});
```

Как и `deviceorientation`, событие `devicemotion` поддерживается только в Safari для iOS 4.2+, Chrome и WebKit для Android.

События касаний и жестов

Поскольку устройства с iOS используются без мыши и клавиатуры, обычных событий мыши и клавиатуры было недостаточно для создания полноценных интерактивных веб-страниц для мобильного браузера Safari. В связи с этим разработчики Safari для iOS добавили в него несколько фирменных специализированных событий. С выходом WebKit для Android многие такие события стали стандартом де-факто, что подтолкнуло консорциум W3C к разработке спецификации Touch Events (https://dvcs.w3.org/hg/webevents/raw-file/tip/touchevents.html). События, описанные в этом разделе, работают только на устройствах с поддержкой сенсорного ввода.

События касаний

Смартфон iPhone 3G с системой iOS 2.0 поставлялся с новой версией Safari, поддерживающей несколько новых событий, связанных с сенсорным вводом. Позднее такие же события были реализованы в браузере для Android. *События касаний* (touch events) генерируются, когда пользователь касается экрана устройства пальцем, перемещает палец и отрывает его от экрана:

- `touchstart` — генерируется при касании экрана пальцем, даже если пользователь уже касается экрана другим пальцем;
- `touchmove` — постоянно генерируется при перемещении пальца по экрану (вызов метода `preventDefault()` во время этого события предотвращает прокрутку);
- `touchend` — генерируется, когда пользователь отрывает палец от экрана;
- `touchcancel` — генерируется, когда система прекращает отслеживать касание (из документации не понятно, когда может произойти это событие).

Каждое из этих событий всплывает и может быть отменено. Хотя события касаний не входят в спецификацию DOM, они реализованы в соответствии с DOM. Так,

объект event каждого события касания содержит свойства, общие для событий мыши: bubbles, cancelable, view, clientX, clientY, screenX, screenY, detail, altKey, shiftKey, ctrlKey и metaKey.

В дополнение к этим общим DOM-свойствам события касания имеют три свойства для отслеживания касаний:

- touches — массив объектов Touch, которые представляют касания, отслеживаемые в текущий момент;
- targetTouches — массив объектов Touch, специфичных для целевого элемента события;
- changedTouches — массив объектов Touch, измененных в результате последнего действия пользователя.

Каждый объект Touch, в свою очередь, имеет следующие свойства:

- clientX — координата x целевого элемента касания в области просмотра;
- clientY — координата y целевого элемента касания в области просмотра;
- identifier — уникальный идентификатор касания;
- pageX — координата x целевого элемента касания на странице;
- pageY — координата y целевого элемента касания на странице;
- screenX — координата x целевого элемента касания на экране;
- screenY — координата y целевого элемента касания на экране;
- target — целевой элемент касания (DOM-узел).

С помощью этих свойств можно отслеживать касание экрана, например:

Листинг TouchEventsExample01.htm


```
function handleTouchEvent(event){

    // работает только для одного касания
    if (event.touches.length == 1)

        var output = document.getElementById("output");
        switch(event.type){
            case "touchstart":
                output.innerHTML = "Touch started (" +
                                   event.touches[0].clientX + "," +
                                   event.touches[0].clientY + ")";
                break;
            case "touchend":
                output.innerHTML += "<br>Touch ended (" +
                                    event.changedTouches[0].clientX + "," +
                                    event.changedTouches[0].clientY + ")";
                break;
            case "touchmove":
                event.preventDefault();    // предотвращение прокрутки
```

```
                output.innerHTML += "<br>Touch moved (" +
                                    event.changedTouches[0].clientX + "," +
                                    event.changedTouches[0].clientY + ")";
                break;
        }
    }
}

EventUtil.addHandler(document, "touchstart", handleTouchEvent);
EventUtil.addHandler(document, "touchend", handleTouchEvent);
EventUtil.addHandler(document, "touchmove", handleTouchEvent);
```

Ради простоты примера здесь мы отслеживаем только одно активное касание. Когда происходит событие `touchstart`, обработчик выводит координаты касания в элементе `<div>`. При событии `touchmove` мы отменяем действие, предлагаемое по умолчанию, чтобы страница не прокручивалась (перемещение пальца по экрану обычно сопровождается прокруткой), а затем выводим информацию об измененном касании. Наконец, при событии `touchend` выводятся финальные координаты касания. При обработке события `touchend` коллекция `touches` пуста, так как активного касания больше нет, поэтому вместо нее нужно использовать коллекцию `changedTouches`.

Эти события генерируются для всех элементов документа, что позволяет работать с разными частями страницы по отдельности. Если пользователь касается элемента и тут же отпускает его без перемещения пальца, события генерируются в следующем порядке:

1. `touchstart`.
2. `mouseover`.
3. `mousemove` (один раз).
4. `mousedown`.
5. `mouseup`.
6. `click`.
7. `touchend`.

События касаний поддерживаются в Safari для iOS, WebKit для Android, Dolfin для bada, BlackBerry WebKit для OS6+, Opera Mobile 10.1+ и браузере Phantom для операционных систем производства LG. Множественные касания поддерживаются только в Safari для iOS. Кроме того, события касаний реализованы в браузерах Firefox 6+ и Chrome для настольных компьютеров.

События жестов

В Safari для iOS 2.0 также были представлены события жестов. *Жест* (gesture) имеет место, если экрана касаются два пальца, и обычно вызывает изменение масштаба или поворот элемента. Событий жестов три:

- `gesturestart` — генерируется при касании экрана, если пользователь уже касается его другим пальцем;
- `gesturechange` — генерируется при изменении положения любого из пальцев, касающихся экрана;
- `gestureend` — генерируется, когда пользователь отрывает от экрана один из пальцев.

Эти события генерируются, только если пользователь касается элемента двумя пальцами. Чтобы генерировались события жестов, оба пальца должны находиться в пределах целевого элемента, которому назначены обработчики. Поскольку эти события всплывают, можно также обрабатывать все жесты на уровне документа. При использовании этого подхода целевым элементом события будет элемент, в пределах которого находятся оба пальца.

События касаний и жестов связаны. При касании экрана первым пальцем возникает событие `touchstart`. При касании экрана вторым пальцем сначала генерируется событие `gesturestart`, а за ним следует событие `touchstart` для этого пальца. При перемещении одного или обоих пальцев генерируется событие `gesturechange`, а когда пользователь отрывает один из пальцев от экрана, возникает событие `gestureend`, за которым следует событие `touchend` для этого пальца.

Как и при касаниях, при каждом жесте объект `event` содержит все стандартные свойства мыши: `bubbles`, `cancelable`, `view`, `clientX`, `clientY`, `screenX`, `screenY`, `detail`, `altKey`, `shiftKey`, `ctrlKey` и `metaKey`. Кроме того, у него есть дополнительные свойства `rotation` и `scale`. Свойство `rotation` указывает угол поворота пальцев в градусах, при этом положительные значения соответствует повороту по часовой стрелке, а отрицательные — против (начальное значение равно 0). Свойство `scale` указывает изменение расстояния между пальцами. Оно начинается с единицы и увеличивается или уменьшается при раздвигании или сдвигании пальцев соответственно.

Эти события можно использовать следующим образом:

Листинг GestureEventsExample01.htm


```
function handleGestureEvent(event){
    var output = document.getElementById("output");
    switch(event.type){
        case "gesturestart":
            output.innerHTML = "Gesture started (rotation=" +
                               event.rotation + ",scale=" +
                               event.scale + ")";
            break;
        case "gestureend":
            output.innerHTML += "<br>Gesture ended (rotation=" +
                                event.rotation + ",scale=" +
                                event.scale + ")";
            break;
        case "gesturechange":
            output.innerHTML += "<br>Gesture changed (rotation=" +
```

```
                                event.rotation + ",scale=" +
                                event.scale + ")";
            break;
    }
}

document.addEventListener("gesturestart", handleGestureEvent, false);
document.addEventListener("gestureend", handleGestureEvent, false);
document.addEventListener("gesturechange", handleGestureEvent, false);
```

Как и пример с событиями касаний, этот код просто подключает ко всем трем событиям один и тот же обработчик, в котором выводит сведения о каждом из них.

События касаний тоже возвращают свойства rotation и scale, но их значения изменяются, только если пользователь касается экрана двумя пальцами. Применять в подходящих ситуациях события жестов обычно проще, чем управлять всеми взаимодействиями с помощью событий касаний.

Память и быстродействие

События обеспечивают интерактивность современных веб-приложений, но многие разработчики злоупотребляют ими. В языках, таких как C#, используемых для создания GUI, можно без заметного падения производительности обрабатывать событие `click` каждой кнопки, однако в JavaScript от количества обработчиков событий на странице напрямую зависит ее общее быстродействие. Это объясняется несколькими причинами. Во-первых, каждая функция является объектом и занимает память; чем больше объектов в памяти, тем хуже быстродействие. Во-вторых, первоначальное назначение всех обработчиков событий в DOM задерживает момент, когда можно приступить к работе со страницей. Используя обработчики событий с умом, можно увеличить быстродействие страниц.

Делегирование событий

Проблему чрезмерного количества обработчиков событий можно решить за счет *делегирования событий* (event delegation). Этот прием, основанный на всплытии событий, предполагает, что все события определенного типа обрабатываются одним обработчиком. Например, событие `click` всплывает до уровня `document`, благодаря чему можно назначить один обработчик `onclick` всей странице, а не по одному для каждого элемента, поддерживающего щелчок. Рассмотрим следующий HTML-код:

Листинг EventDelegationExample01.htm


```
<ul id="myLinks">
    <li id="goSomewhere">Go somewhere</li>
    <li id="doSomething">Do something</li>
    <li id="sayHi">Say hi</li>
</ul>
```

При щелчке на каждом из трех элементов выполняется то или иное действие. В традиционном подходе мы просто подключили бы к ним три обработчика событий:

```
var item1 = document.getElementById("goSomewhere");
var item2 = document.getElementById("doSomething");
var item3 = document.getElementById("sayHi");

EventUtil.addHandler(item1, "click", function(event){
    location.href = "http://www.wrox.com";
});

EventUtil.addHandler(item2, "click", function(event){
    document.title = "I changed the document's title";
});

EventUtil.addHandler(item3, "click", function(event){
    alert("hi");
});
```

Если бы в сложном веб-приложении это было нужно сделать для всех элементов, поддерживающих щелчок, только подключение обработчиков заняло бы несколько десятков, а то и сотен строк кода. Делегирование событий решает эту проблему, позволяя подключить единственный обработчик к наивысшей точке в DOM-дереве, например:

Листинг EventDelegationExample01.htm


```
var list = document.getElementById("myLinks");

EventUtil.addHandler(list, "click", function(event){
    event = EventUtil.getEvent(event);
    var target = EventUtil.getTarget(event);

    switch(target.id){
        case "doSomething":
            document.title = "I changed the document's title";
            break;

        case "goSomewhere":
            location.href = "http://www.wrox.com";
            break;

        case "sayHi":
            alert("hi");
            break;
    }
});
```

Этот код подключает единственный обработчик события `click` к элементу `<ul>`, к которому всплывают события всех элементов списка. Целевым элементом события является элемент списка, на котором был выполнен щелчок, так что для выбора нужного действия можно использовать его свойство `id`. В сравнении с предыдущим

примером без делегирования событий этот код изначально менее требователен к ресурсам, потому что он получает единственный DOM-элемент и подключает только один обработчик события. Для пользователя ничего не меняется, но такой код требует гораздо меньше памяти. Потенциально он подходит для обработки любых событий кнопок (большинство событий мыши и клавиатуры).

Все события конкретного типа на странице можно также обрабатывать на уровне объекта `document`, если это оправданно. В сравнении с традиционным такой подход обеспечивает определенные преимущества.

- Обработчик события можно назначить объекту `document` в любой момент жизненного цикла страницы (дожидаться события `DOMContentLoaded` или `load` не требуется). Это означает, что щелчки на элементе будут правильно обрабатываться сразу после его визуализации.
- Для регистрации обработчиков событий требуется меньше времени и меньше обращений к DOM.
- Благодаря меньшему потреблению памяти повышается общее быстродействие страницы.

Для делегирования более всего подходят события `click`, `mousedown`, `mouseup`, `keydown`, `keyup` и `keypress`. События `mouseover` и `mouseout` всплывают, но их сложнее обработать правильно, потому что для этого часто требуется вычислить позицию элемента (событие `mouseout` генерируется при наведении указателя мыши на один из дочерних узлов элемента и при его перемещении за пределы элемента).

Удаление обработчиков событий

При назначении обработчиков событий элементам формируются связи между кодом, который выполняет браузер, и JavaScript-кодом, предназначенным для взаимодействия со страницей. Чем больше таких связей, тем медленнее работает страница. Один из способов решения этой проблемы — делегировать обработку событий для уменьшения количества связей. Другой способ — удалять обработчики событий, ставшие ненужными. Если просто оставлять их в памяти, веб-приложение будет тратить гораздо больше ресурсов, чем могло бы.

Проблема возникает на двух этапах жизненного цикла страницы. Первый имеет место при удалении элемента, к которому подключены обработчики событий. Иногда при этом используются методы `removeChild()` и `replaceChild()`, но чаще всего это происходит при замене фрагмента кода с помощью свойства `innerHTML`. Любые обработчики событий, назначенные элементу, который был удален с помощью свойства `innerHTML`, невозможно правильно удалить в ходе сборки мусора. Рассмотрим пример:

```
<div id="myDiv">
    <input type="button" value="Click Me" id="myBtn">
</div>
```

```
<script type="text/javascript">
    var btn = document.getElementById("myBtn");
    btn.onclick = function(){

        // какие-то действия

        document.getElementById("myDiv").innerHTML =
            "Processing...";        // Плохо!!!
    };
</script>
```

Здесь элемент `<div>` содержит кнопку, которая при щелчке заменяется сообщением. Этот прием предотвращает двойной щелчок на кнопке и очень часто используется на веб-сайтах. Проблема в том, что при удалении кнопки с помощью свойства `innerHTML` к ней остается подключенным обработчик события. В некоторых браузерах, особенно в Internet Explorer 8 и более ранних версий, ссылки на элемент и обработчик события обычно остаются в памяти. Таким образом, перед удалением элемента лучше вручную удалить подключенные к нему обработчики событий, например:

```
<div id="myDiv">
    <input type="button" value="Click Me" id="myBtn">
</div>
<script type="text/javascript">
    var btn = document.getElementById("myBtn");
    btn.onclick = function(){

        // какие-то действия

        btn.onclick = null;    // удаление обработчика события

        document.getElementById("myDiv").innerHTML = "Processing...";
    };
</script>
```

Этот код удаляет обработчик события кнопки, прежде чем задать свойство `innerHTML` элемента `<div>`. Это гарантирует, что память будет возвращена среде, и позволяет безопасно удалить кнопку из DOM-структуры.

Имейте в виду, что удаление кнопки в обработчике события отменяет всплытие события. Событие всплывает, только если его целевой элемент все еще находится в документе.

В этой ситуации может помочь делегирование событий. Если вам известно, что какой-то фрагмент страницы будет заменен с помощью свойства innerHTML, не подключайте обработчики к элементам в этом фрагменте, а обработайте их события на более высоком уровне.

Проблема обработчиков событий удаленных элементов возникает также при выгрузке страницы. В той или иной степени она присуща всем браузерам, но и здесь выделяется Internet Explorer 8 и более ранних версий. Если не удалить обработчики

событий перед выгрузкой страницы, они остаются в памяти, при этом количество объектов в памяти растет с каждым последующим циклом загрузки и выгрузки страницы (в результате перезагрузки или щелчках на кнопках Назад и Вперед).

Перед выгрузкой страницы имеет смысл удалить все обработчики событий в обработчике `onunload`. Делегирование событий полезно и здесь, потому что следить за обработчиками событий проще, если их меньше. Как правило, все, что делается в обработчике `onload`, следует отменять в обработчике `onunload`.

Помните, что если на странице обрабатывается событие unload, она не сохраняется в кэше состояния страниц. Если это важно, можно использовать событие unload для удаления обработчиков событий только в Internet Explorer.

Имитация событий

События предназначены для уведомления о важных моментах жизненного цикла веб-страницы и часто генерируются при взаимодействии с пользователем или вызове тех или иных функций браузера. Однако мало кому известно, что в JavaScript можно в любое время инициировать определенные события, которые ничем не отличаются от событий, генерируемых браузером, — они точно так же всплывают и вызывают назначенные им обработчики. Эта возможность очень полезна при тестировании веб-приложений. Способы имитации событий, определенные в спецификации DOM Level 3, поддерживаются в Internet Explorer 9, Opera, Firefox, Chrome и Safari. В Internet Explorer 8 и более ранних версий используется фирменный способ имитации событий.

Имитация DOM-событий

Вы можете в любой момент создать объект `event`, вызвав для объекта `document` метод `createEvent()`. Он принимает строку, которая указывает тип создаваемого события. В DOM Level 2 тип события указывался во множественном числе, но в DOM Level 3 используется единственное число. Перечислим возможные типы событий:

- `UIEvents` — универсальное событие пользовательского интерфейса, от которого наследуются события мыши и клавиатуры. В DOM Level 3 используется имя `UIEvent`.
- `MouseEvents` — универсальное событие мыши. В DOM Level 3 используется имя `MouseEvent`.
- `MutationEvents` — универсальное событие изменения DOM. В DOM Level 3 используется имя `MutationEvent`.
- `HTMLEvents` — универсальное HTML-событие. Эквивалентного события в спецификации DOM Level 3 нет, потому что в ней HTML-события распределены по другим группам.

События клавиатуры не определены в DOM Level 2 Events, но были представлены в DOM Level 3 Events. В настоящее время они поддерживаются только в Internet Explorer 9, а в других браузерах их можно имитировать иными способами.

Как только объект `event` создан, его нужно инициализировать сведениями о событии. Для этого используется специальный метод объекта `event`, зависящий от аргумента, который был передан в метод `createEvent()`.

Наконец, для имитации события нужно сгенерировать его. Это делается с помощью метода `dispatchEvent()`, который доступен для всех DOM-узлов, поддерживающих события. В качестве аргумента он принимает объект `event` генерируемого события. Как только этот метод вызван, событие всплывает и запускает обработчики.

Имитация событий мыши

Событие мыши можно имитировать, создав его объект и назначив ему нужные данные. Чтобы создать объект события мыши, требуется вызвать метод `createEvent()` с аргументом `"MouseEvents"`. После этого можно назначить данные события возвращенному объекту с помощью его метода `initMouseEvent()`. Этот метод принимает 15 перечисленных далее аргументов — по одному для каждого свойства, обычно имеющегося у события мыши.

- `type` (строка) — тип генерируемого события, например `"click"`.
- `bubbles` (логическое значение) — указывает, должно ли событие всплывать. Для правильной имитации события мыши этот аргумент должен быть равен `true`.
- `cancelable` (логическое значение) — указывает, можно ли отменить событие. Для правильной имитации события мыши этот аргумент должен быть равен `true`.
- `view` (AbstractView) — представление, связанное с событием. Им почти всегда является объект `document.defaultView`.
- `detail` (целое число) — дополнительные сведения о событии. Этот аргумент используется только обработчиками событий, но обычно имеет значение 0.
- `screenX` (целое число) — координата x события относительно экрана.
- `screenY` (целое число) — координата y события относительно экрана.
- `clientX` (целое число) — координата x события относительно области просмотра.
- `clientY` (целое число) — координата y события относительно области просмотра.
- `ctrlKey` (логическое значение) — указывает, нажата ли клавиша Ctrl. По умолчанию `false`.
- `altKey` (логическое значение) — указывает, нажата ли клавиша Alt. По умолчанию `false`.
- `shiftKey` (логическое значение) — указывает, нажата ли клавиша Shift. По умолчанию `false`.

- `metaKey` (логическое значение) — указывает, нажата ли клавиша Meta. По умолчанию `false`.
- `button` (целое число) — кнопка, которая была нажата. По умолчанию 0.
- `relatedTarget` (объект) — объект, связанный с событием. Используется только при имитации событий `mouseover` и `mouseout`.

Как видите, у событий мыши аргументы метода `initMouseEvent()` напрямую соответствуют свойствам объекта `event`. Для правильного генерирования события важны только первые четыре аргумента, потому что их использует браузер; остальные аргументы применяются только в обработчиках событий. Свойство `target` объекта `event` задается автоматически, когда он передается в метод `dispatchEvent()`. Например, следующий код имитирует щелчок на кнопке с аргументами, предлагаемыми по умолчанию:

Листинг SimulateDOMClickExample01.htm


```
var btn = document.getElementById("myBtn");

// создание объекта event
var event = document.createEvent("MouseEvents");

// инициализация объекта event
event.initMouseEvent("click", true, true, document.defaultView,
                     0, 0, 0, 0, 0, false, false, false, false, 0, null);

// генерирование события
btn.dispatchEvent(event);
```

В браузерах, соответствующих DOM, точно так же можно имитировать и все остальные события мыши, включая `dblclick`.

Имитация событий клавиатуры

События клавиатуры были определены в черновой версии DOM Level 2 Events, но не вошли в окончательную спецификацию, поэтому имитировать их непросто. События клавиатуры из черновой версии были реализованы в Firefox, но события, определенные в DOM Level 3, существенно отличаются от них.

В DOM Level 3 для создания события клавиатуры нужно передать строку `"KeyboardEvent"` в метод `createEvent()`. Он возвращает объект `event` с методом `initKeyboardEvent()`, который принимает следующие параметры:

- `type` (строка) — тип генерируемого события, например `"keydown"`.
- `bubbles` (логическое значение) — указывает, должно ли событие всплывать. Для правильной имитации события клавиатуры этот аргумент должен быть равен `true`.
- `cancelable` (логическое значение) — указывает, можно ли отменить событие. Для правильной имитации события клавиатуры этот аргумент должен быть равен `true`.

- `view` (AbstractView) — представление, связанное с событием. Им почти всегда является объект `document.defaultView`.
- `key` (строка) — строковый код нажатой клавиши.
- `location` (целое число) — расположение нажатой клавиши: 0 — клавиатура, предлагаемая по умолчанию; 1 — левая часть клавиатуры; 2 — правая часть клавиатуры; 3 — цифровая клавиатура; 4 — клавиатура мобильного устройства (виртуальная); 5 — джойстик.
- `modifiers` (строка) — список разделенных пробелами клавиш-модификаторов, таких как `"Shift"`.
- `repeat` (целое число) — количество нажатий клавиши подряд.

В DOM Level 3 Events событие `keypress` объявлено устаревшим, поэтому этим способом можно имитировать только события `keydown` и `keyup`, например:

Листинг SimulateDOMKeyEventExample01.htm


```
var textbox = document.getElementById("myTextbox"),
    event;

// создание объекта event в стиле DOM Level 3
if (document.implementation.hasFeature("KeyboardEvents", "3.0")){
    event = document.createEvent("KeyboardEvent");

    // инициализация объекта event
    event.initKeyboardEvent("keydown", true, true, document.defaultView,
                            "a", 0, "Shift", 0);
}

// генерирование события
textbox.dispatchEvent(event);
```

Этот пример имитирует нажатие клавиши A при нажатой клавише Shift. Прежде чем вызывать метод `document.createEvent("KeyboardEvent")`, всегда проверяйте, поддерживаются ли события клавиатуры DOM Level 3; если нет, браузер может возвратить нестандартный объект события клавиатуры.

В Firefox можно создать событие клавиатуры, передав строку `"KeyEvents"` в метод `createEvent()`. Он возвращает объект `event` с методом `initKeyEvent()`, который принимает десять аргументов:

- `type` (строка) — тип генерируемого события, например `"keydown"`.
- `bubbles` (логическое значение) — указывает, должно ли событие всплывать. Для правильной имитации события клавиатуры этот аргумент должен быть равен `true`.
- `cancelable` (логическое значение) — указывает, можно ли отменить событие. Для правильной имитации события клавиатуры этот аргумент должен быть равен `true`.

- `view` (AbstractView) — представление, связанное с событием. Им почти всегда является объект `document.defaultView`.
- `ctrlKey` (логическое значение) — указывает, нажата ли клавиша Ctrl. По умолчанию `false`.
- `altKey` (логическое значение) — указывает, нажата ли клавиша Alt. По умолчанию `false`.
- `shiftKey` (логическое значение) — указывает, нажата ли клавиша Shift. По умолчанию `false`.
- `metaKey` (логическое значение) — указывает, нажата ли клавиша Meta. По умолчанию `false`.
- `keyCode` (целое число) — код нажатой или отпущенной клавиши. Используется с событиями `keydown` и `keyup`. По умолчанию 0.
- `charCode` (целое число) — ASCII-код символа, сгенерированного нажатием клавиши. Используется с событием `keypress`. По умолчанию 0.

Чтобы сгенерировать событие клавиатуры, нужно передать настроенный объект `event` в метод `dispatchEvent()`, например:

Листинг SimulateFFKeyEventExample01.htm


```
// только для Firefox
var textbox = document.getElementById("myTextbox");

// создание объекта event
var event = document.createEvent("KeyEvents");

// инициализация объекта event
event.initKeyEvent("keydown", true, true, document.defaultView, false,
                   false, true, false, 65, 65);

// генерирование события
textbox.dispatchEvent(event);
```

Этот пример имитирует нажатие клавиши A при нажатой клавише Shift. Так же можно имитировать события `keyup` и `keypress`.

В других браузерах нужно создать универсальное событие и назначить ему данные, специфичные для клавиатуры, например:

```
var textbox = document.getElementById("myTextbox");

// создание объекта event
var event = document.createEvent("Events");

// инициализация объекта event
event.initEvent(type, bubbles, cancelable);
event.view = document.defaultView;
```

```
event.altKey = false;
event.ctrlKey = false;
event.shiftKey = false;
event.metaKey = false;
event.keyCode = 65;
event.charCode = 65;

// генерирование события
textbox.dispatchEvent(event);
```

Этот код создает универсальное событие, инициализирует его методом `initEvent()` и назначает ему данные о событии клавиатуры. Универсальное событие требуется вместо события пользовательского интерфейса потому, что последнее не позволяет добавить новые свойства к объекту `event` (это возможно только в Safari). Такой подход имитирует событие клавиатуры не совсем точно: оно генерируется, но никакой текст в текстовое поле не добавляется.

Имитация других событий

Чаще всего в браузерах имитируют события мыши и клавиатуры, но можно также имитировать события изменения DOM-структуры и HTML-события. Для имитации события изменения DOM-структуры используется метод `createEvent` с аргументом `"MutationEvents"`. Он возвращает объект события с методом `initMutationEvent()`, который принимает аргументы `type`, `bubbles`, `cancelable`, `relatedNode`, `prevValue`, `newValue`, `attrName` и `attrChange`. Вот пример имитации события изменения DOM-структуры:

```
var event = document.createEvent("MutationEvents");
event.initMutationEvent("DOMNodeInserted", true, false, someNode,
                        "","","",0);
target.dispatchEvent(event);
```

Этот код имитирует событие `DOMNodeInserted`. Все остальные события изменения DOM-структуры можно имитировать так же, используя подходящие аргументы.

Для имитации HTML-события нужно создать объект `event`, вызвав метод `createEvent("HTMLEvents")`, и инициализировать его с помощью метода `initEvent()`, например:

```
var event = document.createEvent("HTMLEvents");
event.initEvent("focus", true, false);
target.dispatchEvent(event);
```

Этот код генерирует событие `focus` для объекта `target`. Другие HTML-события можно имитировать также.

События изменения DOM-структуры и HTML-события редко применяются в браузерах, потому что пользы от них мало.

Пользовательские DOM-события

В DOM Level 3 определены также так называемые *пользовательские события* (custom events). Они изначально не генерируются, а предоставляются для того, чтобы разработчики могли создавать собственные события. Создать пользовательское событие можно, вызвав метод `createEvent("CustomEvent")`. Он возвращает объект `event` с методом `initCustomEvent()`, который принимает четыре аргумента:

- `type` (строка) — тип генерируемого события, например `"keydown"`;
- `bubbles` (логическое значение) — указывает, должно л и событие всплывать;
- `cancelable` (логическое значение) — указывает, можно ли отменить событие;
- `detail` (объект) — любое значение, которое присваивается свойству `detail` объекта `event`.

Затем созданное событие можно сгенерировать в DOM, как и любое другое, например:

Листинг SimulateDOMCustomEventExample01.htm


```
var div = document.getElementById("myDiv"),
    event;

EventUtil.addHandler(div, "myevent", function(event){
    alert("DIV: " + event.detail);
});
EventUtil.addHandler(document, "myevent", function(event){
    alert("DOCUMENT: " + event.detail);
});

if (document.implementation.hasFeature("CustomEvents", "3.0")){
    event = document.createEvent("CustomEvent");
    event.initCustomEvent("myevent", true, false, "Hello world!");
    div.dispatchEvent(event);
}
```

Этот код создает всплывающее событие `"myevent"` с простой строкой в качестве значения `event.detail`. Событие прослушивается у элемента `<div>` и на уровне документа, к которому оно всплывает благодаря второму аргументу метода `initCustomEvent()`.

Пользовательские DOM-события поддерживаются только в Internet Explorer 9+ и Firefox 6+.

Имитация событий в Internet Explorer

Общий алгоритм имитации событий в Internet Explorer 8 и более ранних версий такой же, что и в случае DOM-событий: вы создаете объект `event`, назначаете ему нужные значения и генерируете событие с его помощью. Конечно, в Internet Explorer это делается немного иначе.

Создать объект event можно с помощью метода createEventObject() объекта document. В отличие от DOM-метода, он не принимает аргументов и возвращает универсальный объект event, которому затем необходимо вручную назначить все нужные свойства (соответствующего метода нет). После этого нужно вызвать для целевого элемента события метод fireEvent(), передав ему имя обработчика события и настроенный объект event. При его вызове объекту event автоматически назначаются свойства srcElement и type, а все остальные свойства нужно задать вручную. Этот способ применяется для имитации всех событий, которые поддерживает Internet Explorer. Например, следующий код генерирует событие click для кнопки:

Листинг SimulateIEClickExample01.htm


```
var btn = document.getElementById("myBtn");

// создание объекта event
var event = document.createEventObject();

// инициализация объекта event
event.screenX = 100;
event.screenY = 0;
event.clientX = 0;
event.clientY = 0;
event.ctrlKey = false;
event.altKey = false;
event.shiftKey = false;
event.button = 0;

// генерирование события
btn.fireEvent("onclick", event);
```

Здесь мы создаем объект event, а затем инициализируем его некоторыми значениями. Свойства события задаются произвольным образом, при этом можно задать даже те свойства, которые по умолчанию не поддерживаются в Internet Explorer 8 и более ранних версий. Значения свойств не влияют на событие, потому что они используются только его обработчиками.

По такому же алгоритму можно сгенерировать событие keypress:

Листинг SimulateIEKeyEventExample01.htm


```
var textbox = document.getElementById("myTextbox");

// создание объекта event
var event = document.createEventObject();

// инициализация объекта event
event.altKey = false;
event.ctrlKey = false;
event.shiftKey = false;
event.keyCode = 65;

// генерирование события
textbox.fireEvent("onkeypress", event);
```

Поскольку объекты `event` событий мыши, клавиатуры и других событий не различаются, для генерирования любых событий можно использовать универсальный объект `event`. Как и при имитации DOM-события клавиатуры, в этом примере никакие знаки в текстовом поле не появляются, хотя обработчик события выполняется нормально.

Резюме

События — это основной механизм подключения JavaScript-кода к веб-страницам. События, используемые чаще всего, определены в спецификациях DOM Level 3 Events и HTML5, но во многих браузерах доступны также дополнительные «фирменные» события, позволяющие лучше контролировать взаимодействие с пользователями. Некоторые «фирменные» события предназначены для работы с конкретными устройствами, например событие `orientationchange` в мобильной версии Safari специфично для устройств с системой iOS.

При использовании событий следует помнить о том, что они потребляют ресурсы, и соблюдать перечисленные здесь принципы.

- Обработчиков событий на странице не должно быть слишком много, потому что они занимают память и замедляют отклик страницы на действия пользователя.
- Для уменьшения количества обработчиков на странице можно использовать механизм делегирования событий, в основе которого лежит всплытие событий.
- Рекомендуется удалять обработчики событий перед выгрузкой страницы.

С помощью JavaScript можно имитировать события в браузере. Спецификации DOM Level 2 и 3 позволяют с легкостью имитировать все определенные в них события. Имитировать события клавиатуры сложнее, но тоже возможно. В Internet Explorer 8 и более ранних версий используется «фирменный» способ имитации событий.

Обработка событий — одна из важнейших составляющих JavaScript-программирования. Каждый серьезный веб-программист должен понимать, как они работают и как влияют на быстродействие кода.

14 Работа с формами

- Общие сведения о формах
- Работа с текстовыми полями и проверка их содержимого
- Использование других элементов управления форм

Одной из первоначальных целей применения JavaScript было разделение ответственности за обработку форм между сервером и браузером. В отличие от веб-технологий и JavaScript, веб-формы с тех пор изменились незначительно. К сожалению, стандартного функционала веб-форм оказалось недостаточно для решения типичных проблем, и со временем разработчики начали расширять возможности форм с помощью JavaScript.

Общие сведения о формах

Веб-форма представляется HTML-элементом `<form>` в HTML и типом `HTMLFormElement` в JavaScript. Тип `HTMLFormElement` наследуется от типа `HTMLElement`, от которого получает все стандартные свойства HTML-элементов. К ним он добавляет следующие свойства и методы:

- `acceptCharset` — кодировки, которые может обрабатывать сервер (эквивалент HTML-атрибута `accept-charset`);
- `action` — URL-адрес для отправки запроса (эквивалент HTML-атрибута `action`);
- `elements` — коллекция `HTMLCollection`, содержащая все элементы управления формы;
- `enctype` — тип кодировки запроса (эквивалент HTML-атрибута `enctype`);

- `length` — количество элементов управления формы;
- `method` — тип отправляемого HTTP-запроса, обычно `"get"` или `"post"` (эквивалент HTML-атрибута `method`);
- `name` — имя формы (эквивалент HTML-атрибута `name`);
- `reset()` — сбрасывает все поля формы, восстанавливая значения, предлагаемые по умолчанию;
- `submit()` — отправляет данные формы;
- `target` — имя окна, используемого для отправки запроса и получения ответа (эквивалент HTML-атрибута `target`).

Ссылку на элемент `<form>` можно получить разными способами. Чаще всего ему назначают атрибут `id` подобно другим элементам, что позволяет использовать метод `getElementById()`, например:

```
var form = document.getElementById("form1");
```

Все формы на странице содержатся в коллекции `document.forms`. Каждая форма в ней доступна по числовому индексу и по имени:

```
var firstForm = document.forms[0];    // получение первой формы на странице
var myForm = document.forms["form2"]; // получение формы с именем "form2"
```

Старые браузеры и браузеры со строгой обратной совместимостью добавляют также каждую форму с именем к объекту `document` в качестве его свойства, например форма `"form2"` доступна как `document.form2`. Использовать этот формат доступа не рекомендуется, потому что он подвержен ошибкам и в будущем его поддержка браузерами может быть прекращена.

Имейте в виду, что формы могут иметь и идентификатор, и имя (`id` и `name`), которые могут быть разными.

Отправка данных формы

Данные формы отправляются серверу, когда пользователь щелкает на кнопке отправки или на графической кнопке. Кнопку отправки представляет элемент `<input>` или `<button>`, у которого атрибут `type` имеет значение `"submit"`, а графическую кнопку — элемент `<input>`, у которого атрибут `type` имеет значение `"image"`. Вот три примера кнопок, каждая из которых отправляет свою форму серверу:

```
<!-- обобщенная кнопка отправки -->
<input type="submit" value="Submit Form">

<!-- пользовательская кнопка отправки -->
<button type="submit">Submit Form</button>

<!-- графическая кнопка -->
<input type="image" src="graphic.gif">
```

Кнопка отправки может передать данные формы при нажатии клавиши Enter, если фокус принадлежит одному из элементов управления формы (исключение — поле `textarea`, для которого при нажатии клавиши Enter выполняется перевод строки). Данные формы без кнопки отправки при нажатии клавиши Enter не передаются.

Когда данные формы отправляется таким способом, непосредственно перед отправкой запроса серверу генерируется событие `submit`. В его обработчике можно проверить введенные в форме данные и при необходимости заблокировать их отправку, отменив для события действие, предлагаемое по умолчанию, например:

```
var form = document.getElementById("myForm");
EventUtil.addHandler(form, "submit", function(event){

    // получение объекта event
    event = EventUtil.getEvent(event);

    // отмена отправки данных формы
    EventUtil.preventDefault(event);
});
```

Для кроссбраузерной обработки события здесь используется объект `EventUtil` (см. предыдущую главу). Вызов его метода `preventDefault()` останавливает отправку данных формы. Как правило, так делают, если форма содержит недопустимые данные, которые не имеет смысла отправлять серверу.

Кроме того, в любой момент можно отправить данные формы программно, вызвав ее метод `submit()`, который не требует наличия кнопки отправки на форме, например:

```
var form = document.getElementById("myForm");

// отправка данных формы
form.submit();
```

При такой отправке данных формы событие `submit` не генерируется, так что не забудьте проверить данные до вызова метода `submit()`.

Одна из главных проблем с отправкой данных форм — многократная отправка. Иногда пользователи от нетерпения щелкают на кнопке отправки несколько раз, что в лучшем случае просто нагружает сервер (которому приходится обрабатывать несколько идентичных запросов), а в худшем может привести к убыткам и другим неприятностям (например, если пользователь оформит несколько заказов вместо одного). Эта проблема имеет два решения: можно отключать кнопку отправки после отправки данных формы или отменять последующие попытки отправки данных в обработчике события `submit`.

Сброс формы

Сбросить форму можно, нажав на кнопку сброса — элемент `<input>` или `<button>`, у которого атрибут `type` имеет значение `"reset"`, например:

```
<!-- обобщенная кнопка сброса -->
<input type="reset" value="Reset Form">

<!-- пользовательская кнопка сброса -->
<button type="reset">Reset Form</button>
```

Обе эти кнопки сбрасывают форму, при этом в ее полях восстанавливаются значения, которые имели место при начальной визуализации страницы. Если поле первоначально было пустым, оно снова становится пустым.

При сбросе формы с помощью кнопки сброса генерируется событие `reset`, позволяющее отменить сброс, например:

```
var form = document.getElementById("myForm");
EventUtil.addHandler(form, "reset", function(event){

    // получение объекта event
    event = EventUtil.getEvent(event);

    // отмена сброса формы
    EventUtil.preventDefault(event);
});
```

Кроме того, сбросить форму можно программно, вызвав метод `reset()`:

```
var form = document.getElementById("myForm");

// сброс формы
form.reset();
```

В отличие от метода `submit()`, метод `reset()` генерирует событие `reset`, как если бы был выполнен щелчок на кнопке сброса.

Многие разработчики плохо относятся к сбросу веб-форм. Он часто дезориентирует пользователя и при случайном выполнении раздражает. Сброс формы почти никогда не требуется — как правило, достаточно предоставить вместо кнопки сброса кнопку возврата на предыдущую страницу.

Поля форм

Как и другие элементы страницы, элементы форм доступны с помощью встроенных DOM-методов. Кроме того, все элементы любой формы содержатся в ее коллекции `elements` — упорядоченном списке ссылок на все поля формы и все элементы `<input>`, `<textarea>`, `<button>`, `<select>` и `<fieldset>`. Поля формы хранятся в коллекции `elements` в том порядке, в котором они расположены в разметке, и индексируются по позиции и имени, например:

```
var form = document.getElementById("form1");

// получение первого поля формы
```

```
var field1 = form.elements[0];

// получение поля с именем "textbox1"
var field2 = form.elements["textbox1"];

// получение количества полей
var fieldCount = form.elements.length;
```

Если одно имя идентифицирует несколько элементов управления формы, как в случае переключателей, все такие элементы возвращаются в коллекции `HTMLCollection`. Возьмем для примера следующий HTML-код:

Листинг FormFieldsExample01.htm


```
<form method="post" id="myForm">
    <ul>
        <li><input type="radio" name="color" value="red">Red</li>
        <li><input type="radio" name="color" value="green">Green</li>
        <li><input type="radio" name="color" value="blue">Blue</li>
    </ul>
</form>
```

Эта форма содержит три переключателя с именем `"color"`, которое связывает их вместе. В этом случае выражение `elements["color"]` возвращает коллекцию `NodeList`, содержащую все три элемента, тогда как выражение `elements[0]` возвращает только первый элемент:

Листинг FormFieldsExample01.htm

```
var form = document.getElementById("myForm");

var colorFields = form.elements["color"];
alert(colorFields.length);    // 3

var firstColorField = colorFields[0];
var firstFormField = form.elements[0];
alert(firstColorField === firstFormField);    // true
```

Этот код показывает, что первое поле формы, доступное как `form.elements[0]`, совпадает с первым элементом коллекции `form.elements["color"]`.

Элементы формы доступны также как ее свойства, например свойство form[0] представляет первое поле формы, а form[«color»] — именованное поле. Эти свойства всегда возвращают те же значения, что и их эквиваленты в коллекции elements. Этот подход поддерживается ради сохранения обратной совместимости со старыми браузерами, и вместо него следует использовать коллекцию elements.

Общие свойства полей форм

За исключением элемента `<fieldset>`, все поля форм имеют несколько общих свойств. Так как многие поля форм представляет тип `<input>`, одни свойства

используются только с полями некоторых типов, а другие доступны независимо от типа поля. Общие свойства и методы полей форм таковы:

- `disabled` — логическое значение, указывающее, отключено ли поле;
- `form` — указатель на форму, к которой относится поле (это свойство доступно только для чтения);
- `name` — имя поля;
- `readOnly` — логическое значение, указывающее, доступно ли поле только для чтения;
- `tabIndex` — порядок перехода по нажатию клавиши табуляции;
- `type` — тип поля (`"checkbox"`, `"radio"` и т. д.);
- `value` — значение поля, отправляемое серверу; у полей добавления файлов это свойство доступно только для чтения и содержит путь к файлу на компьютере.

Все свойства, кроме `form`, можно изменять динамически, например:

```
var form = document.getElementById("myForm");
var field = form.elements[0];

// изменение значения
field.value = "Another value";

//проверка значения form
alert(field.form === form);    // true

// установка фокуса для поля
field.focus();

// отключение поля
field.disabled = true;

// изменение типа поля (не рекомендуется, но возможно для элементов <input>)
field.type = "checkbox";
```

Возможность динамически изменять свойства полей форм позволяет в любой момент модифицировать форму самыми разными способами. Например, при работе с веб-формами пользователи иногда щелкают на кнопке отправки несколько раз. В приложениях электронной коммерции это может привести к выставлению нескольких счетов, что крайне нежелательно. Для решения этой проблемы можно отключить кнопку отправки в обработчике события `submit`:

Листинг FormFieldsExample02.htm


```
// предотвращение многократной отправки данных формы
EventUtil.addHandler(form, "submit", function(event){
    event = EventUtil.getEvent(event);
    var target = EventUtil.getTarget(event);
```

```
    // получение кнопки отправки
    var btn = target.elements["submit-btn"];

    // отключение кнопки отправки
    btn.disabled = true;
});
```

Этот код подключает к форме обработчик события `submit`, который получает кнопку отправки и присваивает ее свойству `disabled` значение `true`. Событие `click` кнопки отправки в этом случае бесполезно, потому что одни браузеры генерируют его раньше, чем событие `submit`, а другие наоборот. Если браузер сначала генерирует событие `click`, кнопка будет отключена до отправки данных формы, которые вообще нельзя будет отправить. По этой причине кнопку отправки следует отключать в обработчике события `submit`. Имейте в виду, что этот подход не годится, если данные формы отправляются без использования кнопки отправки, потому что событие `submit` генерируется только этой кнопкой.

Свойство `type` есть у всех полей формы, кроме `<fieldset>`. У элементов `<input>` оно имеет то же значение, что и HTML-атрибут `type`. Значения свойства `type` у других элементов представлены в таблице.

Описание	Пример HTML-кода	Значение свойства type
Список с возможностью одиночного выбора	<select>...</select>	"select-one"
Список с возможностью множественного выбора	<select multiple>...</select>	"select-multiple"
Пользовательская кнопка	<button>...</button>	"submit"
Пользовательская кнопка (не отправки)	<button type="button">...</button>	"button"
Пользовательская кнопка сброса	<button type="reset">...</button>	"reset"
Пользовательская кнопка отправки	<button type="submit">...</button>	"submit"

У элементов `<input>` и `<button>` свойство `type` можно изменять динамически, а у элемента `<select>` оно доступно только для чтения.

Общие методы полей форм

У каждого поля формы есть методы `focus()` и `blur()`. Метод `focus()` назначает фокус полю формы, то есть делает его активным, после чего поле начинает реагировать на события клавиатуры. Например, в текстовом поле, получившем фокус, появляется курсор, показывающий, что оно готово принимать ввод. Метод `focus()` чаще всего используется для привлечения внимания к какой-то части страницы. Например,

при загрузке формы фокус часто назначают ее первому полю. Это можно сделать, вызвав метод `focus()` для первого поля в обработчике события `load`:

```
EventUtil.addHandler(window, "load", function(event){
    document.forms[0].elements[0].focus();
});
```

При выполнении этого кода возникнет ошибка, если первым полем формы является элемент `<input>` типа `"hidden"` или если поле было скрыто с помощью CSS-свойства `display` или `visibility`.

В спецификации HTML5 для полей формы предлагается атрибут `autofocus`, который в поддерживающих его браузерах автоматически назначает фокус элементу без использования JavaScript-кода, например:

```
<input type="text" autofocus>
```

Чтобы предыдущий пример правильно работал с атрибутом `autofocus`, нужно сначала определить, задан ли он, и если да, не вызывать метод `focus()`:

Листинг FocusExample01.htm


```
EventUtil.addHandler(window, "load", function(event){
    var element = document.forms[0].elements[0];

    if (element.autofocus !== true){
        element.focus(); console.log("JS focus");
    }
});
```

Поскольку `autofocus` является логическим атрибутом, в поддерживающих его браузерах свойство `autofocus` равно `true` (если атрибут не поддерживается, оно содержит пустую строку). Таким образом, этот код вызывает метод `focus()`, только если свойство `autofocus` не равно `true`, что обеспечивает совместимость с последующими версиями браузеров. Свойство `autofocus` поддерживается в Firefox 4+, Safari 5+, Chrome и Opera 9.6+.

По умолчанию фокус можно назначать только элементам форм. Чтобы назначить фокус любому другому элементу, нужно присвоить его свойству tabIndex значение –1 и вызвать метод focus(). Этот прием поддерживают все браузеры, кроме Opera.

Метод `blur()` противоположен методу `focus()`: он отменяет для элемента фокус, но не назначает его никакому другому элементу. Когда не было атрибута `readonly`, этот метод использовался для создания полей, доступных только для чтения, а сейчас он применяется редко. Вот пример его вызова:

```
document.forms[0].elements[0].blur();
```

Общие события полей форм

В дополнение к событиям мыши, клавиатуры, изменения DOM-структуры и HTML-событиям все поля форм поддерживают следующие три события:

- `blur` — генерируется при утрате фокуса полем;
- `change` — генерируется для элементов `<input>` и `<textarea>` при утрате фокуса, если свойство `value` было изменено, а также для элементов `<select>` при выборе другого элемента списка;
- `focus` — генерируется при получении фокуса полем.

События `blur` и `focus` возникают в результате действий пользователя и при вызове методов `blur()` и `focus()` соответственно. Эти события одинаковы у всех полей формы, тогда как событие `change` генерируется для разных элементов управления в разное время. Для элементов `<input>` и `<textarea>` оно возникает при утрате фокуса, если за время нахождения элемента в фокусе у него изменилось свойство `value`. Для элементов `<select>` событие `change` возникает, когда пользователь выбирает другой элемент списка, при этом не требуется, чтобы элемент утратил фокус.

События `focus` и `blur` обычно применяют для изменения пользовательского интерфейса, вывода визуальных подсказок или доступа к дополнительной функциональности элемента (например, для вывода раскрывающегося меню с возможными значениями текстового поля). С помощью события `change` обычно проверяют данные, введенные в поле. Например, если текстовое поле должно принимать только числа, событие `focus` можно использовать для изменения фонового цвета поля при его активации, событие `blur` — для восстановления цвета, предлагаемого по умолчанию, а событие `change` — для изменения фонового цвета на красный при вводе нецифровых символов:

Листинг FormFieldEventsExample01.htm


```
var textbox = document.forms[0].elements[0];

EventUtil.addHandler(textbox, "focus", function(event){
    event = EventUtil.getEvent(event);
    var target = EventUtil.getTarget(event);

    if (target.style.backgroundColor != "red"){
        target.style.backgroundColor = "yellow";
    }
});

EventUtil.addHandler(textbox, "blur", function(event){
    event = EventUtil.getEvent(event);
    var target = EventUtil.getTarget(event);

    if (/[^\d]/.test(target.value)){
        target.style.backgroundColor = "red";
    } else {
        target.style.backgroundColor = "";
```

```
    }
});

EventUtil.addHandler(textbox, "change", function(event){
    event = EventUtil.getEvent(event);
    var target = EventUtil.getTarget(event);

    if (/[^\d]/.test(target.value)){
        target.style.backgroundColor = "red";
    } else {
        target.style.backgroundColor = "";
    }
});
```

Обработчик события `focus` просто изменяет фоновый цвет текстового поля на желтый, чтобы сразу было ясно, что оно активно. Обработчики событий `blur` и `change` изменяют фоновый цвет поля на красный при обнаружении в нем нецифровых символов. Для проверки символов значение текстового поля сопоставляется с простым регулярным выражением. Проверка выполняется в обоих обработчиках событий, чтобы поведение текстового поля оставалось согласованным независимо от изменений его значения.

Отношения между событиями blur и change не формализованы. В некоторых браузерах событие blur генерируется раньше, чем change, в других — наоборот, так что при их обработке будьте внимательны.

Работа с текстовыми полями

Однострочное текстовое поле представляется HTML-элементом `<input>`, а многострочное — элементом `<textarea>`. Эти два элемента управления очень похожи и в большинстве случаев работают одинаково, но между ними есть важные различия.

По умолчанию элемент `<input>` отображает текстовое поле, даже если его атрибут `type` опущен (по умолчанию он имеет значение `"text"`). Атрибут `size` определяет ширину текстового поля в видимых символах. Атрибут `value` указывает первоначальное значение текстового поля, а атрибут `maxlength` — максимально допустимое количество символов в поле. Например, следующий код создает текстовое поле, которое может показывать 25 символов, но поддерживает значения длиной до 50 символов:

```
<input type="text" size="25" maxlength="50" value="initial value">
```

Элемент `<textarea>` всегда выводит на экран многострочное текстовое поле. Для указания его размеров можно использовать атрибуты `rows` и `cols`, которые определяют соответственно высоту и ширину текстового поля в символах. В отличие от элемента `<input>`, первоначальное значение `<textarea>` должно быть указано между тегами `<textarea>` и `</textarea>`:

```
<textarea rows="25" cols="5">initial value</textarea>
```

Кроме того, для элемента `<textarea>` нельзя ограничить количество символов в HTML-коде.

Несмотря на различия разметки, содержимое текстовых полей обоих типов хранится в свойстве `value`, которое можно использовать для чтения и задания значения текстового поля, например:

```
var textbox = document.forms[0].elements["textbox1"];
alert(textbox.value);

textbox.value = "Some new value";
```

Читать и задавать значения текстовых полей с помощью свойства `value` предпочтительнее, чем использовать для этого стандартные DOM-методы. Например, не следует задействовать метод `setAttribute()` для установки атрибута `value` элемента `<input>` или изменять первый дочерний узел элемента `<textarea>`. Изменения свойства `value` также не всегда отражаются в DOM-структуре, так что для работы со значениями текстовых полей DOM-методы лучше не использовать.

Выделение текста

У текстовых полей обоих типов есть метод `select()`, который выделяет все содержимое поля. Большинство браузеров автоматически назначают фокус текстовому полю при вызове его метода `select()` (Opera не назначает). Этот метод не принимает аргументов и его можно вызвать в любое время, например:

```
var textbox = document.forms[0].elements["textbox1"];
textbox.select();
```

Разработчики часто выделяют весь текст в текстовом поле, когда оно получает фокус, особенно если у него есть значение, предлагаемое по умолчанию. Идея в том, что так пользователь может сразу удалить весь текст, если нужно ввести другое значение. Вот как это можно сделать:

Листинг TextboxSelectExample01.htm


```
EventUtil.addHandler(textbox, "focus", function(event){
    event = EventUtil.getEvent(event);
    var target = EventUtil.getTarget(event);

    target.select();
});
```

Этот код выделяет весь текст в текстовом поле, как только оно получает фокус. Это делает работу с формами гораздо удобнее.

Событие select

Методу `select()` сопутствует событие `select`, которое генерируется при выделении текста в текстовом поле. Точный момент возникновения этого события зависит от браузера. В Internet Explorer 9+, Opera, Firefox, Chrome и Safari оно генерируется, когда пользователь завершает выделение текста, а в Internet Explorer 8 и более ранних версий — при выделении одной буквы. Событие `select` также возникает при вызове метода `select()`. Вот простой пример:

Листинг SelectEventExample01.htm


```
var textbox = document.forms[0].elements["textbox1"];
EventUtil.addHandler(textbox, "select", function(event){
    var alert("Text selected: " + textbox.value);
});
```

Получение выделенного текста

Событие `select` информирует о выделении текста, но не сообщает, какой текст выделен, поэтому в HTML5 для получения выделенного текста к текстовым полям были добавлены свойства `selectionStart` и `selectionEnd`. Они содержат отсчитываемые от нуля числа, указывающие границы выделенного текста (смещения его начала и конца). Для получения текста, выделенного в текстовом поле, можно использовать следующий код:

```
function getSelectedText(textbox){
    return textbox.value.substring(textbox.selectionStart,
                                   textbox.selectionEnd);
}
```

Поскольку метод `substring()` работает со смещениями строк, значения `selectionStart` и `selectionEnd` можно передавать в него напрямую.

Это решение работает в Internet Explorer 9+, Firefox, Safari, Chrome и Opera. В Internet Explorer 8 и более ранних версий эти свойства не поддерживаются, так что нужен другой подход.

В старых версиях Internet Explorer доступен объект `document.selection`, который содержит сведения о выделении текста во всем документе, при этом неизвестно, где на странице находится этот текст. Однако при использовании этого объекта вместе с событием `select` выделенный текст наверняка находится в текстовом поле, которое сгенерировало событие. Чтобы получить выделенный текст, нужно сначала создать диапазон (см. главу 12), а затем извлечь из него текст:

Листинг TextboxGet Select edTextExample01.htm

```
function getSelectedText(textbox){
    if (typeof textbox.selectionStart == "number"){
        return textbox.value.substring(textbox.selectionStart,
                                       textbox.selectionEnd);
```

```
    } else if (document.selection){
        return document.selection.createRange().text;
    }
}
```

Новая версия функции возвращает выделенный текст независимо от браузера. Заметьте, что для использования объекта `document.selection` аргумент `textbox` не требуется.

Частичное выделение текста

В HTML5 разрешается выделять фрагменты текстовых полей. Для этого используется метод `setSelectionRange()`, первоначально реализованный в Firefox, а теперь доступный для всех текстовых полей в дополнение к методу `select()`. Как и метод `substring()` строки, он принимает два аргумента: индекс первого выделяемого символа и индекс конца выделения, например:

```
textbox.value = "Hello world!"

// выделение всего текста
textbox.setSelectionRange(0, textbox.value.length);    // "Hello world!"

// выделение трех первых символов
textbox.setSelectionRange(0, 3);    // "Hel"

//выделение символов с 4 по 6
textbox.setSelectionRange(4, 7);    // "o w"
```

Чтобы показать выделение, нужно назначить фокус текстовому полю или непосредственно перед вызовом метода `setSelectionRange()` или после него. Этот подход работает в Internet Explorer 9, Firefox, Safari, Chrome и Opera.

В Internet Explorer 8 и более ранних версий частично выделять текст можно с помощью диапазонов (см. главу 12). Чтобы выделить часть текста в текстовом поле, нужно сначала создать диапазон методом `createTextRange()`, который в Internet Explorer доступен для текстовых полей, свернуть диапазон к началу поля с помощью метода `collapse()`, а затем воспользоваться методами `moveStart()` и `moveEnd()` для настройки границ диапазона. Метод `moveStart()` перемещает начальную и конечную точки диапазона в одно место, а метод `moveEnd()` задает общее количество выделяемых символов. Наконец, для выделения текста нужно вызывать метод `select()` диапазона:

```
textbox.value = "Hello world!";

var range = textbox.createTextRange();

// выделение всего текста
range.collapse(true);
range.moveStart("character", 0);
range.moveEnd("character", textbox.value.length);    // "Hello world!"
range.select();
```

```
// выделение первых трех символов
range.collapse(true);
range.moveStart("character", 0);
range.moveEnd("character", 3);
range.select();                          // "Hel"

// выделение символов с 4 по 6
range.collapse(true);
range.moveStart("character", 4);
range.moveEnd("character", 3);
range.select();                          // "o w"
```

Как и в других браузерах, чтобы выделение было видно, фокус должен принадлежать текстовому полю.

Эти два подхода можно объединить в одной кроссбраузерной функции:

Листинг TextboxPartialSelectionExample01.htm


```
function selectText(textbox, startIndex, stopIndex){
    if (textbox.setSelectionRange){
        textbox.setSelectionRange(startIndex, stopIndex);
    } else if (textbox.createTextRange){
        var range = textbox.createTextRange();
        range.collapse(true);
        range.moveStart("character", startIndex);
        range.moveEnd("character", stopIndex - startIndex);
        range.select();
    }
    textbox.focus();
}
```

Функция `selectText()` принимает три аргумента: целевое текстовое поле, индекс начала выделения и индекс символа, перед которым нужно завершить выделение. Сначала функция проверяет, есть ли у текстового поля метод `setSelectionRange()`. Если есть, он вызывается, в противном случае функция проверяет, поддерживает ли текстовое поле метод `createTextRange()`. Если да, текст выделяется с помощью диапазона. Последним действием функции является назначение фокуса текстовому полю, чтобы выделение было видно. Функцию `selectText()` можно использовать следующим образом:

```
textbox.value = "Hello world!"

// выделение всего текста
selectText(textbox, 0, textbox.value.length);    // "Hello world!"

// выделение первых трех символов
selectText(textbox, 0, 3);            // "Hel"

// выделение символов с 4 по 6
selectText(textbox, 4, 7);            // "o w"
```

Частичное выделение текста полезно при реализации полей для ввода текста с расширенными возможностями, например полей с автозавершением ввода.

Фильтрация ввода

Часто требуется, чтобы текстовые поля принимали только данные определенного типа или в определенном формате, например данные должны содержать какие-то символы или соответствовать некоторому шаблону. По умолчанию возможности проверки контента текстовых полей весьма ограниченны, поэтому для *фильтрации ввода* (input filtering) необходимо задействовать JavaScript. Используя события и другие возможности DOM, можно превратить обычное текстовое поле в элемент, по-настоящему «понимающий» свои данные.

Блокировка символов

Некоторые типы значений никогда не могут содержать определенные символы или, наоборот, требуют их наличия. Скажем, текстовое поле для ввода номера телефона должно принимать только цифры. Для вставки символов в текстовое поле используется событие `keypress`, но, отменив действие, предлагаемое по умолчанию, можно заблокировать ввод символов. Например, следующий код блокирует нажатия всех клавиш:

```
EventUtil.addHandler(textbox, "keypress", function(event){
    event = EventUtil.getEvent(event);
    EventUtil.preventDefault(event);
});
```

По сути, этот код делает текстовое поле доступным только для чтения, блокируя нажатия всех клавиш. Можно также блокировать только определенные символы по их кодам. Например, следующий код блокирует все символы, кроме цифр:

```
EventUtil.addHandler(textbox, "keypress", function(event){
    event = EventUtil.getEvent(event);
    var target = EventUtil.getTarget(event);
    var charCode = EventUtil.getCharCode(event);

    if (!/\d/.test(String.fromCharCode(charCode))){
        EventUtil.preventDefault(event);
    }
});
```

В этом примере для получения кода символа используется кроссбраузерный метод `EventUtil.getCharCode()`. Затем метод `String.fromCharCode()` преобразует код символа в строку, которая сопоставляется с регулярным выражением `/\d/`, определяющим все цифры. Если символ не является цифрой, событие блокируется методом `EventUtil.preventDefault()`.

Хотя событие keypress должно генерироваться только при нажатии клавиши ввода символа, некоторые браузеры генерируют его и для других клавиш. Firefox и Safari (до версии 3.1) генерируют событие keypress для клавиш со стрелками, а также клавиш Backspace и Delete, а Safari 3.1 и более поздних версий — нет. Это означает, что просто заблокировать все нецифровые клавиши нельзя, потому что иначе будут заблокированы и эти полезные клавиши. К счастью, можно легко определить, когда нажата одна из них. В Firefox все несимвольные клавиши, которые генерируют событие keypress, имеют код 0, тогда как Safari до версии 3 назначает всем им код 8. В общем, чтобы не блокировать никакие коды символов, меньшие 10, функцию можно переписать следующим образом:

```
EventUtil.addHandler(textbox, "keypress", function(event){
    event = EventUtil.getEvent(event);
    var target = EventUtil.getTarget(event);
    var charCode = EventUtil.getCharCode(event);

    if (!/\d/.test(String.fromCharCode(charCode)) && charCode > 9){
        EventUtil.preventDefault(event);
    }
});
```

Этот обработчик правильно работает во всех браузерах, блокируя нецифровые символы, но разрешая нажатия всех основных клавиш, которые также генерируют событие keypress.

Нам осталось решить проблему с копированием, вставкой и любыми другими действиями с нажатием клавиши Ctrl. Во всех браузерах, кроме Internet Explorer, предыдущий код блокирует сочетания Ctrl+C, Ctrl+V и любые другие сочетания с клавишей Ctrl, поэтому перед блокировкой мы должны убедиться в том, что она не нажата:

Листинг TextboxInputFilteringExample01.htm


```
EventUtil.addHandler(textbox, "keypress", function(event){
    event = EventUtil.getEvent(event);
    var target = EventUtil.getTarget(event);
    var charCode = EventUtil.getCharCode(event);

    if (!/\d/.test(String.fromCharCode(charCode)) && charCode > 9 &&
            !event.ctrlKey){
        EventUtil.preventDefault(event);
    }
});
```

Теперь все варианты поведения, доступные для текстовых полей по умолчанию, будут работать. Слегка изменив этот код, можно разрешить или запретить ввод любых символов в текстовом поле.

Работа с буфером обмена

Поддержка событий, связанных с буфером обмена, и поддержка доступа к нему из JavaScript-кода впервые была реализована в Internet Explorer. Эта реализация

стала стандартом де-факто, на основе которого в Safari 2, Chrome и Firefox 3 были реализованы похожие возможности (Opera 11 все еще не поддерживает работу с буфером обмена средствами JavaScript). Еще позже события буфера обмена были добавлены в спецификацию HTML5. Этих событий шесть:

- `beforecopy` — генерируется непосредственно перед копированием;
- `copy` — генерируется при копировании;
- `beforecut` — генерируется непосредственно перед вырезанием;
- `cut` — генерируется при вырезании;
- `beforepaste` — генерируется непосредственно перед вставкой;
- `paste` — генерируется при вставке.

Поскольку это довольно новый стандарт доступа к буферу обмена, соответствующие события и объекты работают по-разному в зависимости от браузера. В Safari, Chrome и Firefox события `beforecopy`, `beforecut` и `beforepaste` генерируются только при выводе контекстного меню для текстового поля (в предвидении возможного события буфера обмена), а Internet Explorer генерирует их также непосредственно перед событиями `copy`, `cut` и `paste`. События `copy`, `cut` и `paste` возникают во всех браузерах без сюрпризов: при выборе соответствующих команд в контекстном меню и нажатии требуемых сочетаний клавиш.

События `beforecopy`, `beforecut` и `beforepaste` позволяют изменить данные, отправляемые в буфер обмена или получаемые из него, до фактического выполнения операции. Однако отмена одного из этих событий не отменяет операцию с буфером обмена — для этого нужно отменить событие `copy`, `cut` или `paste`.

Данные в буфере обмена доступны через объект `clipboardData`, который принадлежит либо объекту `window` (в Internet Explorer), либо объекту `event` (в Firefox 4+, Safari и Chrome). В Firefox, Safari и Chrome объект `clipboardData` доступен только при обработке событий буфера обмена, что предотвращает несанкционированный доступ к буферу обмена; в Internet Explorer он доступен все время. Чтобы код поддерживали все браузеры, лучше использовать этот объект только при обработке событий буфера обмена.

У объекта `clipboardData` есть три метода: `getData()`, `setData()` и `clearData()`. Метод `getData()` получает строковые данные из буфера обмена, принимая в качестве аргумента формат этих данных. Internet Explorer поддерживает два формата: `"text"` и `"URL"`. Firefox, Safari и Chrome вместо этого используют MIME-типы, но поддерживают и значение `"text"` как эквивалент `"text/plain"`.

Метод `setData()` принимает тип данных и текст, который нужно поместить в буфер обмена. В этом случае Internet Explorer также поддерживает типы `"text"` и `"URL"`, а Safari и Chrome ожидают MIME-тип, но, в отличие от метода `getData()`, они не распознают тип `"text"` и игнорируют вызов `setData()` с таким аргументом. Чтобы нивелировать особенности браузеров, можно добавить в объект `EventUtil` следующие кроссбраузерные методы:

Листинг EventUtil.js


```
var EventUtil = {

    // другой код

    getClipboardText: function(event){
        var clipboardData = (event.clipboardData || window.clipboardData);
        return clipboardData.getData("text");
    },

    // другой код

    setClipboardText: function(event, value){
        if (event.clipboardData){
            return event.clipboardData.setData("text/plain", value);
        } else if (window.clipboardData){
            return window.clipboardData.setData("text", value);
        }
    },

    // другой код

};
```

Метод `getClipboardText()` сравнительно прост. Он лишь выясняет способ доступа к объекту `clipboardData`, а затем вызывает его метод `getData()` с аргументом `"text"`. Второй метод, `setClipboardText()`, чуть сложнее. Определив способ доступа к объекту `clipboardData`, он вызывает метод `setData()` с типом данных, зависящим от реализации (`"text/plain"` для Firefox, Safari и Chrome; `"text"` для Internet Explorer).

Чтение текста из буфера обмена полезно, если текстовое поле принимает только определенные символы или текст в определенном формате. Например, если текстовое поле принимает только числа, желательно проверять также значения, вставляемые в него из буфера обмена. В обработчике события `paste` можно выяснить, соответствует ли требованиям содержимое буфера обмена, и если нет — отменить действие, предлагаемое по умолчанию:

Листинг TextboxClipboardExample01.htm


```
EventUtil.addHandler(textbox, "paste", function(event){
    event = EventUtil.getEvent(event);
    var text = EventUtil.getClipboardText(event);

    if (!/^\d*$/.test(text)){
        EventUtil.preventDefault(event);
    }
});
```

Этот обработчик события `paste` позволяет вставлять в текстовое поле только числовые значения. Если содержимое буфера обмена не соответствует шаблону, операция отменяется. В Firefox, Safari и Chrome доступ к методу `getData()` возможен только в обработчике события `paste`.

Поскольку не все браузеры поддерживают доступ к буферу обмена, часто проще заблокировать одну или несколько операций с ним. В браузерах, которые поддерживают события `copy`, `cut` и `paste` (Internet Explorer, Safari, Chrome и Firefox 3+), отменить для них действие, предлагаемое по умолчанию, легко. В Opera для этого нужно заблокировать соответствующие нажатия клавиш и запретить вывод контекстного меню.

Автоматический переход по нажатию клавиши табуляции

JavaScript предоставляет несколько способов упростить работу с полями форм. Один из наиболее популярных — это автоматическое перемещение фокуса к следующему полю после заполнения текущего. Так часто делают, если длина данных известна, например если поле служит для ввода номера телефона. В США номера телефонов обычно состоят из трех частей: кода области, номера АТС и номера абонента. На веб-страницах для их ввода часто предлагается три поля:

Листинг TextboxTabForwardExample01.htm

```
<input type="text" name="tel1" id="txtTel1" maxlength="3">
<input type="text" name="tel2" id="txtTel2" maxlength="3">
<input type="text" name="tel3" id="txtTel3" maxlength="4">
```

Чтобы упростить и ускорить ввод этих данных, можно автоматически перемещать фокус к следующему полю, как только в текущее введено максимальное количество знаков. Иначе говоря, после ввода трех символов в первое поле можно назначить фокус второму, а после ввода очередных трех символов — третьему. Это можно сделать следующим образом:

Листинг TextboxTabForwardExample01.htm


```
(function(){

    function tabForward(event){
        event = EventUtil.getEvent(event);
        var target = EventUtil.getTarget(event);

        if (target.value.length == target.maxLength){
            var form = target.form;

            for (var i=0, len=form.elements.length; i < len; i++) {
                if (form.elements[i] == target) {
                    if (form.elements[i+1]){
                        form.elements[i+1].focus();
                    }
                    return;
                }
            }
        }
    }
```

```
    var textbox1 = document.getElementById("txtTel1");
    var textbox2 = document.getElementById("txtTel2");
    var textbox3 = document.getElementById("txtTel3");

    EventUtil.addHandler(textbox1, "keyup", tabForward);
    EventUtil.addHandler(textbox2, "keyup", tabForward);
    EventUtil.addHandler(textbox3, "keyup", tabForward);
})();
```

Главную роль в этом фрагменте играет функция `tabForward()`. Она проверяет, достигнута ли максимальная длина содержимого текстового поля, сравнивая его текущую длину с атрибутом `maxlength`. Если они равны (а больше символов браузер ввести не позволит), функция находит в коллекции элементов текущее текстовое поле и назначает фокус следующему элементу. Далее эта функция назначается каждому текстовому полю как обработчик события `keyup`. Поскольку оно генерируется после вставки в текстовое поле каждого нового символа, это идеальная возможность для проверки длины содержимого поля. В результате при заполнении этой простой формы пользователю никогда не придется нажимать клавишу табуляции для перемещения между полями и отправки данных формы.

Имейте в виду, что этот код специфичен для приведенной разметки и не учитывает возможное наличие скрытых полей.

API проверки ограничений в HTML5

HTML5 предлагает механизм проверки данных формы перед их отправкой серверу. Это позволяет выполнять несложную проверку данных, даже если JavaScript-сценарии недоступны или не могут быть загружены. Браузер сам проверяет данные на основе правил, а затем показывает соответствующие сообщения об ошибках (без дополнительного JavaScript-кода). Конечно, это возможно только в браузерах, которые поддерживают данную часть спецификации HTML5, в том числе в Firefox 4+, Safari 5+, Chrome и Opera 10+.

Браузер автоматически проверяет только те поля формы, для которых заданы ограничения в HTML-разметке.

Обязательные поля

Первое ограничение, которое мы обсудим, задается с помощью атрибута `required`, например:

```
<input type="text" name="username" required>
```

Поле формы, помеченное атрибутом `required` как обязательное, не может быть пустым, иначе отправить данные формы не удастся. Этот атрибут можно задать для полей `<input>`, `<textarea>` и `<select>` (в Opera до версии 11 включительно его не поддерживает поле `<select>`). В JavaScript-коде можно проверить, обязательно ли поле формы, используя свойство `required` элемента:

```
var isUsernameRequired = document.forms[0].elements["username"].required;
```

Можно также проверить, поддерживает ли браузер атрибут `required`:

```
var isRequiredSupported = "required" in document.createElement("input");
```

Этот код проверяет, есть ли у нового элемента `<input>` свойство `required`, используя простое распознавание возможностей.

При отправке формы с пустым обязательным полем браузеры ведут себя по-разному. Firefox 4 и Opera 11 блокируют отправку данных формы и выводят ниже поля всплывающую подсказку, а Safari 5 и Chrome 9 ничего не делают и отправляют данные формы как ни в чем не бывало.

Альтернативные типы ввода

В HTML5 определено несколько новых значений атрибута `type` элемента `<input>`. Они не только предоставляют дополнительные сведения о типе ожидаемых данных, но и по умолчанию обеспечивают некоторую проверку. Лучше всего поддерживаются типы `"email"` и `"url"`, каждый из которых применяет к данным специальные правила проверки:

```
<input type="email" name="email">
<input type="url" name="homepage">
```

Тип `"email"` проверяет введенный текст на соответствие шаблону адреса электронной почты, а тип `"url"` сопоставляет его с шаблоном URL-адреса. Имейте в виду, что в браузерах, упомянутых ранее в этом разделе, сопоставление с шаблоном работает неидеально, например `"-@-"` окажется допустимым адресом электронной почты. Производители браузеров все еще работают над решением этих проблем.

Чтобы узнать, поддерживает ли браузер эти новые типы, можно создать элемент на JavaScript-коде, присвоить его свойству `type` значение `"email"` или `"url"`, а затем прочитать его. Старые браузеры автоматически заменяют неизвестные им типы значением `"text"`, а браузеры, поддерживающие новые типы, возвращают правильные значения, например:

```
var input = document.createElement("input");
input.type = "email";

var isEmailSupported = (input.type == "email");
```

Помните, что пустое поле также считается допустимым, если для него не задан атрибут `required`. Кроме того, указание специального типа ввода не мешает пользователю ввести недопустимое значение, а только включает некоторые правила проверки, применяемые по умолчанию.

Числовые диапазоны

Кроме "email" и "url", HTML5 определяет еще несколько новых типов элементов ввода: "number", "range", "datetime", "datetime-local", "date", "month", "week" и "time". Все они являются числовыми. Эти типы не очень хорошо поддерживаются браузерами, поэтому использовать их следует осторожно, пока производители браузеров работают над логикой функционала и взаимной совместимостью. Сведения в этом разделе скорее приведены с расчетом на будущее.

Для каждого из этих числовых типов можно задать атрибуты min (наименьшее возможное значение), max (наибольшее возможное значение) и step (разница между отдельными значениями от min до max). Например, следующий код принимает только числа от 0 до 100 с интервалом 5:

```
<input type="number" min="0" max="100" step="5" name="count">
```

Некоторые браузеры для этого кода выводят счетчик с кнопками вверх и вниз для изменения значения счетчика.

Каждому из атрибутов соответствуют свойства элемента, которые можно читать и изменять с помощью JavaScript-кода. Кроме того, у них есть методы stepUp() и stepDown(), которые принимают число, вычитаемое из текущего значения или добавляемое к нему (по умолчанию оно увеличивается или уменьшается на единицу). Эти методы еще не реализованы в браузерах, но будут использоваться следующим образом:

```
input.stepUp();        // увеличение на 1
input.stepUp(5);       // увеличение на 5
input.stepDown();      // уменьшение на 1
input.stepDown(10);    // уменьшение на 10
```

Шаблоны ввода

В HTML5 для текстовых полей представлен атрибут pattern, указывающий регулярное выражение, с которым должно быть сопоставлено введенное значение. Например, следующее текстовое поле принимает только числа:

```
<input type="text" pattern="\d+" name="count">
```

Предполагается, что в начале и в конце шаблона есть знаки ^ и $ соответственно. Это означает, что ввод должен точно соответствовать шаблону от начала до конца.

Как и в коде с альтернативными типами ввода, шаблон не мешает пользователю ввести недопустимый текст, а просто указывает браузеру, допустимо значение или нет. Прочитать шаблон можно с помощью свойства pattern:

```
var pattern = document.forms[0].elements["count"].pattern;
```

Проверить, поддерживает ли браузер атрибут pattern, можно следующим образом:

```
var isPatternSupported = "pattern" in document.createElement("input");
```

Проверка допустимости

С помощью метода checkValidity() можно проверить, допустимо ли значение конкретного поля формы. Он доступен для всех элементов и возвращает true, если значение поля допустимо, и false в противном случае. При проверке используются условия, описанные ранее в этом разделе, так что обязательное поле без значения или поле со значением, которое не соответствует шаблону, считаются недопустимыми, например:

```
if (document.forms[0].elements[0].checkValidity()){
    // значение поля допустимо, продолжаем
} else {
    // поле недопустимо
}
```

Вызвав метод checkValidity() для самой формы, можно проверить все ее поля. Если все они допустимы, метод возвратит true, а если хотя бы одно из полей недопустимо — false:

```
if(document.forms[0].checkValidity()){
    // форма допустима, продолжаем
} else {
    // поле формы недопустимо
}
```

В то время как метод checkValidity() просто сообщает, допустимо ли значение поля, свойство validity указывает точную причину, почему оно допустимо или нет. У этого объекта есть следующие свойства логического типа:

- customError — true, если с помощью метода setCustomValidity() было задано пользовательское сообщение, иначе false;
- patternMismatch — true, если значение не соответствует заданному атрибуту pattern;
- rangeOverflow — true, если значение больше, чем значение max;
- rangeUnderflow — true, если значение меньше, чем значение min;
- stepMisMatch — true, если значение не соответствует атрибуту step с учетом значений min и max;
- tooLong — true, если значение содержит больше символов, чем допускает свойство maxlength (некоторые браузеры, такие как Firefox 4, автоматически ограничивают количество символов, так что это значение может всегда быть равно false);

- `typeMismatch` — `true`, если значение не соответствует требуемому формату `"email"` или `"url"`;
- `valid` — `true`, если все остальные свойства равны `false` (это то же значение, которое возвращает метод `checkValidity()`);
- `valueMissing` — `true`, если поле, отмеченное как обязательное, пусто.

С помощью свойства `validity` можно получить более конкретные сведения о том, что не так с формой, например:

```
if (input.validity && !input.validity.valid){
    if (input.validity.valueMissing){
        alert("Please specify a value.")          // Укажите значение
    } else if (input.validity.typeMismatch){
        alert("Please enter an email address."); // Укажите адрес
                                                  // электронной почты
    } else {
        alert("Value is invalid.");               // Значение недопустимо
    }
}
```

Отключение проверки

С помощью атрибута `novalidate` можно отключить для формы все виды проверки:

```
<form method="post" action="signup.php" novalidate>
    <!-- код элементов формы -->
</form>
```

Этот атрибут можно также получить или задать, используя свойство `noValidate`, которое равно `true`, если атрибут задан, и `false`, если он отсутствует:

```
document.forms[0].noValidate = true;    // отключение проверки
```

Если у формы несколько кнопок отправки, можно отключить проверку формы для конкретной кнопки, добавив к ней атрибут `formnovalidate`:

```
<form method="post" action="foo.php">
    <!-- код элементов формы -->
    <input type="submit" value="Regular Submit">
    <input type="submit" formnovalidate name="btnNoValidate"
           value="Non-validating Submit">
</form>
```

В этом примере форма проверяется при щелчке на первой кнопке отправки, но не на второй. То же самое можно сделать и в JavaScript-коде:

```
// отключение проверки
document.forms[0].elements["btnNoValidate"].formNoValidate = true;
```

Работа со списками

Списки создают с помощью элементов `<select>` и `<option>`. Чтобы упростить работу с ними, тип `HTMLSelectElement` предоставляет следующие свойства и методы в дополнение к тем, которые доступны для всех полей форм:

- `add(новыйЭлемент, связанныйЭлемент)` — добавляет новый элемент (`<option>`) перед связанным элементом;
- `multiple` — логическое значение, указывающее, разрешен ли множественный выбор (эквивалент HTML-атрибута `multiple`);
- `options` — коллекция `HTMLCollection` элементов `<option>` в элементе управления;
- `remove(индекс)` — удаляет элемент в указанной позиции;
- `selectedIndex` — отсчитываемый от нуля индекс выбранного элемента или значение –1, если никакой элемент не выбран (если список поддерживает множественный выбор, это всегда первый выбранный элемент);
- `size` — количество видимых строк списка (эквивалент HTML-атрибута `size`).

Свойство `type` списка может иметь значение `"select-one"` или `"select-multiple"` в зависимости от того, задан ли атрибут `multiple`. Значение свойства `value` определяется для списка на основе выбранного в нем элемента по следующим правилам:

- если никакой элемент не выбран, значением списка является пустая строка;
- если выбран элемент, для которого задан атрибут `value`, значением списка является атрибут `value` выбранного элемента (это верно, даже если атрибут `value` является пустой строкой);
- если выбран элемент, для которого не задан атрибут `value`, значением списка является текст элемента;
- если выбрано несколько эдементов, значение списка определяется по первому выбранному элементу с использованием двух предыдущих правил.

Рассмотрим следующий список:

```
<select name="location" id="selLocation">
    <option value="Sunnyvale, CA">Sunnyvale</option>
    <option value="Los Angeles, CA">Los Angeles</option>
    <option value="Mountain View, CA">Mountain View</option>
    <option value="">China</option>
    <option>Australia</option>
</select>
```

Если выбрать первый элемент этого списка, значением списка будет `"Sunnyvale, CA"`. Если выбрать элемент с текстом `"China"`, значением списка будет пустая строка,

потому что атрибут value пуст. Наконец, если выбрать последний элемент, значением списка будет "Australia", потому что у этого элемента <option> нет атрибута value.

Каждому элементу <option> соответствует DOM-объект HTMLOptionElement, который предоставляет следующие дополнительные свойства, позволяющий упростить доступ к данным:

- index — индекс элемента в коллекции options;
- label — надпись элемента (эквивалент HTML-атрибута label);
- selected — логическое значение, указывающее, выбран ли элемент (чтобы выбрать элемент, присвойте этому свойству значение true);
- text — текст элемента;
- value — значение элемента (эквивалент HTML-атрибута value).

Большинство свойств элемента <option> просто ускоряют доступ к данным элементов списка. Вместо них можно использовать и обычные возможности DOM, но это менее эффективно, например:

```
var selectbox = document.forms[0].elements["location"];

// не рекомендуется
var text = selectbox.options[0].firstChild.nodeValue;     // текст элемента
var value = selectbox.options[0].getAttribute("value");   // значение элемента
```

Этот код получает текст и значение первого элемента списка с помощью стандартных средств DOM. Сравните его с использованием специальных свойств:

```
var selectbox = document.forms[0].elements["location"];

// предпочтительный вариант
var text = selectbox.options[0].text;      // текст элемента
var value = selectbox.options[0].value;    // значение элемента
```

При работе с элементами списков лучше использовать специальные свойства, которые хорошо поддерживаются всеми браузерами, тогда как взаимодействия элементов управления форм при манипулировании DOM-узлами могут зависеть от браузера. Изменять текст или значения элементов <option> с помощью стандартных средств модели DOM не рекомендуется.

Использование события change со списками также имеет особенности. В отличие от других полей форм, которые генерируют его после изменения значения и утраты фокуса, для списка оно генерируется при выборе одного из его элементов.

Значение, возвращаемое свойством value, зависит от браузера. Во всех браузерах свойство value всегда равно атрибуту value. Если он не указан, Internet Explorer 8 и более ранних версий возвращают пустую строку, тогда как Internet Explorer 9+, Safari, Firefox, Chrome и Opera возвращают значение свойства text.

Выбор элементов списка

Самый простой способ получить выбранный элемент списка с одиночным выбором — воспользоваться свойством `selectedIndex`, например:

```
var selectedOption = selectbox.options[selectbox.selectedIndex];
```

Этот код можно задействовать для вывода всех сведений о выбранном элементе:

Листинг SelectboxExample01.htm


```
var selectedIndex = selectbox.selectedIndex;
var selectedOption = selectbox.options[selectedIndex];
alert("Selected index: " + selectedIndex +
      "\nSelected text: " + selectedOption.text +
      "\nSelected value: " + selectedOption.value);
```

Этот код выводит на экран оповещение с индексом, текстом и значением выбранного элемента.

Если список поддерживает множественный выбор, свойство `selectedIndex` все равно работает так, как если бы можно было выбрать только один элемент. При установке свойства `selectedIndex` прежний выбор отменяется и выбирается единственный указанный элемент, а при чтении этого свойства возвращается индекс только первого выбранного элемента.

Элемент также можно выбрать, получив ссылку на него и присвоив его свойству `selected` значение `true`. Например, следующий код выбирает первый элемент списка:

```
selectbox.options[0].selected = true;
```

В отличие от `selectedIndex`, установка свойства `selected` для элемента не отменяет выбор других элементов списка с множественным выбором, что позволяет динамически выбирать любое количество элементов. При изменении свойства `selected` для элемента списка с одиночным выбором выбор другого элемента отменяется. Присвоение значения `false` свойству `selected` не влияет на список с одиночным выбором.

С помощью свойства `selected` можно узнать, какие элементы списка выбраны. Чтобы получить все выбранные элементы, можно перебрать набор элементов в цикле, проверяя их свойство `selected`:

Листинг SelectboxExample03.htm

```
function getSelectedOptions(selectbox){
    var result = new Array();
    var option = null;

    for (var i=0, len=selectbox.options.length; i < len; i++){
        option = selectbox.options[i];
        if (option.selected){
```

```
            result.push(option);
        }
    }

    return result;
}
```

Эта функция возвращает массив элементов, выбранных в переданном ей списке. Сначала она создает массив для хранения результатов, а затем в цикле `for` перебирает все элементы списка, проверяя у каждого из них свойство `selected`. Если элемент выбран, он добавляется в массив `result`, который в конце возвращается из функции. Использовать ее для получения сведений о выбранных элементах можно следующим образом:

Листинг SelectboxExample03.htm


```
var selectbox = document.getElementById("selLocation");
var selectedOptions = getSelectedOptions(selectbox);
var message = "";

for (var i=0, len=selectedOptions.length; i < len; i++){
    message += "Selected index: " + selectedOptions[i].index +
       "\nSelected text: " + selectedOptions[i].text +
       "\nSelected value: " + selectedOptions[i].value + "\n\n";
}

alert(message);
```

Этот пример составляет в цикле `for` сообщение со сведениями о выбранных элементах, а затем выводит его на экран. Сообщение включает индекс, текст и значение каждого выбранного элемента. Этот код можно использовать со списками, поддерживающими как одиночный, так и множественный выбор.

Добавление элементов в список

JavaScript поддерживает несколько способов динамического создания элементов и их добавления в списки. Первый основан на использовании DOM:

Листинг SelectboxExample04.htm

```
var newOption = document.createElement("option");
newOption.appendChild(document.createTextNode("Option text"));
newOption.setAttribute("value", "Option value");

selectbox.appendChild(newOption);
```

Этот код создает элемент `<option>`, добавляет в него текстовый узел, задает атрибут `value` и присоединяет новый элемент к списку, при этом список сразу же обновляется.

Новые элементы также можно создавать с помощью конструктора `Option`, который использовался еще до появления DOM. Он принимает два аргумента, `text` и `value`,

хотя второй аргумент не обязателен. Несмотря на то что этот конструктор создает экземпляр `Object`, браузеры, соответствующие DOM, возвращают элемент `<option>`. Это означает, что для добавления элемента в список можно использовать метод `appendChild()`, например:

Листинг SelectboxExample04.htm

```
var newOption = new Option("Option text", "Option value");
selectbox.appendChild(newOption);  // проблемы в IE до версии 8 включительно
```

Этот подход нормально работает во всех браузерах, кроме Internet Explorer 8 и более ранних версий, где текст нового элемента задается неправильно.

Другой способ добавить новый элемент в список — использовать метод `add()` списка. В спецификации DOM сказано, что этот метод принимает два аргумента: новый элемент и элемент, перед которым его нужно вставить. Чтобы добавить элемент в конец списка, нужно передать в качестве второго аргумента значение `null`. В Internet Explorer 8 и более ранних версий реализация метода `add()` отличается тем, что второй аргумент не обязателен и представляет индекс элемента, перед которым нужно вставить новый элемент. В браузерах, соответствующих модели DOM, второй аргумент обязателен, так что в кроссбраузерном коде нельзя просто использовать один аргумент (Internet Explorer 9 соответствует DOM). Если передать в качестве второго аргумента значение `undefined`, элемент будет без проблем добавлен в конец списка во всех браузерах, например:

Листинг SelectboxExample04.htm

```
var newOption = new Option("Option text", "Option value");
selectbox.add(newOption, undefined);    // оптимальное решение
```

Этот код работает надлежащим образом во всех версиях Internet Explorer и всех браузерах, соответствующих DOM. Если нужно вставить новый элемент не в конец списка, следует использовать DOM-подход с методом `insertBefore()`.

Как и в HTML, задавать значение элемента списка не требуется. Конструктор Option работает и с одним аргументом (текст элемента).

Удаление элементов списка

Удалить элемент списка можно несколькими способами. Во-первых, можно вызвать DOM-метод `removeChild()`, передав ему элемент, который нужно удалить:

```
selectbox.removeChild(selectbox.options[0]);    // удаление первого элемента
```

Во-вторых, можно использовать метод `remove()` списка, который принимает индекс удаляемого элемента:

```
selectbox.remove(0);    // удаление первого элемента
```

В-третьих, можно просто присвоить элементу значение `null`. Этот способ использовался еще до появления DOM:

```
selectbox.options[0] = null;    // удаление первого элемента
```

Чтобы полностью очистить список, нужно перебрать все элементы и удалить каждый из них:

```
function clearSelectbox(selectbox){
    for(var i=0, len=selectbox.options.length; i < len; i++){
        selectbox.remove(0);
    }
}
```

Эта функция просто циклически удаляет первый элемент списка. Так как при этом все остальные элементы автоматически смещаются на одну позицию, в итоге получается пустой список.

Перемещение и переупорядочение элементов списка

До появления DOM переместить элемент из одного списка в другой было непросто. Для этого требовалось удалить элемент из первого списка, создать новый одноименный элемент с таким же значением и добавить его во второй список. В DOM для перемещения элемента в другой список достаточно воспользоваться методом `appendChild()`, который удаляет полученный элемент из родительского элемента и помещает его в конец другого элемента. Например, следующий код перемещает первый элемент исходного списка в конец второго списка:

Листинг SelectboxExample05.htm


```
var selectbox1 = document.getElementById("selLocations1");
var selectbox2 = document.getElementById("selLocations2");

selectbox2.appendChild(selectbox1.options[0]);
```

Перемещение элементов похоже на удаление в том смысле, что свойство `index` каждого элемента при этом сбрасывается.

Переупорядочение элементов выполняется очень похоже, и DOM-методы идеально подходят для этого. Чтобы переместить элемент в конкретное место списка, лучше всего использовать метод `insertBefore()`, хотя переместить элемент в последнюю позицию можно также с помощью метода `appendChild()`. Поднять элемент списка на одну позицию можно следующим образом:

Листинг SelectboxExample06.htm

```
var optionToMove = selectbox.options[1];
selectbox.insertBefore(optionToMove,
                       selectbox.options[optionToMove.index-1]);
```

Этот код выбирает элемент, который нужно переместить, и вставляет его перед предыдущим элементом. Вторую инструкцию можно использовать с любым элементом списка, кроме первого. Переместить элемент на одну позицию вниз можно аналогичным образом:

Листинг SelectboxExample06.htm

```
var optionToMove = selectbox.options[1];
selectbox.insertBefore(optionToMove,
                       selectbox.options[optionToMove.index+2]);
```

Этот код работает со всеми элементами списка, включая последний.

В Internet Explorer 7 на вывод элементов, переупорядоченных с помощью DOM-методов, иногда требуется несколько секунд из-за проблем с перерисовкой.

Сериализация форм

С появлением Ajax (см. главу 21) одним из стандартных требований к веб-приложениям стала поддержка *сериализации форм* (form serialization). Сериализовать форму в JavaScript можно, используя свойства `type`, `name` и `value` ее полей. При отправке данных формы серверу браузер соблюдает определенные правила.

- Имена и значения полей кодируются в формате URL, а их пары разделяются амперсандами.
- Отключенные поля не отправляются.
- Флажки и переключатели отправляются, только если они заданы.
- Кнопки типов `"reset"` и `"button"` не отправляются.
- У полей с множественным выбором отправляется запись для каждого выбранного значения.
- Кнопка отправки отправляется только в том случае, если она была использована для отправки данных формы. Любые элементы `<input>` типа `"image"` обрабатываются так же, как кнопки отправки.
- Значением элемента `<select>` является атрибут `value` выбранного элемента `<option>`. Если у элемента `<option>` нет атрибута `value`, значением является текст элемента `<option>`.

В сериализованное представление формы обычно не включаются никакие поля типа `button`, потому что итоговая строка, вероятно, будет отправлена другим способом. Все остальные правила должны быть соблюдены. Код сериализации формы таков:

Листинг FormSerializationExample01.htm


```
function serialize(form){
```

```
var parts = [],
    field = null,
    i,
    len,
    j,
    optLen,
    option,
    optValue;

for (i=0, len=form.elements.length; i < len; i++){
    field = form.elements[i];

    switch(field.type){
        case "select-one":
        case "select-multiple":

            if (field.name.length){
                for (j=0, optLen=field.options.length; j < optLen; j++){
                    option = field.options[j];
                    if (option.selected){
                        optValue = "";
                        if (option.hasAttribute){
                            optValue = (option.hasAttribute("value") ?
                                        option.value : option.text);
                        } else {
                            optValue =
                               (option.attributes["value"].specified ?
                                option.value : option.text);
                        }
                        parts.push(encodeURIComponent(field.name) +
                                   "=" + encodeURIComponent(optValue));
                    }
                }
            }
            break;

        case undefined:        // коллекция полей
        case "file":           // поле добавления файлов
        case "submit":         // кнопка отправки
        case "reset":          // кнопка сброса
        case "button":         // пользовательская кнопка
            break;

        case "radio":          // переключатель
        case "checkbox":       // флажок
            if (!field.checked){
                break;
            }
            /* переход к варианту, предлагаемому по умолчанию */

        default:
            // поля формы без имен не сериализуются
            if (field.name.length){
                parts.push(encodeURIComponent(field.name) + "=" +
                           encodeURIComponent(field.value));
```

```
                }
            }
        }
        return parts.join("&");
}
```

Функция `serialize()` начинается с определения массива `parts` для хранения частей итоговой строки. Затем цикл `for` перебирает все поля формы, сохраняя их по очереди в переменной `field`. Как только получена ссылка на поле, его тип проверяется с помощью инструкции `switch`. Сложнее всего сериализовать элемент `<select>` с одиночным или множественным выбором. Для этого мы перебираем все элементы списка, добавляя в массив те из них, которые выделены. В списках с одиночным выбором может быть выбран только один элемент, а в списках с множественным выбором — сколько угодно (в том числе ни одного), но код подходит для списков обоих типов, потому что браузер сам контролирует количество выбранных элементов. Обнаружив выбранный элемент, мы должны выяснить, какое значение следует использовать. Если атрибут `value` отсутствует, нужно сериализовать текст элемента, хотя атрибут `value` с пустой строкой допустим. Чтобы проверить его наличие, мы используем метод `hasAttribute()` в браузерах, соответствующих DOM, и свойство `specified` атрибута в Internet Explorer 8 и более ранних версий.

Если форма содержит элемент `<fieldset>`, он доступен в коллекции элементов, но не имеет свойства `type`. Если свойство `type` равно `undefined`, сериализация не требуется. Это верно для всех типов кнопок и полей добавления файлов (поля добавления файлов содержат при отправке файла его содержимое, но их нельзя воспроизвести, так что они обычно опускаются при сериализации). У переключателей и флажков проверяется свойство `checked`, и если оно имеет значение `false`, выполняется выход из инструкции `switch`. Если свойство `checked` имеет значение `true`, мы переходим к разделу, предлагаемому по умолчанию, который кодирует имя и значение поля и добавляет их в массив `parts`. Заметьте, что мы не сериализуем поля формы без имен, чтобы имитировать отправку данных формы браузером. В конце функции вызывается метод `join()` для составления строки с амперсандами между полями.

Функция `serialize()` возвращает строку в формате строки запроса, но ее можно легко адаптировать для сериализации формы в другом формате.

Редактирование форматированного текста

Одной из наиболее востребованных возможностей веб-приложений в свое время была возможность редактировать форматированный текст на веб-странице, что иногда называли WYSIWYG-редактированием (What You See Is What You Get — *что видишь, то и получаешь*). На основе этого функционала, представленного в Internet Explorer и теперь поддерживаемого в Opera, Safari, Chrome и Firefox, возник стандарт де-факто, который, однако, не описан ни в какой спецификации. Методика основана на добавлении в страницу встроенного фрейма (`iframe`), содержащего

пустой HTML-файл. С помощью свойства `designMode` можно включить редактирование этого пустого документа, то есть HTML-кода элемента `<body>` страницы. Свойство `designMode` может иметь значение `"off"` (по умолчанию) или `"on"`. Если оно равно `"on"`, можно редактировать весь документ так же, как в текстовом редакторе, используя сочетания клавиш, которые изменяют начертание на полужирное, курсивное и т. д.

В качестве источника встроенного фрейма можно использовать совсем простую пустую HTML-страницу, например:

```
<!DOCTYPE html>
<html>
    <head>
        <title>Blank Page for Rich Text Editing</title>
    </head>
    <body>
    </body>
</html>
```

Эта страница загружается во встроенный фрейм как любая другая. Чтобы ее можно было редактировать, нужно присвоить свойству `designMode` значение `"on"`, но это возможно только после полной загрузки документа. Обработчик события `load` страницы-контейнера подходит как нельзя лучше:

```
<iframe name="richedit" style="height: 100px; width: 100px" src="blank.htm">
</iframe>

<script type="text/javascript">
EventUtil.addHandler(window, "load", function(){
    frames["richedit"].document.designMode = "on";
});
</script>
```

Как только этот код загрузится, вы увидите некоторое подобие текстового поля. Оно по умолчанию будет иметь такой же стиль, что и любая веб-страница, хотя его можно изменить, применив к пустой странице CSS-стиль.

Атрибут contenteditable

Другой способ работы с форматированным текстом, также впервые реализованный в Internet Explorer, включает использование специального атрибута `contenteditable`. Применив его к любому элементу на странице, можно разрешить редактирование этого элемента пользователем. Многие предпочитают этот подход предыдущему, потому что он не требует использования встроенного фрейма, пустой страницы и JavaScript-кода. Вместо этого можно просто добавить атрибут к элементу:

```
<div class="editable" id="richedit" contenteditable></div>
```

Любой текст, уже содержащийся в элементе, при этом автоматически становится редактируемым подобно элементу `<textarea>`. Включать и отключать режим редактирования можно динамически, используя свойство `contentEditable` элемента:

```
var div = document.getElementById("richedit");
richedit.contentEditable = "true";
```

Свойство `contentEditable` поддерживает три значения: `"true"` включает режим редактирования, `"false"` отключает, а `"inherit"` указывает, что нужно наследовать параметр родительского элемента (требуется потому, что в элементе со свойством `contenteditable` можно создавать другие элементы и уничтожать их). Атрибут `contentEditable` поддерживается в Internet Explorer, Firefox, Chrome, Safari и Opera. Что касается мобильных устройств, то он поддерживается в Safari для iOS 5+ и WebKit для Android 3+.

Работа с форматированным текстом

Основным механизмом взаимодействия с редактором форматированного текста является метод `document.execCommand()`, который выполняет для документа именованные команды, поддерживая большинство изменений формата. Метод `document.execCommand()` имеет три аргумента: имя команды, которую нужно выполнить, логическое значение, указывающее, должен ли браузер предоставить пользовательский интерфейс для команды, и значение, необходимое команде для работы (или `null`, если оно не требуется). Второй аргумент в кроссбраузерном коде всегда должен быть равен `false`, потому что Firefox генерирует ошибку, если он равен `true`.

Каждый браузер поддерживает свой набор команд. Команды с наиболее широкой поддержкой указаны в таблице.

Команда	Значение (третий аргумент)	Описание
backcolor	Строка цвета	Задает фоновый цвет документа
bold	null	Включает и отключает полужирное начертание для выделенного текста
copy	null	Копирует выделенный текст в буфер обмена
createlink	Строка URL-адреса	Преобразует выделенный текст в ссылку с указанным URL-адресом
cut	null	Вырезает выделенный текст в буфер обмена
delete	null	Удаляет выделенный текст
fontname	Название шрифта	Изменяет шрифт для выделенного текста

Продолжение

Команда	Значение (третий аргумент)	Описание
fontsize	Число от 1 до 7	Изменяет размер шрифта для выделенного текста
forecolor	Строка цвета	Изменяет цвет выделенного текста
formatblock	HTML-элемент, в который должен быть заключен блок, например <h1>	Форматирует все текстовое поле с выделенным текстом, используя указанный HTML-элемент
indent	null	Задает отступ для текста
inserthorizontalrule	null	Вставляет элемент <hr> в позиции курсора
insertimage	URL-адрес изображения	Вставляет изображение в позиции курсора
insertorderedlist	null	Вставляет элемент в позиции курсора
insertparagraph	null	Вставляет элемент <p> в позиции курсора
insertunorderedlist	null	Вставляет элемент в позиции курсора
italic	null	Включает и отключает курсивное начертание для выделенного текста
justifycenter	null	Центрирует блок текста, в котором находится курсор
justifyleft	null	Выравнивает по левому краю блок текста, в котором находится курсор
outdent	null	Отменяет отступ для текста
paste	null	Вставляет выделенный текст из буфера обмена
removeformat	null	Отменяет форматирование блока, в котором находится курсор. Эта команда противоположна команде formatblock
selectall	null	Выделяет весь текст в документе
underline	null	Включает и отключает подчеркивание выделенного текста
unlink	null	Удаляет текстовую ссылку. Эта команда противоположна команде createlink

Реализация команд для работы с буфером обмена во многом зависит от браузера. Opera не поддерживает никакие из них, а в Firefox они по умолчанию отключены (чтобы включить их, нужно изменить параметры пользователя). В Safari и Chrome

реализованы команды cut и copy, но не paste. Хотя эти команды недоступны через метод document.execCommand(), соответствующие сочетания клавиш все же работают.

С помощью этих команд можно в любое время изменить вид форматированного текста во встроенном фрейме, например:

Листинг RichTextEditingExample01.htm


```
// переключение полужирного начертания текста во встроенном фрейме
frames["richedit"].document.execCommand("bold", false, null);

// переключение курсивного начертания текста во встроенном фрейме
frames["richedit"].document.execCommand("italic", false, null);

// создание ссылки на сайт www.wrox.com во встроенном фрейме
frames["richedit"].document.execCommand("createlink", false,
                                        "http://www.wrox.com");

// создание заголовка первого уровня во встроенном фрейме
frames["richedit"].document.execCommand("formatblock", false, "<h1>");
```

С помощью этих же команд можно форматировать текст в разделе contenteditable страницы, но тогда вместо встроенного фрейма нужно использовать объект document текущего окна:

Листинг RichTextEditingExample01.htm

```
// переключение полужирного начертания текста
document.execCommand("bold", false, null);

// переключение курсивного начертания текста
document.execCommand("italic", false, null);

// создание ссылки на сайт www.wrox.com
document.execCommand("createlink", false, "http://www.wrox.com");

// создание заголовка первого уровня
document.execCommand("formatblock", false, "<h1>");
```

Даже если команда поддерживается во всех браузерах, HTML-код, который она генерирует, может сильно различаться. Например, команда bold применяет к тексту элемент в Internet Explorer и Opera, элемент <b> в Safari и Chrome, элемент в Firefox. Из-за различий в реализации команд и в преобразовании HTML-кода свойством innerHTML полагаться на единообразие HTML-кода, генерируемого редактором форматированного текста, не следует.

Есть несколько других методов, связанных с командами. Метод queryCommandEnabled() определяет, можно ли выполнить команду при текущем выделенном тексте или положении курсора. Он принимает имя проверяемой команды и возвращает true, если она допустима при текущем состоянии области редактирования, или false в противном случае, например:

```
var result = frames["richedit"].document.queryCommandEnabled("bold");
```

Если команду "bold" можно выполнить для текущего выделенного текста, этот код возвратит true. Метод queryCommandEnabled() не указывает, разрешено ли выполнение команды, а только сообщает, допустимо ли это при текущем выделенном тексте. Например, в Firefox вызов queryCommandEnabled("cut") возвращает true, хотя по умолчанию эта команда запрещена.

С помощью метода queryCommandState() можно узнать, была ли выполнена конкретная команда для выделенного текста. Например, следующий код определяет, является ли текущий выделенный текст полужирным:

Листинг RichTextEditingExample01.htm

```
var isBold = frames["richedit"].document.queryCommandState("bold");
```

Если команда "bold" была применена к выделенному тексту, этот код возвратит true. Так редакторы форматированного текста обновляют кнопки задания полужирного и курсивного начертания и т. п.

Наконец, метод queryCommandValue() возвращает значение, с которым была выполнена команда (третий аргумент метода execCommand). Например, для диапазона текста, к которому была применена команда "fontsize" со значением 7, следующий код возвращает "7":

Листинг RichTextEditingExample01.htm

```
var fontSize = frames["richedit"].document.queryCommandValue("fontsize");
```

С помощью этого метода можно определить, как команда была выполнена для выделенного текста, чтобы выяснить, допустима ли следующая команда.

Выделение форматированного текста

Получить текст, выделенный в редакторе форматированного текста, можно с помощью метода getSelection() встроенного фрейма. Этот метод, доступный для объектов document и window, возвращает объект Selection, у которого есть следующие свойства:

- anchorNode — узел, в котором начинается выделенная область;
- anchorOffset — количество знаков в узле anchorNode, пропущенных перед началом выделенной области;
- focusNode — узел, в котором завершается выделенная область;
- focusOffset — количество выделенных символов в узле focusNode;
- isCollapsed — логическое значение, указывающее, совпадают ли начало и конец выделенной области;
- rangeCount — количество DOM-диапазонов в выделенной области.

Свойства объекта `Selection` не особо полезны. К счастью, у него также доступны следующие методы, позволяющие получить дополнительные сведения о выделенной области и изменить ее:

- `addRange(`*диапазон*`)` — добавляет указанный DOM-диапазон в выделенную область;
- `collapse(`*узел*, *смещение*`)` — свертывает выделенную область к указанному смещению в указанном узле;
- `collapseToEnd()` — свертывает выделенную область к ее концу;
- `collapseToStart()` — свертывает выделенную область к ее началу;
- `containsNode(`*узел*`)` — определяет, содержится ли указанный узел в выделенной области;
- `deleteFromDocument()` — удаляет выделенный текст из документа (то же самое, что и вызов `execCommand("delete", false, null)`);
- `extend(`*узел*, *смещение*`)` — расширяет выделенную область, перемещая `focusNode` и `focusOffset` к указанным значениям;
- `getRangeAt(`*индекс*`)` — возвращает DOM-диапазон по указанному индексу в выделенной области;
- `removeAllRanges()` — удаляет все DOM-диапазоны из выделенной области (по сути, выделенная область при этом удаляется, потому что она должна содержать хотя бы один диапазон);
- `removeRange(`*диапазон*`)` — удаляет указанный DOM-диапазон из выделенной области;
- `selectAllChildren(`*узел*`)` — отменяет выделение и выделяет все дочерние узлы указанного узла;
- `toString()` — возвращает текстовое содержимое выделенной области.

Методы объекта `Selection` очень эффективны и широко используют DOM-диапазоны для управления выделением (см. главу 12). Доступ к DOM-диапазонам обеспечивает даже больший контроль над форматированием текста, чем метод `execCommand()`, позволяя работать непосредственно с DOM-структурой выделенного текста, например:

Листинг RichTextEditingExample01.htm


```
var selection = frames["richedit"].getSelection();

// получение выделенного текста
var selectedText = selection.toString();

// получение диапазона, представляющего выделенную область
var range = selection.getRangeAt(0);
```

```
// закрашивание фона выделенного текста
var span = frames["richedit"].document.createElement("span");
span.style.backgroundColor = "yellow";
range.surroundContents(span);
```

Этот код закрашивает фон выделенного текста желтым цветом. Для этого метод `surroundContents()` заключает DOM-диапазон выделенной области в элемент `<span>` с желтым фоном.

Метод `getSelection()` определен в HTML5 и реализован в Internet Explorer 9, Firefox, Safari, Chrome и Opera 8. В Firefox 3.6+ вызов `document.getSelection()` ошибочно возвращает строку из-за проблем с поддержкой унаследованного кода. Вместо него для получения объекта `Selection` в Firefox 3.6+ можно использовать метод `window.getSelection()`. В Firefox 8 ошибка исправлена и метод `document.getSelection()` возвращает то же значение, что и `window.getSelection()`.

Internet Explorer 8 и более ранних версий не поддерживают DOM-диапазоны, но позволяют взаимодействовать с выделенным текстом с помощью фирменного объекта `selection`, являющегося свойством объекта `document`. Чтобы получить текст, выделенный в редакторе форматированного текста, нужно сначала создать текстовый диапазон (см. главу 12), а затем прочитать его свойство `text`:

```
var range = frames["richedit"].document.selection.createRange();
var selectedText = range.text;
```

Манипулировать HTML-кодом с помощью текстовых диапазонов в Internet Explorer не так безопасно, как использовать DOM-диапазоны, хотя и возможно. Например, для закрашивания фона выделенной области, как в предыдущем фрагменте, можно использовать свойство `htmlText` в сочетании с методом `pasteHTML()`:

```
var range = frames["richedit"].document.selection.createRange();
range.pasteHTML("<span style=\"background-color:yellow\">" +
                range.htmlText + "</span>");
```

Этот код получает HTML-код текущей выделенной области с помощью свойства `htmlText`, а затем заключает его в элемент `<span>` и вставляет обратно методом `pasteHTML()`.

Форматированный текст в формах

Поскольку редактирование форматированного текста выполняется с помощью встроенного фрейма или элемента `contenteditable`, а не элемента управления формы, редактор форматированного текста технически не является частью формы. Это означает, что для отправки HTML-кода серверу нужно извлечь его вручную и отправить самостоятельно. Для этого обычно используют скрытое поле формы, обновляя его HTML-кодом из встроенного фрейма или элемента `contenteditable`. Непосредственно перед отправкой формы HTML-код извлекают

из встроенного фрейма или элемента и вставляют в скрытое поле. Например, если используется встроенный фрейм, в обработчике события `submit` формы можно сделать следующее:

Листинг RichTextEditingExample01.htm


```
EventUtil.addHandler(form, "submit", function(event){
    event = EventUtil.getEvent(event);
    var target = EventUtil.getTarget(event);

    target.elements["comments"].value =
        frames["richedit"].document.body.innerHTML;
});
```

Здесь мы получаем HTML-код из встроенного фрейма с помощью свойства `innerHTML` тела документа и вставляем его в поле формы с именем `"comments"`, заполняя поле непосредственно перед отправкой данных формы. Не забудьте сделать это, если вы отправляете форму вручную методом `submit()`. С элементом `contenteditable` можно поступить аналогичным образом:

```
EventUtil.addHandler(form, "submit", function(event){
    event = EventUtil.getEvent(event);
    var target = EventUtil.getTarget(event);

    target.elements["comments"].value =
        document.getElementById("richedit").innerHTML;
});
```

Резюме

В отличие от HTML- и веб-приложений, веб-формы за время их существования изменились незначительно. Используя в JavaScript-коде свойства, методы и события форм и их полей, можно расширять возможности форм и делать работу с ними более удобной. Перечислим некоторые из концепций, рассмотренных в этой главе.

- Используя различные стандартные и нестандартные методы, можно выделять содержимое текстовых полей полностью или частично.
- Для взаимодействия с выделенным текстом во всех браузерах имеются стандартные средства, первоначально реализованные в Firefox.
- Прослушивая события клавиатуры и проверяя вводимые символы, можно блокировать добавление определенных символов в текстовое поле.

Все браузеры, кроме Opera, поддерживают события буфера обмена, включая `copy`, `cut` и `paste`, но реализации этих событий в разных браузерах различна.

- Данные в буфере обмена доступны посредством JavaScript-кода в Internet Explorer, Firefox, Chrome и Safari, но не в Opera.

- Способы работы с буфером обмена в Internet Explorer, Chrome и Safari различаются.
- В Firefox, Safari и Chrome читать данные из буфера обмена можно только при обработке события `paste`, а в Internet Explorer нет такого ограничения.
- В Firefox, Safari и Chrome данные в буфере обмена доступны только при обработке событий буфера обмена, а в Internet Explorer — в любое время.

Заблокировав событие `paste` для буфера обмена, можно предотвратить вставку определенных символов в текстовое поле.

Благодаря DOM управлять списками стало гораздо проще. Используя стандартные DOM-приемы, можно добавлять элементы в списки, удалять их, изменять их порядок и перемещать из одного списка в другой.

Для редактирования форматированного текста можно использовать встроенный фрейм с пустым HTML-документом. Чтобы включить для страницы такой же режим редактирования, как в текстовом редакторе, нужно присвоить свойству `designMode` документа значение `"on"` или задать для элемента атрибут `contenteditable`. По умолчанию во время редактирования можно переключать стили шрифтов, такие как полужирный и курсивный, работать с буфером обмена. Для доступа к некоторым из этих возможностей средствами JavaScript-кода можно использовать метод `execCommand()`, а получить сведения о выделенном тексте можно с помощью методов `queryCommandEnabled()`, `queryCommandState()` и `queryCommandValue()`. Поле формы для редактора форматированного текста не создается, поэтому для отправки такого текста серверу нужно предварительно скопировать его HTML-код из встроенного фрейма или элемента `contenteditable` в поле формы.

15 Рисование на холсте

- Общие сведения об элементе <canvas>
- Рисование простой двухмерной графики
- Рисование трехмерной графики с помощью WebGL

Вероятно, наиболее популярной новинкой в HTML5 является элемент `<canvas>`, который определяет область страницы для динамического рисования средствами JavaScript. Первоначально он был предложен компанией Apple для использования в мини-приложениях, а вскоре был добавлен в HTML5 и реализован в браузерах. В той или иной степени элемент `<canvas>` поддерживают Internet Explorer 9+, Firefox 1.5+, Safari 2+, Opera 9+, Chrome, Safari для iOS и WebKit для Android.

Подобно другим компонентам среды браузера, `<canvas>` состоит из нескольких API, поддержка которых зависит от браузера. Разработчикам доступен двухмерный контекст с базовыми средствами рисования, а также предварительная версия трехмерного контекста, который называется WebGL-контекстом. Новейшие версии браузеров поддерживают двухмерный контекст и текстовый API, а WebGL-контекст пока остается экспериментальным, хотя его поддержка постепенно расширяется. Firefox 4+ и Chrome поддерживают ранние версии спецификации WebGL, но в старых операционных системах, таких как Windows XP, использовать WebGL-контекст невозможно даже в этих браузерах из-за отсутствия нужных графических драйверов.

Основы работы с элементом <canvas>

Для использования элемента `<canvas>` нужно как минимум задать его атрибуты `width` и `height`, указывающие ширину и высоту рисунка. Контент между его открывающим

и закрывающим тегами добавляется для страховки и выводится на экран, только если элемент `<canvas>` не поддерживается, например:

```
<canvas id="drawing" width="200" height="200">A drawing of something.</canvas>
```

Как и у других элементов, атрибуты `width` и `height` доступны в качестве свойств соответствующего объекта, которые в любой момент можно изменить. Весь элемент можно стилизовать средствами CSS, а пока это не сделано и на холсте ничего не нарисовано, он остается невидимым.

Чтобы начать рисовать на холсте, нужно получить контекст рисования методом `getContext()`, который принимает имя контекста. Например, передав ему значение `"2d"`, можно получить объект двухмерного контекста:

```
var drawing = document.getElementById("drawing");

// проверка полной поддержки элемента <canvas>
if (drawing.getContext){

    var context = drawing.getContext("2d");

    // другой код
}
```

Перед использованием элемента `<canvas>` важно проверить наличие метода `getContext()`. Для элементов, которые не входят официально в HTML, некоторые браузеры создают объекты, предлагаемые по умолчанию, и тогда метод `getContext()` становится недоступен, хотя объект `drawing` содержит действительную ссылку на элемент.

Изображения, созданные в элементе `<canvas>`, можно экспортировать методом `toDataURL()`. Он принимает целевой формат изображения в виде MIME-типа и работает независимо от того, какой контекст был использован для создания изображения. Например, чтобы возвратить изображение с холста в формате PNG, используйте следующий код:

Листинг 2DDataUrlExample01.htm


```
var drawing = document.getElementById("drawing");

// проверка полной поддержки элемента <canvas>
if (drawing.getContext){

    // получение URI изображения
    var imgURI = drawing.toDataURL("image/png");

    // вывод изображения на экран
    var image = document.createElement("img");
    image.src = imgURI;
    document.body.appendChild(image);
}
```

По умолчанию браузеры кодируют изображения в формате PNG. В Firefox и Opera можно также использовать формат JPEG (тип `"image/jpeg"`). Он доступен в более поздних версиях браузеров, включая Internet Explorer 9, Firefox 3.5 и Opera 10.

Если на холсте нарисовано изображение из другого домена, метод toDataURL() генерирует ошибку. Подробности мы обсудим немного позже.

Двухмерный контекст

Двухмерный контекст рисования предоставляет методы для рисования простых двухмерных фигур, таких как прямоугольники, дуги и пути. Началом координат в двухмерном контексте является верхний левый угол элемента `<canvas>`, который считается точкой (0,0). Значение x увеличивается слева направо, а y — сверху вниз. По умолчанию свойства `width` и `height` указывают, сколько пикселей доступно в каждом направлении.

Заливка и рисование контура

Для двухмерного контекста доступны две основные операции рисования: заливка и рисование контура. При заливке фигура автоматически заполняется указанным содержимым с указанным стилем (цвет, градиент или изображение), а при рисовании контура окрашиваются только контуры фигуры. У большинства операций в двухмерном контексте есть варианты с заливкой и рисованием контура, которые настраиваются с помощью свойств `fillStyle` и `strokeStyle`.

Каждое из этих свойств может быть строкой, градиентом или узором и имеет по умолчанию значение `"#000000"`. Если используется строка, она должна определять цвет в одном из форматов CSS: по имени, шестнадцатеричному коду, в формате `rgb`, `rgba`, `hsl` или `hsla`. Вот пример:

```
var drawing = document.getElementById("drawing");

    // проверка полной поддержки элемента <canvas>
    if (drawing.getContext){

        var context = drawing.getContext("2d");
        context.strokeStyle = "red";
        context.fillStyle = "#0000ff";
}
```

Этот код присваивает свойству `strokeStyle` значение `"red"` (именованный CSS-цвет), а свойству `fillStyle` — `"#0000ff"` (синий цвет). Все последующие операции заливки и рисования контура будут использовать эти значения, пока они не изменятся. Этим свойствам также можно назначить градиент или узор, о чем мы поговорим позже.

Рисование прямоугольников

Прямоугольник — единственная фигура, которую можно нарисовать непосредственно в двухмерном контексте. Для работы с прямоугольниками можно использовать методы `fillRect()`, `strokeRect()` и `clearRect()`. Каждый из них принимает четыре аргумента: координаты x и y, ширину и высоту прямоугольника. Значения аргументов измеряются в пикселях.

Метод `fillRect()` рисует на холсте прямоугольник, залитый цветом, указанным с помощью свойства `fillStyle`, например:

Листинг 2DFillRectExample01.htm


```
var drawing = document.getElementById("drawing");

// проверка полной поддержки элемента <canvas>
if (drawing.getContext){

    var context = drawing.getContext("2d");

    /*
     * Код основан на документации Mozilla:
     * http://developer.mozilla.org/en/docs/Canvas_tutorial:Basic_usage
     */

    // рисование красного квадрата
    context.fillStyle = "#ff0000";
    context.fillRect(10, 10, 50, 50);

    // рисование синего полупрозрачного квадрата
    context.fillStyle = "rgba(0,0,255,0.5)";
    context.fillRect(30, 30, 50, 50);
}
```

Этот код сначала назначает свойству `fillStyle` красный цвет и рисует квадрат с левым верхним углом в точке (10,10) и стороной, равной 50 пикселей. Затем он назначает свойству `fillStyle` синий полупрозрачный цвет методом `rgba()` и рисует второй квадрат, который перекрывает первый. В результате красный квадрат виден через синий (рис. 15.1).

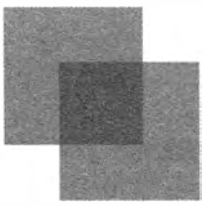

Рис. 15.1

Метод `strokeRect()` рисует контур прямоугольника, используя цвет, указанный с помощью свойства `strokeStyle`, например:

Листинг 2DStrokeRectExample01.htm

```
var drawing = document.getElementById("drawing");

// проверка полной поддержки элемента <canvas>
if (drawing.getContext){

    var context = drawing.getContext("2d");

    /*
     * Код основан на документации Mozilla:
     * http://developer.mozilla.org/en/docs/Canvas_tutorial:Basic_usage
     */

    // рисование красного контура квадрата
    context.strokeStyle = "#ff0000";
    context.strokeRect(10, 10, 50, 50);

    // рисование синего полупрозрачного контура квадрата
    context.strokeStyle = "rgba(0,0,255,0.5)";
    context.strokeRect(30, 30, 50, 50);
}
```

Этот код рисует только контуры перекрывающихся квадратов (рис. 15.2).

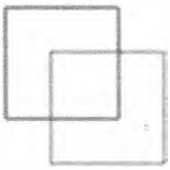

Рис. 15.2

Толщина контура определяется свойством lineWidth, которому можно присвоить любое целое число. Свойство lineCap описывает концы линий (доступные значения: «butt» (срез), «round» (круг), «square» (квадрат)), а lineJoin указывает, как должны соединяться линии (доступные значения: «round» (закругление), «bevel» (скос), «miter» (клин)).

Метод `clearRect()` очищает область холста и используется, если нужно сделать прямоугольную часть контекста рисования прозрачной. Рисуя фигуры и очищая их области, можно создавать интересные эффекты, например, вырезать фрагменты других фигур:

Листинг 2DClearRectExample01.htm


```
var drawing = document.getElementById("drawing");

// проверка полной поддержки элемента <canvas>
```

```
if (drawing.getContext){

    var context = drawing.getContext("2d");

    /*
     * Код основан на документации Mozilla:
     * http://developer.mozilla.org/en/docs/Canvas_tutorial:Basic_usage
     */

    // рисование красного квадрата
    context.fillStyle = "#ff0000";
    context.fillRect(10, 10, 50, 50);

    // рисование синего полупрозрачного квадрата
    context.fillStyle = "rgba(0,0,255,0.5)";
    context.fillRect(30, 30, 50, 50);

    // очистка квадрата в области наложения двух квадратов
    context.clearRect(40, 40, 10, 10);
}
```

Этот код сначала рисует два перекрывающихся квадрата с заливкой, а затем очищает в области их наложения меньший квадрат (рис. 15.3).

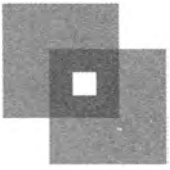

Рис. 15.3

Рисование путей

Двухмерный контекст рисования поддерживает несколько методов рисования путей, позволяющих создавать сложные фигуры и линии. Чтобы приступить к созданию пути, нужно вызвать метод `beginPath()`, после чего для продолжения пути можно использовать следующие методы:

- `arc(`*x*, *y*, *радиус*, *начальныйУгол*, *конечныйУгол*, *противЧасовойСтрелки*`)` — рисует дугу указанного радиуса с центром в точке (*x*, *y*) между начальным и конечным углами (в радианах). Последним аргументом является логическое значение, указывающее, следует ли отсчитывать углы против часовой стрелки или по ней.
- `arcTo(`*x1*, *y1*, *x2*, *y2*, *радиус*`)` — рисует дугу указанного радиуса от последней точки до (*x2*, *y2*), проходящую через (*x1*, *y1*).
- `bezierCurveTo(`*c1x*, *c1y*, *c2x*, *c2y*, *x*, *y*`)` — рисует кривую от последней точки до точки (*x*, *y*), используя контрольные точки (*c1x*, *c1y*) и (*c2x*, *c2y*).

- `lineTo(`x, y`)` — рисует линию от последней точки до точки (x, y).
- `moveTo(`x, y`)` — перемещает курсор в точку (x, y) без рисования линии.
- `quadraticCurveTo(`cx, cy, x, y`)` — рисует квадратичную кривую от последней точки до точки (x, y), используя контрольную точку (cx, cy).
- `rect(`x, y, *ширина*, *высота*`)` — рисует прямоугольник с левым верхним углом в точке (x, y) и указанными значениями ширины и высоты. Этот метод отличается от методов `strokeRect()` и `fillRect()` тем, что создает путь, а не отдельную фигуру.

Как только путь создан, возможны несколько вариантов. Можно вызвать метод `closePath()`, чтобы провести линию к началу пути. Если путь уже завершен и вы хотите залить его, используя стиль `fillStyle`, вызовите метод `fill()`. Можно также отобразить путь без заливки, вызвав метод `stroke()`, при этом будет задействован стиль `strokeStyle`. Последний вариант — создать на основе пути область отсечения с помощью метода `clip()`.

Рассмотрим, например, следующий код, который рисует часы без чисел:

Листинг 2DPathExample01.htm


```
var drawing = document.getElementById("drawing");

// проверка полной поддержки элемента <canvas>
if (drawing.getContext){

    var context = drawing.getContext("2d");

    // начало пути
    context.beginPath();

    // рисование внешней окружности
    context.arc(100, 100, 99, 0, 2 * Math.PI, false);

    // рисование внутренней окружности
    context.moveTo(194, 100);
    context.arc(100, 100, 94, 0, 2 * Math.PI, false);

    // рисование минутной стрелки
    context.moveTo(100, 100);
    context.lineTo(100, 15);

    // рисование часовой стрелки
    context.moveTo(100, 100);
    context.lineTo(35, 100);

    // вывод на экран пути без заливки
    context.stroke();
}
```

Чтобы создать границу часов, мы рисуем две концентрические окружности методом `arc()`. Внешняя окружность имеет радиус 99 пикселей и центр в точке (100,100), которая совпадает с центром холста. Чтобы нарисовать полную окружность, мы

проводим дугу от 0 до 2π радиан, используя в качестве числа π значение `Math.PI`. Перед рисованием внутренней окружности нужно переместить путь в точку, которая будет находиться на ней, чтобы предотвратить рисование лишней линии. Во втором вызове `arc()` для создания эффекта границы используется немного меньший радиус. После этого методами `moveTo()` и `lineTo()` мы рисуем минутную и часовую стрелки. Наконец, метод `stroke()` придает часам вид, показанный на рис. 15.4.

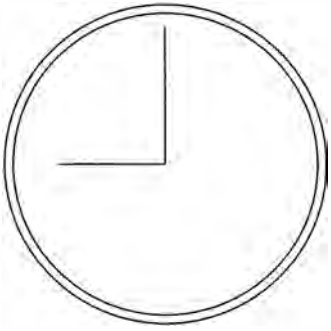

Рис. 15.4

Пути — основной механизм рисования в двухмерном контексте, потому что они обеспечивают более точный контроль над фигурами. Чтобы упростить работу с ними, можно использовать метод `isPointInPath()`, который принимает координаты точки по осям x и y. С помощью этого метода можно узнать, содержит ли путь конкретную точку:

```
if (context.isPointInPath(100, 100)){
    alert("Point (100, 100) is in the path.");
}
```

API путей достаточно надежен и позволяет создавать весьма сложные изображения с разнообразными заливками, контурами и т. д.

Рисование текста

Поскольку рисунки часто должны содержать не только графику, но и текст, двухмерный контекст рисования предоставляет методы `fillText()` и `strokeText()` для рисования текста. Каждый из них принимает четыре аргумента: строку, которую нужно нарисовать, координаты x и y и необязательное значение максимальной ширины текста в пикселях. Оба метода рисуют текст, используя значения трех свойств:

- `font` — начертание, размер и семейство шрифта в формате CSS, например `"10px Arial"`.
- `textAlign` — способ выравнивания текста. Возможные значения: `"start"`, `"end"`, `"left"`, `"right"` и `"center"`. Рекомендуется использовать значения `"start"` и `"end"`

вместо "left" и "right", потому что они правильно отражают суть дела в языках с написанием как слева направо, так и справа налево.

- textBaseline — базовая линия текста. Возможные значения: "top", "hanging", "middle", "alphabetic", "ideographic" и "bottom".

У этих свойств есть значения, предлагаемые по умолчанию, так что задавать их каждый раз при рисовании текста не требуется. Метод fillText() использует при рисовании текста свойство fillStyle, а метод strokeText() — свойство strokeStyle. Вероятно, в большинстве случаев вы будете использовать метод fillText(), так как он имитирует обычное отображение текста на веб-страницах. Например, следующий код выводит число 12 в верхней части часов, созданных в предыдущем разделе:

Листинг 2DTextExample01.htm


```
context.font = "bold 14px Arial";
context.textAlign = "center";
context.textBaseline = "middle";
context.fillText("12", 100, 20);
```

Итоговое изображение показано на рис. 15.5.

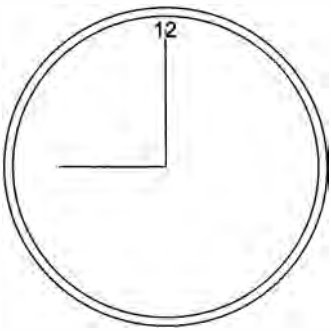

Рис. 15.5

Поскольку свойство textAlign имеет значение "center", а textBaseline — "middle", координаты (100, 20) указывают центр текста по горизонтали и его среднюю линию по вертикали. Если бы свойство textAlign имело значение "start" или "end", координата *x* представляла бы в языке с письмом слева направо начало или конец текста соответственно:

Листинг 2DTextExample02.htm

```
// выравнивание по центру
context.font = "bold 14px Arial";
context.textAlign = "center";
context.textBaseline = "middle";
context.fillText("12", 100, 20);

// выравнивание по началу текста
```

```
context.textAlign = "start";
context.fillText("12", 100, 40);

// выравнивание по концу текста
context.textAlign = "end";
context.fillText("12", 100, 60);
```

Этот код трижды отображает строку `"12"` с одним и тем же значением x, но с тремя разными значениями координаты y и свойства `textAlign`. Итоговое изображение показано на рис. 15.6.

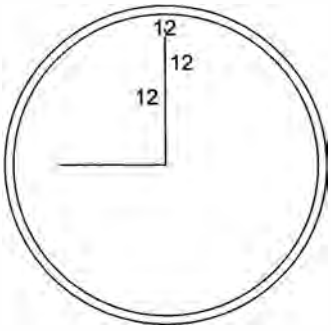

Рис. 15.6

Минутная стрелка часов располагается вертикально, так что выравнивание текста очевидно. Настроить выравнивание текста по вертикали можно с помощью свойства `textBaseline`. Если оно имеет значение `"top"` или `"bottom"`, координата y определяет соответственно верх или низ текста, а значения `"hanging"`, `"alphabetic"` и `"ideographic"` указывают специфические координаты базовой линии шрифта.

Чтобы рисовать текст было проще, особенно если требуется отобразить его в определенной области, можно определить его размеры методом `measureText()`. Он принимает текст, который нужно нарисовать, и возвращает объект `TextMetrics`. В настоящее время этот объект имеет единственное свойство `width`, но предполагается, что к нему будут добавлены и другие параметры.

Для вычисления размеров текста метод `measureText()` использует текущие значения `font`, `textAlign` и `textBaseline`. Предположим, например, что нам нужно вместить текст `"Hellow world!"` в прямоугольник шириной 140 пикселей. Для этого следующий код задает первоначальный размер шрифта (100 пикселей) и уменьшает его, пока текст не поместится в прямоугольник:

Листинг 2DTextExample03.htm


```
var fontSize = 100;
context.font = fontSize + "px Arial";

while(context.measureText("Hello world!").width > 140){
    fontSize--;
```

```
    context.font = fontSize + "px Arial";
}

context.fillText("Hello world!", 10, 10);
context.fillText("Font size is " + fontSize + "px", 10, 50);
```

У методов `fillText()` и `strokeText()` есть также четвертый аргумент, максимальная ширина текста. Он не обязателен и пока поддерживается не во всех браузерах (впервые он появился в Firefox 4). Если этот аргумент указан и строка, переданная в метод `fillText()` или `strokeText()`, превышает максимальную ширину, текст сжимается по горизонтали, при этом его высота не меняется (рис. 15.7).

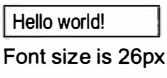

Рис. 15.7

Из-за сложности рисования текста соответствующий API пока доступен в браузерах не полностью.

Преобразования

Путем преобразования контекста можно манипулировать изображениями на холсте. Двухмерный контекст рисования поддерживает все базовые преобразования. При создании контекста рисования матрица преобразования инициализируется значениями, предлагаемыми по умолчанию, при которых все операции рисования выполняются без изменений. Если применить преобразование к контексту рисования, операции будут выполняться с другой матрицей, а потому и их результаты окажутся другими.

Матрицу преобразования можно настроить с помощью следующих методов:

- `rotate(`*угол*`)` — поворачивает изображение вокруг начала координат на указанный угол в радианах.
- `scale(`*масштабX*, *масштабY*`)` — масштабирует изображение по осям x и y. По умолчанию оба аргумента равны 1.0.
- `translate(`x, y`)` — перемещает начало координат в точку (x, y).
- `transform(`$m1\_1, m1\_2, m2\_1, m2\_2, dx, dy$`)` — умножает матрицу преобразования на следующую матрицу:

```
m1_1 m1_2 dx
m2_1 m2_2 dy
0    0    1
```

- `setTransform(`$m1\_1, m1\_2, m2\_1, m2\_2, dx, dy$`)` — сбрасывает матрицу преобразования, а затем вызывает метод.

Преобразования могут быть сколь угодно простыми или сложными. Скажем, в примере с часами для рисования стрелок можно переместить начало координат в центр часов:

Листинг 2DTransformExample01.htm


```
var drawing = document.getElementById("drawing");

// проверка полной поддержки элемента <canvas>
if (drawing.getContext){

    var context = drawing.getContext("2d");

    // начало пути
    context.beginPath();

    // рисование внешней окружности
    context.arc(100, 100, 99, 0, 2 * Math.PI, false);

    // рисование внутренней окружности
    context.moveTo(194, 100);
    context.arc(100, 100, 94, 0, 2 * Math.PI, false);

    // перенос начала координат в центр часов
    context.translate(100, 100);

    // рисование минутной стрелки
    context.moveTo(0, 0);
    context.lineTo(0, -85);

    // рисование часовой стрелки
    context.moveTo(0, 0);
    context.lineTo(-65, 0);

    // отображение пути без заливки
    context.stroke();
}
```

После переноса начала координат в центр часов с координатами (100, 100) нарисовать стрелки стало еще проще, потому что все координаты теперь рассчитываются относительно точки (0,0). Можно не ограничиваться этим и повернуть стрелки методом `rotate()`:

Листинг 2DTransformExample01.htm


```
var drawing = document.getElementById("drawing");

// проверка полной поддержки элемента <canvas>
if (drawing.getContext){

    var context = drawing.getContext("2d");

    // начало пути
    context.beginPath();
```

```
    // рисование внешней окружности
    context.arc(100, 100, 99, 0, 2 * Math.PI, false);

    // рисование внутренней окружности
    context.moveTo(194, 100);
    context.arc(100, 100, 94, 0, 2 * Math.PI, false);

    // перенос начала координат в центр часов
    context.translate(100, 100);

    // поворот стрелок
    context.rotate(1);

    // рисование минутной стрелки
    context.moveTo(0, 0);
    context.lineTo(0, -85);

    // рисование часовой стрелки
    context.moveTo(0, 0);
    context.lineTo(-65, 0);

    // отображение пути без заливки
    context.stroke();
}
```

Поскольку начало координат к моменту поворота уже перенесено в центр часов, эта точка остается на месте, а изображение поворачивается вокруг нее по часовой стрелке (рис. 15.8).

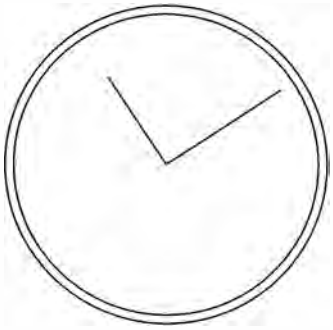

Рис. 15.8

Все эти преобразования и свойства, такие как `fillStyle` и `strokeStyle`, остаются в силе, пока их не изменить явно. Вручную восстановить их значения, предлагаемые по умолчанию, нельзя, но с помощью двух методов можно отслеживать их изменения. Если вы думаете, что позднее вам придется вернуться к текущей конфигурации свойств и преобразований, вызовите метод `save()`, чтобы сохранить все текущие параметры в стеке. После этого можно продолжить изменять контекст, а когда решите вернуться к сохраненным параметрам, просто вызовите метод `restore()`,

чтобы извлечь их из стека. Сохранять и восстанавливать конфигурации с помощью этих методов можно многократно в произвольном порядке, например:

Листинг 2DSaveRestoreExample01.htm


```
context.fillStyle = "#ff0000";
context.save();

context.fillStyle = "#00ff00";
context.translate(100, 100);
context.save();

context.fillStyle = "#0000ff";
// рисование синего прямоугольника
// с левым верхним углом в точке (100, 100)
context.fillRect(0, 0, 100, 200);

context.restore();
// рисование зеленого прямоугольника
// с левым верхним углом в точке (110, 110)
context.fillRect(10, 10, 100, 200);

context.restore();
// рисование красного прямоугольника
// с левым верхним углом в точке (0, 0)
context.fillRect(0, 0, 100, 200);
```

Здесь мы сначала назначаем свойству `fillStyle` красный цвет и вызываем метод `save()`, после чего изменяем значение `fillStyle` на зеленый цвет, переносим начало координат в точку (100,100) и еще раз вызываем метод `save()` для сохранения параметров. Затем мы назначаем свойству `fillStyle` синий цвет и рисуем прямоугольник, левый верхний угол которого из-за переноса начала координат отображается в точке (100,100). Первый вызов метода `restore()` восстанавливает для свойства `fillStyle` зеленый цвет, так что следующий прямоугольник рисуется зеленым цветом, но его левый верхний угол находится в точке (110,110), потому что перенос все еще действует. Второй вызов `restore()` отменяет перенос и восстанавливает первоначальное значение свойства `fillStyle`, поэтому последний прямоугольник рисуется красным цветом с левым верхним углом в точке (0,0).

Имейте в виду, что метод `save()` сохраняет только параметры и преобразования, примененные к контексту рисования, но не его содержимое.

Рисование изображений

Двухмерный контекст рисования содержит встроенные средства для работы с изображениями. Готовое изображение рисуется на холсте методом `drawImage()`, который можно вызывать с тремя разными наборами аргументов в зависимости от нужного результата. Если передать в метод HTML-элемент `<img>` и координаты, он просто выведет изображение в указанном месте, например:

Листинг 2DDrawImageExample01.htm

```
var image = document.images[0];
context.drawImage(image, 10, 10);
```

Этот код получает первое изображение в документе и выводит его в контексте рисования в позиции (10, 10) с сохранением масштаба. В метод можно также передать ширину и высоту итогового изображения, чтобы масштабировать его без изменения матрицы преобразования контекста, например:

Листинг 2DDrawImageExample01.htm

```
context.drawImage(image, 50, 10, 20, 30);
```

Этот код изменяет ширину и высоту изображения на 20 и 30 пикселей соответственно.

Кроме того, можно вывести в контексте только часть изображения. Для этого нужно передать в метод `drawImage()` девять аргументов: исходное изображение, его координаты x и y, ширину и высоту, а также координаты x и y, ширину и высоту целевого изображения. Эта перегруженная версия метода `drawImage()` обеспечивает наибольший контроль над рисованием, например:

Листинг 2DDrawImageExample01.htm

```
context.drawImage(image, 0, 10, 50, 50, 0, 100, 40, 60);
```

Этот метод выводит на холсте только часть исходного изображения, которая занимает 50 пикселов в ширину и высоту и имеет левый верхний угол в точке (0, 10). Итоговое изображение выводится в области с размерами 40×60 пикселов и левым верхним углом в точке (0, 100).

Эти операции рисования позволяют создавать интересные эффекты вроде тех, что показаны на рис. 15.9.

Рис. 15.9

Кроме HTML-элемента `<img>` в метод `drawImage()` в качестве первого аргумента можно передать другой элемент `<canvas>`, чтобы нарисовать содержимое одного холста на другом.

Используя метод `drawImage()` с другими методами, можно легко выполнять базовые операции над изображениями, результат которых можно получить с помощью метода `toDataURL()`. Однако если в контексте рисования выводится изображение не с текущего сайта, а из другого источника, при вызове метода `toDataURL()` произойдет ошибка. Например, если на странице на сайте `www.example.com` попытаться вывести изображение с сайта `www.wrox.com`, контекст будет воспринят как «грязный», что приведет к ошибке.

Тени

При рисовании фигур и путей в двухмерном контексте к ним автоматически добавляются тени согласно значениям следующих свойств:

- `shadowColor` — цвет тени в формате CSS. По умолчанию черный.
- `shadowOffsetX` — ширина тени по оси x. По умолчанию 0.
- `shadowOffsetY` — ширина тени по оси y. По умолчанию 0.
- `shadowBlur` — ширина области размытия по краям тени в пикселях. Если 0, края тени не размываются. По умолчанию 0.

Все эти свойства контекста можно читать и записывать. Достаточно задать их значения перед рисованием, и тени будут добавлены автоматически, например:

Листинг 2DFillRectShadowExample01.htm


```
var context = drawing.getContext("2d");

// настройка тени
context.shadowOffsetX = 5;
context.shadowOffsetY = 5;
context.shadowBlur = 4;
context.shadowColor = "rgba(0, 0, 0, 0.5)";

// рисование красного квадрата
context.fillStyle = "#ff0000";
context.fillRect(10, 10, 50, 50);

// рисование синего квадрата
context.fillStyle = "rgba(0,0,255,1)";
context.fillRect(30, 30, 50, 50);
```

Этот код выводит на экран два квадрата с одинаковыми тенями (рис. 15.10).

Поддержка теней в разных браузерах имеет некоторые особенности. Internet Explorer 9, Firefox 4 и Opera 11 всегда работают правильно, тогда как другие браузеры показывают тени со странными эффектами или вообще не показывают.

Так, Chrome до версии 10 включительно ошибочно применяет тень с заливкой к контуру фигуры, Chrome и Safari (до версии 5 включительно) не выводят тени для изображений с прозрачными пикселями, а Safari также не отображает тени для градиентов.

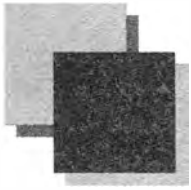

Рис. 15.10

Градиенты

Работая с двухмерным контекстом, можно с легкостью создавать и изменять градиенты, которые представляются экземплярами типа `CanvasGradient`. Создать линейный градиент можно с помощью метода `createLinearGradient()`, который принимает четыре аргумента: начальные координаты x и y и конечные координаты x и y. Получив эти данные, он создает объект `CanvasGradient` соответствующих размеров и возвращает его.

После создания объекта градиента следует назначить ему границы методом `addColorStop()`. Он принимает два аргумента: смещение границы градиента и CSS-цвет. Смещением границы может быть число в интервале от 0 (первый цвет) до 1 (последний цвет), например:

Листинг 2DFillRectGradientExample01.htm


```
var gradient = context.createLinearGradient(30, 30, 70, 70);

gradient.addColorStop(0, "white");
gradient.addColorStop(1, "black");
```

Этот объект `gradient` определяет градиент от точки (30, 30) до точки (70, 70) с начальным белым цветом и конечным черным. Далее можно назначить его свойству `fillStyle` или `strokeStyle` и нарисовать фигуру с градиентом:

Листинг 2DFillRectGradientExample01.htm

```
// рисование красного квадрата
context.fillStyle = "#ff0000";
context.fillRect(10, 10, 50, 50);

// рисование квадрата с градиентной заливкой
context.fillStyle = gradient;
context.fillRect(30, 30, 50, 50);
```

Чтобы увидеть весь диапазон цветов градиента, нужно правильно подобрать координаты фигур. Приведенному коду соответствует рис. 15.11.

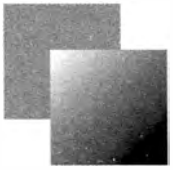

Рис. 15.11

Если сместить квадрат с градиентной заливкой ниже и правее, можно будет увидеть только часть градиента:

Листинг 2DFillRectGradientExample02.htm

```
context.fillStyle = gradient;
context.fillRect(50, 50, 50, 50);
```

Этот код рисует черный квадрат с небольшим серым фрагментом в левом верхнем углу. Это объясняется тем, что левый верхний угол квадрата приходится как раз на середину градиента, который становится черным, даже не достигнув центра квадрата. Чтобы упростить расчет градиентов, можно использовать следующую функцию:

Листинг 2DFillRectGradientExample03.htm


```
function createRectLinearGradient(context, x, y, width, height){
    return context.createLinearGradient(x, y, x+width, y+height);
}
```

Эта функция создает градиент на основе его начальных координат x и y, ширины и высоты, то есть с ней можно использовать те же числа, что и с методом `fillRect()`:

Листинг 2DFillRectGradientExample03.htm

```
var gradient = createRectLinearGradient(context, 30, 30, 50, 50);

gradient.addColorStop(0, "white");
gradient.addColorStop(1, "black");

// рисование квадрата с градиентной заливкой
context.fillStyle = gradient;
context.fillRect(30, 30, 50, 50);
```

Отслеживание координат при работе с холстом — важная и нетривиальная задача, которую можно упростить за счет вспомогательных функций, таких как `createRectLinearGradient()`.

Радиальные градиенты создают методом `createRadialGradient()`, который принимает шесть аргументов. Первые три определяют центр и радиус начальной окружности,

а последние три — те же параметры для конечной окружности. При работе с радиальными градиентами полезно представлять усеченный конус, основание и секущая плоскость которого соответствуют окружностям градиента.

В симметричном радиальном градиенте центры окружностей должны совпадать. Например, чтобы создать радиальный градиент в центре черного квадрата из предыдущего примера, нужно центрировать обе окружности в точке (55, 55), потому что противоположные углы квадрата расположены в точках (30, 30) и (80, 80):

Листинг 2DFillRectGradientExample04.htm

```
var gradient = context.createRadialGradient(55, 55, 10, 55, 55, 30);

gradient.addColorStop(0, "white");
gradient.addColorStop(1, "black");

// рисование красного квадрата
context.fillStyle = "#ff0000";
context.fillRect(10, 10, 50, 50);

// рисование квадрата с градиентной заливкой
context.fillStyle = gradient;
context.fillRect(30, 30, 50, 50);
```

Результат выполнения этого кода показан на рис. 15.12.

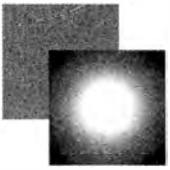

Рис. 15.12

С радиальными градиентами работать немного сложнее, чем с линейными. В большинстве случаев для создания эффектов с их помощью используют окружности с совпадающими центрами и разными радиусами.

Узоры

Узоры — это просто повторяющиеся изображения, которые можно использовать для заливки или рисования контуров фигуры. Чтобы создать узор, вызовите метод `createPattern()`, передав в него HTML-элемент `<img>` и строку, определяющую способ повтора изображения. Второй аргумент может принимать такие же значения, что и CSS-свойство `background-repeat`, то есть `"repeat"`, `"repeat-x"`, `"repeat-y"` и `"no-repeat"`, например:

Листинг 2DFillRectPatternExample01.htm


```
var image = document.images[0],
    pattern = context.createPattern(image, "repeat");

// рисование квадрата
context.fillStyle = pattern;
context.fillRect(10, 10, 150, 150);
```

Как и градиент, узор на самом деле начинается на холсте в точке (0, 0). Если узор задан в качестве стиля заливки, в конкретном месте холста просто демонстрируется его соответствующая часть. Например, приведенному коду соответствует узор, показанный на рис. 15.13.

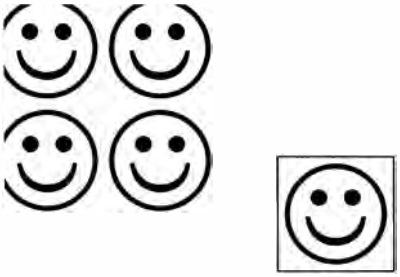

Рис. 15.13

Первым аргументом метода `createPattern()` также может быть элемент `<video>` или другой элемент `<canvas>`.

Работа с данными изображений

Одним из наиболее мощных механизмов двухмерного контекста является механизм получения необработанных данных изображения методом `getImageData()`. Он принимает четыре аргумента: координаты левого верхнего угла, ширину и высоту изображения, данные которого нужно получить. Например, получить данные области с размерами 50×50 и началом в точке (10, 5) можно следующим образом:

```
var imageData = context.getImageData(10, 5, 50, 50);
```

Этот метод возвращает экземпляр типа `ImageData`, который содержит всего три свойства: `width`, `height` и `data`. Свойство `data` является массивом с необработанными данными изображения. Каждый пиксель представляется в массиве четырьмя элементами, которые соответствуют красному, зеленому и синему компонентам, а также прозрачности (альфа-каналу) пикселя. Таким образом, данные первого пикселя содержатся в элементах с 0 по 3:

```
var data = imageData.data,
    red = data[0],
```

```
    green = data[1],
    blue = data[2],
    alpha = data[3];
```

Каждое значение в массиве может быть числом от 0 до 255 включительно. Доступ к необработанным данным изображения позволяет обрабатывать его разными способами. Например, можно создать простой черно-белый фильтр:

Листинг 2DImageDataExample01.htm


```
var drawing = document.getElementById("drawing");

// проверка полной поддержки элемента <canvas>
if (drawing.getContext){

    var context = drawing.getContext("2d"),
        image = document.images[0],
        imageData, data,
        i, len, average,
        red, green, blue, alpha;

    // вывод изображения без масштабирования
    context.drawImage(image, 0, 0);

    // получение данных изображения
    imageData = context.getImageData(0, 0, image.width, image.height);
    data = imageData.data;

    for (i=0, len=data.length; i < len; i+=4){

        red = data[i];
        green = data[i+1];
        blue = data[i+2];
        alpha = data[i+3];

        // получение среднего значения компонентов rgb
        average = Math.floor((red + green + blue) / 3);

        // задание новых цветов (без изменения прозрачности)
        data[i] = average;
        data[i+1] = average;
        data[i+2] = average;
    }

    // вывод черно-белого изображения
    imageData.data = data;
    context.putImageData(imageData, 0, 0);
}
```

Этот код выводит на экран исходное изображение, получает его данные и перебирает каждый пиксель в цикле `for`. Обратите внимание, что на каждой итерации цикла к значению `i` добавляется 4. Как только значения красного, зеленого и синего цветов получены, они усредняются, а затем среднее значение записывается в массив вместо каждого из трех исходных элементов. В результате все цвета заменяются

серым цветом той же яркости. После цикла измененный массив `data` снова назначается объекту `imageData`. Наконец, метод `putImageData()` выводит черно-белое изображение на холсте.

Конечно, преобразование цветного изображения в черно-белое — это лишь пример того, что можно делать с необработанными значениями пикселей. Дополнительные сведения о создании фильтров для таких данных см. в статье Илмари Хайкинен «Создание фильтров изображений с помощью холста» (Ilmari Heikkinen, «Making Image Filters with Canvas») по адресу www.html5rocks.com/en/tutorials/canvas/imagefilters/.

Данные изображения доступны, только если холст не «загрязнен» ресурсом из другого домена, в противном случае при доступе к ним возникает JavaScript-ошибка.

Композиция изображений

Во всех операциях рисования в двухмерном контексте учитываются свойства `globalAlpha` и `globalCompositionOperation`. Свойство `globalAlpha` содержит число от 0 до 1 включительно, которое задает прозрачность для всех операций рисования. По умолчанию оно равно 0. Если с несколькими операциями нужно использовать одно значение прозрачности, следует присвоить его свойству `globalAlpha`, выполнить рисование и снова обнулить свойство, например:

Листинг 2DGlobalAlphaExample01.htm


```
// рисование красного квадрата
context.fillStyle = "#ff0000";
context.fillRect(10, 10, 50, 50);

// глобальное изменение прозрачности
context.globalAlpha = 0.5;

// рисование синего квадрата
context.fillStyle = "rgba(0,0,255,1)";
context.fillRect(30, 30, 50, 50);

// сброс значения
context.globalAlpha = 0;
```

Этот код рисует синий квадрат поверх красного. Поскольку перед его рисованием свойству `globalAlpha` присваивается значение 0.5, в итоговом изображении красный квадрат виден через синий.

Свойство `globalCompositionOperation` показывает, как новые фигуры должны сочетаться с уже имеющимся в контексте изображением. Оно может содержать одно из следующих строковых значений:

- `source-over` (значение по умолчанию) — новое изображение рисуется перед существующим;

- `source-in` — новое изображение рисуется только там, где оно перекрывает существующее, а все остальное место становится прозрачным;
- `source-out` — новое изображение рисуется только там, где оно не перекрывает существующее, а все остальное место становится прозрачным;
- `source-atop` — новое изображение рисуется только там, где оно перекрывает существующее, а остальные места существующего изображения остаются неизменными;
- `destination-over` — новое изображение рисуется позади существующего и видно только через его прозрачные пиксели;
- `destination-in` — новое изображение рисуется позади существующего, а все места, где изображения не перекрываются, становятся прозрачными;
- `destination-out` — новое изображение стирает части существующего, с которыми перекрывается;
- `destination-atop` — новое изображение рисуется позади существующего, которое становится прозрачным там, где изображения не перекрываются;
- `lighter` — новое изображение объединяется с существующим, в результате получается более светлое изображение;
- `copy` — новое изображение стирает существующее, полностью заменяя его;
- `xor` — результат рисования получается путем применения исключающего ИЛИ к существующему и новому изображениям.

Эти операции сложно описать словами или пояснить черно-белыми изображениями. Цветные примеры каждой из них доступны по ссылке https://developer.mozilla.org/samples/canvas-tutorial/6_1_canvas_composite.html. Рекомендуется открыть ее в Internet Explorer 9+ или Firefox 4+, потому что в этих браузерах холст реализован наиболее полно. Вот простой пример композиции изображений:

Листинг 2DGlobalCompositeOperationExample01.htm


```
// рисование красного квадрата
context.fillStyle = "#ff0000";
context.fillRect(10, 10, 50, 50);

// настройка композиции
context.globalCompositeOperation = "destination-over";

// рисование синего квадрата
context.fillStyle = "rgba(0,0,255,1)";
context.fillRect(30, 30, 50, 50);
```

В обычной ситуации этот код рисует синий квадрат перед красным, но из-за того, что свойству `globalCompositeOperation` присвоено значение `"destination-over"`, синий квадрат оказывается сзади.

При использовании свойства `globalCompositionOperation` не забывайте тестировать код в нескольких браузерах, потому что реализации этих операций все еще заметно различаются. В Safari и Chrome имеются проблемы, которые можно увидеть, перейдя по приведенному URL-адресу и сравнив результат с той же страницей в Internet Explorer или Firefox.

WebGL

WebGL — это трехмерный контекст холста. В отличие от других веб-технологий, WebGL разрабатывается не в W3C, а в Khronos Group. Согласно веб-сайту этой организации, «Khronos Group — это некоммерческий консорциум, задачей которого является разработка свободных открытых стандартов параллельных вычислений, графических и динамических мультимедийных технологий для широкого диапазона платформ и устройств». Khronos Group разработала также ряд других графических API, таких как OpenGL ES 2.0, который лежит в основе WebGL.

Языки для работы с трехмерной графикой, такие как OpenGL, — сложная тема, и мы не будем углубляться в детали. Для использования WebGL рекомендуется познакомиться с OpenGL ES 2.0, потому что многие концепции в них одинаковы.

В этом разделе предполагается, что вы обладаете рабочими знаниями концепций OpenGL ES 2.0 и хотите узнать, как некоторые из них реализованы в WebGL. Дополнительные сведения об OpenGL доступны на сайте www.opengl.org, а на сайте www.learningwebgl.com можно найти серию отличных уроков по WebGL.

Типизированные массивы

WebGL требует выполнения сложных вычислений с предсказуемой точностью, для чего стандартные JavaScript-числа не подходят. Вместо них в WebGL используются *типизированные массивы* (typed arrays), то есть массивы, элементами которых являются значения конкретного типа.

В основе типизированных массивов лежит тип `ArrayBuffer`. Его экземпляр представляет указанное количество байтов в памяти, но не определяет их тип. По сути, с помощью `ArrayBuffer` можно лишь выделить для работы некоторое количество байтов. Например, следующий код выделяет 20 байтов:

```
var buffer = new ArrayBuffer(20);
```

Как только объект `ArrayBuffer` создан, единственное, что с ним можно сделать, это узнать количество содержащихся в нем байтов с помощью свойства `byteLength`:

```
var bytes = buffer.byteLength;
```

Хотя сам по себе объект `ArrayBuffer` не особо интересен, он очень важен в WebGL, что становится ясно при использовании представлений.

Представления

Представление (view) буфера массива определяет способ использования содержащихся в нем байтов. Наиболее универсальным является представление `DataView`, которое позволяет выбрать подмножество байтов в объекте `ArrayBuffer`. Чтобы сделать это, создайте экземпляр `DataView`, передав в него объект `ArrayBuffer`, необязательное смещение и необязательное количество выделяемых байтов, например:

```
// создание представления для всего буфера
var view = new DataView(buffer);

// создание представления начиная с байта 9
var view = new DataView(buffer, 9);

// создание представления с байта 9 по байт 18
var view = new DataView(buffer, 9, 10);
```

После создания объекта `DataView` смещение байтов и их количество хранятся в его свойствах `byteOffset` и `byteLength` соответственно:

```
alert(view.byteOffset);
alert(view.byteLength);
```

Эти свойства позволяют с легкостью читать представление. Кроме того, можно получить буфер массива с помощью свойства `buffer`.

Методы чтения и записи представления зависят от типа данных, с которым вы работаете. Поддерживаемые типы данных и связанные с ними методы указаны в таблице.

Тип данных	Метод чтения	Метод записи
8-разрядное целое число со знаком	getInt8(byteOffset)	setInt8(byteOffset, value)
8-разрядное целое число без знака	getUint8(byteOffset)	setUint8(byteOffset, value)
16-разрядное целое число со знаком	getInt16(byteOffset, littleEndian)	setInt16(byteOffset, value, littleEndian)
16-разрядное целое число без знака	getUint16(byteOffset, littleEndian)	setUint16(byteOffset, value, littleEndian)
32-разрядное целое число со знаком	getInt32(byteOffset, littleEndian)	setInt32(byteOffset, value, littleEndian)
32-разрядное целое число без знака	getUint32(byteOffset, littleEndian)	setUint32(byteOffset, value, littleEndian)
32-разрядное число с плавающей точкой	getFloat32(byteOffset, littleEndian)	setFloat32(byteOffset, value, littleEndian)
64-разрядное число с плавающей точкой	getFloat64(byteOffset, littleEndian)	setFloat64(byteOffset, value, littleEndian)

В качестве первого аргумента каждый из этих методов принимает смещение, по которому нужно прочитать или записать данные. Помните, что в зависимости от типа данные могут занимать более одного байта. Так, для хранения 8-разрядного целого числа без знака требуется один байт, а для хранения 32-разрядного числа с плавающей точкой — четыре. При использовании объекта `DataView` вы должны сами следить за размером данных и выбирать правильные методы для работы с ними, например:

Листинг DataViewExample01.htm


```
var buffer = new ArrayBuffer(20),
    view = new DataView(buffer),
    value;

view.setUint16(0, 25);
view.setUint16(2, 50);    // 16-разрядные целые числа занимают по 2 байта
value = view.getUint16(0);
```

Этот код сохраняет в буфере массива два 16-разрядных целых числа без знака. Поскольку каждое из них занимает два байта, первое число сохраняется со смещением 0, а второе — со смещением 2.

У каждого метода, работающего с 16-разрядными или более объемными числами, есть необязательный аргумент `littleEndian`. Это логическое значение указывает, как нужно прочитать или записать значение: с прямым (little-endian — младший байт является первым) или обратным (big-endian — младший байт является последним) порядком следования байтов. Если вы не знаете, какое значение использовать, оставьте вариант, предлагаемый по умолчанию (обратный порядок).

Из-за того что в этих методах используются смещения байтов, а не номера элементов, одни и те же байты доступны разными способами, например:

Листинг DataViewExample02.htm


```
var buffer = new ArrayBuffer(20),
    view = new DataView(buffer),
    value;

view.setUint16(0, 25);
value = view.getInt8(0);

alert(value);    // 0
```

Этот код записывает значение 25 со смещением 0 как 16-разрядное целое число без знака. При чтении того же значения в формате 8-разрядного целого числа со знаком возвращается значение 0, потому что в двоичной форме первый байт значения 25 содержит только нули (рис. 15.14).

Итак, `DataView` обеспечивает доступ к данным в буфере массива на уровне байтов, но при этом вы должны следить, где хранятся отдельные значения и сколько места они занимают. Чтобы не возиться с этими мелочами, можно использовать типизированные представления.

Рис. 15.14

Типизированные представления

Типизированные представления (typed views) обычно называют типизированными массивами, потому что они похожи на обычные массивы, только содержат элементы конкретного типа. Есть несколько типизированных представлений, причем все они наследуются от типа `DataView`:

- `Int8Array` — представляет значения как 8-разрядные целые числа с дополнением до двух;
- `Uint8Array` — представляет значения как 8-разрядные целые числа без знака;
- `Int16Array` — представляет значения как 16-разрядные целые числа с дополнением до двух;
- `Uint16Array` — представляет значения как 16-разрядные целые числа без знака;
- `Int32Array` — представляет значения как 32-разрядные целые числа с дополнением до двух;
- `Uint32Array` — представляет значения как 32-разрядные целые числа без знака;
- `Float32Array` — представляет значения как 32-разрядные IEEE-числа с плавающей точкой;
- `Float64Array` — представляет значения как 64-разрядные IEEE-числа с плавающей точкой.

Каждое представление преподносит данные по-своему, при этом каждый элемент данных может занимать один или несколько байтов. Например, 20 байтов из буфера массива могут быть представлены как 20 значений в объекте `Int8Array` или `Uint8Array`, как 10 значений в объекте `Int16Array` или `Uint16Array`, как 5 значений в объекте `Int32Array`, `Uint32Array` или `Float32Array` либо как 2 значения в объекте `Float64Array`.

Поскольку каждый из этих типов наследуется от `DataView`, их экземпляры можно создавать, передавая в конструктор те же аргументы: объект `ArrayBuffer`, начальное смещение байтов (по умолчанию 0) и количество байтов. Обязателен только первый аргумент. Вот несколько примеров:

```
// создание представления для всего буфера
var int8s = new Int8Array(buffer);
```

```
// создание представления начиная с байта 9
var int16s = new Int16Array(buffer, 9);

// создание представления с байта 9 по байт 18
var uint16s = new Uint16Array(buffer, 9, 10);
```

Благодаря поддержке работы с частями буфера можно хранить в одном буфере числа разных типов. Например, следующий код позволяет хранить 8-разрядные целые числа в начале буфера и 16-разрядные после них:

```
// хранение 8- и 16-разрядных целых чисел в одном буфере
var int8s = new Int8Array(buffer, 0, 10);
var uint16s = new Uint16Array(buffer, 11, 10);
```

У конструктора каждого типизированного представления есть свойство `BYTES_PER_ELEMENT`, которое указывает, сколько байтов занимает каждый элемент. Так, свойство `Uint8Array.BYTES_PER_ELEMENT` равно 1, а `Float32Array.BYTES_PER_ELEMENT` — 4. Эти значения можно использовать для инициализации представления:

```
// требуется место для 10 элементов
var int8s = new Int8Array(buffer, 0, 10 * Int8Array.BYTES_PER_ELEMENT);

// требуется место для 5 элементов
var uint16s = new Uint16Array(buffer, int8s.byteOffset + int8s.byteLength,
                              5 * Uint16Array.BYTES_PER_ELEMENT);
```

Этот код создает для буфера массива два представления. Первые 10 байтов отводятся для хранения 8-разрядных целых чисел, а вторые десять — для хранения 16-разрядных целых чисел без знака. Смещение 16-разрядных чисел рассчитывается в конструкторе `Uint16Array` с помощью свойств `byteOffset` и `byteLength` объекта `Int8Array`.

Чтобы еще больше упростить работу с двоичными данными, можно создавать типизированные представления, не создавая перед этим объект `ArrayBuffer`. Просто укажите в конструкторе, сколько элементов должно содержать представление, и для него автоматически будет создан подходящий объект `ArrayBuffer`, например:

```
// создание представления для десяти 8-разрядных целых чисел (10 байтов)
var int8s = new Int8Array(10);

// создание представления для десяти 16-разрядных целых чисел (20 байтов)
var int16s = new Int16Array(10);
```

Типизированное представление можно также создать из обычного массива с помощью конструктора:

```
// создание представления для пяти 8-разрядных целых чисел (5 байтов)
var int8s = new Int8Array([10, 20, 30, 40, 50]);
```

Это оптимальный способ инициализации типизированных представлений значениями, предлагаемыми по умолчанию, который часто используется в WebGL-проектах.

Такое применение типизированных представлений делает их более похожими на обычные объекты `Array` и гарантирует, что при чтении или записи будут использоваться правильные типы данных.

При работе с типизированным представлением можно получать доступ к его элементам через скобочную нотацию и определять их количество с помощью свойства `length`. Это позволяет перебирать типизированные представления так же, как объекты `Array`:

```
for (var i=0, len=int8s.length; i < len; i++){
    console.log("Value at position " + i + " is " + int8s[i]);
}
```

Присваивать значения элементам типизированного представления также можно с помощью скобочной нотации. Если значение не помещается в отведенном для него месте, оно сохраняется как остаток от деления на максимально возможное число. Например, максимальное 16-разрядное целое число без знака равно 65 535. Если попытаться сохранить в двух байтах число 65 536, оно станет нулем, число 65 537 станет единицей и т. д.:

```
var uint16s = new Uint16Array(10);
uint16s[0] = 65537;
alert(uint16s[0]);    // 1
```

Если типы значения и представления не соответствуют друг другу, никакие ошибки не генерируются — вы сами должны следить за тем, чтобы числа соответствовали ограничениям на размер.

У типизированных представлений есть также метод `subarray()`, который позволяет создать новое представление (подмассив) на основе части базового буфера массива. Он принимает два аргумента: индекс начального элемента и необязательный индекс конечного элемента. Возвращает этот метод объект такого же типа, для которого он был вызван, например:

```
var uint16s = new Uint16Array(10),
    sub = uint16s.subarray(2, 5);
```

Здесь представление `sub` является экземпляром `Uint16Array` и основано на том же объекте `ArrayBuffer`, что и `uint16s`. Преимуществом таких подмассивов является доступ к подмножеству элементов большего массива без риска случайно изменить другие элементы.

Типизированные представления играют важную роль в WebGL.

Контекст WebGL

Поскольку спецификация WebGL еще разрабатывается, WebGL-контекст в поддерживающих его браузерах пока называется `"experimental-webgl"`, а по завершении

разработки будет называться просто "webgl". Браузеры, не поддерживающие WebGL, при попытке получить WebGL-контекст возвращают значение null. Прежде чем использовать контекст, всегда проверяйте возвращенное значение:

Листинг WebGLExample01.htm


```
var drawing = document.getElementById("drawing");

// проверка полной поддержки элемента <canvas>
if (drawing.getContext){

    var gl = drawing.getContext("experimental-webgl");
    if (gl){
        // продолжение работы с WebGL
    }
}
```

В большинстве приложений и примеров WebGL-контекст называется gl, потому что методы и значения, относящиеся к OpenGL ES 2.0, обычно имеют префикс "gl". Следование этой конвенции сделает JavaScript-код более похожим на OpenGL-программу.

Как только WebGL-контекст получен, можно приступать к рисованию трехмерной графики. Как уже отмечалось, WebGL является версией OpenGL ES 2.0 для веб-приложений, поэтому концепции, описанные в этом разделе, на самом деле перенесены в JavaScript из OpenGL.

Чтобы настроить параметры WebGL-контекста, можно передать в метод getContext() второй аргумент — объект с одним или несколькими свойствами из перечисленных:

- alpha — если это свойство равно true, для контекста создается буфер альфа-канала. По умолчанию true.
- depth — если это свойство равно true, доступен 16-разрядный буфер глубины. По умолчанию true.
- stencil — если это свойство равно true, доступен 8-разрядный буфер трафарета. По умолчанию false.
- antialias — если это свойство равно true, выполняется сглаживание с использованием механизма, предлагаемого по умолчанию. По умолчанию true.
- premultipliedAlpha — если это свойство равно true, предполагается, что буфер рисования содержит предварительно умноженные альфа-значения. По умолчанию true.
- preserveDrawingBuffer — если это свойство равно true, буфер рисования сохраняется после завершения рисования. По умолчанию false. Изменяйте это значение, только если хорошо понимаете, что делаете, иначе могут возникнуть проблемы с быстродействием.

Передать объект с параметрами в метод getContext() можно следующим образом:

Листинг WebGLExample01.htm


```
var drawing = document.getElementById("drawing");

// проверка полной поддержки элемента <canvas>
if (drawing.getContext){

    var gl = drawing.getContext("experimental-webgl", { alpha: false});
    if (gl){
        // продолжение работы с WebGL
    }
}
```

Большинство параметров контекста используются лишь в нетривиальных сценариях, а во многих случаях подходят значения, предлагаемые по умолчанию.

Если WebGL-контекст невозможно создать с помощью метода `getContext()`, некоторые браузеры генерируют ошибку, поэтому лучше заключить его вызов в блок `try-catch`:

Листинг WebGLExample01.htm


```
var drawing = document.getElementById("drawing"),
    gl;

// проверка полной поддержки элемента <canvas>
if (drawing.getContext){
    try {
        gl = drawing.getContext("experimental-webgl");
    } catch (ex) {
        // пустой блок
    }

    if (gl){
        // продолжение работы с WebGL
    } else {
        alert("WebGL context could not be created.");
    }
}
```

Константы

Если вы знакомы с OpenGL, вам должны быть известны многие константы с префиксом `GL_`. В WebGL каждая константа доступна в объекте WebGL-контекста с префиксом `gl.` вместо префикса `GL_`. Например, константа `GL_COLOR_BUFFER_BIT` доступна как `gl.COLOR_BUFFER_BIT`. В WebGL есть такие аналоги большинства OpenGL-констант (хотя некоторые константы все же отсутствуют).

Именование методов

Имена многих OpenGL- и WebGL-методов включают сведения о типах их аргументов. Если метод может принимать разное количество аргументов разных типов,

к его имени добавляется соответствующий суффикс. Число в нем указывает количество аргументов (от 1 до 4), а буква — их тип («f» для чисел с плавающей точкой или «i» для целых чисел). Например, метод `gl.uniform4f()` ожидает четыре числа с плавающей точкой, а метод `gl.uniform3i()` — три целых числа.

Многие методы также могут принимать массив вместо отдельных аргументов, на что указывает буква «v» (сокращение от «vector»). Так, метод `gl.uniform3iv()` принимает массив с тремя целыми числами. Помните об этой конвенции при обсуждении WebGL.

Подготовка к рисованию

Одно из первых действий при работе с WebGL-контекстом — заполнение элемента `<canvas>` сплошным цветом для подготовки к рисованию. Для этого сначала нужно задать цвет методом `clearColor()`, который принимает четыре аргумента: значения красного, зеленого и синего цветов, а также прозрачность. Каждый аргумент должен быть числом от 0 до 1, определяющим долю значения в составе окончательного цвета, например:

Листинг WebGLExample01.htm


```
gl.clearColor(0,0,0,1);         // черный
gl.clear(gl.COLOR_BUFFER_BIT);
```

Этот код задает для буфера цвета черный цвет, а затем вызывает метод `clear()`, который эквивалентен OpenGL-методу `glClear()`. Аргумент `gl.COLOR_BUFFER_BIT` указывает WebGL использовать ранее определенный цвет для заполнения области. С ее очистки начинаются практически все операции рисования.

Области просмотра и координаты

Перед началом рисования в WebGL имеет смысл определить область просмотра, которая по умолчанию занимает весь холст. Чтобы изменить область просмотра, вызовите метод `viewport()`, передав ему координаты начала области просмотра, а также ее ширину и высоту относительно холста. Например, следующий вызов определяет область просмотра во весь холст:

```
gl.viewport(0, 0, drawing.width, drawing.height);
```

Система координат области просмотра отличается от той, которая обычно используется на веб-странице. Как показано на рис. 15.15, левый нижний угол элемента `<canvas>` имеет координаты (0, 0), а правый верхний — (ширина–1, высота–1).

Умение рассчитывать область просмотра позволяет использовать для рисования только часть элемента `<canvas>`, например:

```
// область просмотра - левая нижняя четверть элемента <canvas>
gl.viewport(0, 0, drawing.width/2, drawing.height/2);
```

```
// область просмотра - левая верхняя четверть элемента <canvas>
gl.viewport(0, drawing.height/2, drawing.width/2, drawing.height/2);

// область просмотра - правая нижняя четверть элемента <canvas>
gl.viewport(drawing.width/2, 0, drawing.width/2, drawing.height/2);
```

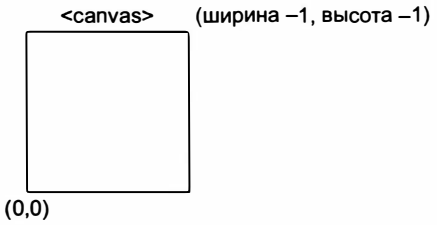

Рис. 15.15

Внутри области просмотра используется другая система координат, показанная на рис. 15.16. Начало координат находится в центре области просмотра, ее левому нижнему углу соответствует точка (–1, –1), а правому верхнему — точка (1, 1).

Рис. 15.16

Если при рисовании указать координаты вне области просмотра, например (1, 2), рисунок будет обрезан.

Буферы

В JavaScript информация о вершинах хранится в типизированных массивах, но для работы ее нужно преобразовать в WebGL-буферы. Для этого сначала необходимо создать буфер методом `gl.createBuffer()`, а затем связать его с WebGL-контекстом методом `gl.bindBuffer()`. После этого можно заполнить буфер данными, например:

```
var buffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, buffer);
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array([0, 0.5, 1]),
              gl.STATIC_DRAW);
```

Вызов метода `gl.bindBuffer()` делает объект `buffer` текущим буфером контекста, после чего все операции с буфером выполняются непосредственно с объектом

buffer. Так, вызов gl.bufferData() не содержит явную ссылку на buffer, но все равно работает с ним. Последняя строка инициализирует буфер данными из массива Float32Array, в котором обычно хранится вся информация о вершинах. Если предполагается задействовать метод drawElements() для вывода содержимого буфера, можно указать константу gl.ELEMENT_ARRAY_BUFFER.

Последний аргумент метода gl.bufferData() показывает, как будет использоваться буфер. Им может быть одна из следующих констант:

- gl.STATIC_DRAW — данные будут загружены один раз и использованы многократно;
- gl.STREAM_DRAW — данные будут загружены один раз и использованы всего несколько раз;
- gl.DYNAMIC_DRAW — данные будут многократно изменяться и использоваться для рисования.

Если у вас нет солидного опыта работы с OpenGL, вероятнее всего, в большинстве случаев вы будете использовать константу gl.STATIC_DRAW.

Буферы остаются в памяти до выгрузки страницы-контейнера. Если буфер больше не требуется, лучше освободить занимаемую им память, вызвав метод gl.deleteBuffer():

```
gl.deleteBuffer(buffer);
```

Ошибки

В отличие от JavaScript, WebGL-операции обычно не генерируют ошибки. Вместо этого после вызова метода, в котором может произойти ошибка, вы должны вызвать метод gl.getError(). Он возвращает одну из следующих констант, описывающих тип произошедшей ошибки:

- gl.NO_ERROR — последняя операция выполнена без ошибок (значение 0);
- gl.INVALID_ENUM — в метод, принимающий одну из WebGL-констант, передан неправильный аргумент;
- gl.INVALID_VALUE — вместо числа без знака использовано отрицательное число;
- gl.INVALID_OPERATION — операция не может быть выполнена в текущем состоянии;
- gl.OUT_OF_MEMORY — недостаточно памяти для выполнения операции;
- gl.CONTEXT_LOST_WEBGL — WebGL-контекст утрачен из-за внешнего события, такого как сбой электропитания.

Метод gl.getError() возвращает одно значение ошибки, поэтому каждый последующий его вызов может возвращать другое значение. Если ошибок несколько, это продолжается, пока не будет возвращено значение gl.NO_ERROR. Если вы выполнили несколько операций, для обработки ошибок можно использовать цикл с методом getError():

```
var errorCode = gl.getError();
while(errorCode){
    console.log("Error occurred: " + errorCode); // Сообщение об ошибке
    errorCode = gl.getError();
}
```

Если WebGL-сценарий не выводит правильный результат, добавление нескольких вызовов `gl.getError()` в код может помочь решить проблему.

Шейдеры

Шейдеры (shaders) — это еще одна концепция из OpenGL. В WebGL доступны шейдеры двух типов: *вершинные* (vertex shaders) и *фрагментные* (fragment shaders). Вершинные шейдеры служат для преобразования трехмерных вершин в двухмерные точки с целью их визуализации, а фрагментные — для вычисления правильного цвета пикселей. WebGL-шейдеры примечательны тем, что их создают не на JavaScript, а на *GLSL* (OpenGL Shading Language), который никак не пересекается с JavaScript или C.

Создание шейдеров

GLSL — это C-подобный язык, специально предназначенный для определения OpenGL-шейдеров. Поскольку WebGL является реализацией OpenGL ES 2, OpenGL-шейдеры можно без изменений использовать в WebGL, что позволяет легко переносить графику из приложений для настольных компьютеров в веб-приложения.

У каждого шейдера есть метод `main()`, который многократно выполняется во время рисования. Передать данные в шейдер можно двумя способами: с помощью *атрибутов* (attributes) и *однородных значений* (uniforms). Атрибуты используются для передачи вершин в вершинные шейдеры, а однородные значения — для передачи констант в шейдеры обоих типов. Атрибуты и однородные значения определяются вне метода `main()` с помощью ключевых слов `attribute` и `uniform` соответственно, после которых указывается тип данных, а за ним — имя. Вот простой пример вершинного шейдера:

Листинг WebGLExample02.htm


```
// OpenGL Shading Language
// Шейдер от Бартека Дроздзя (Bartek Drozdz)
attribute vec2 aVertexPosition;

void main() {
        gl_Position = vec4(aVertexPosition, 0.0, 1.0);
}
```

Для этого шейдера определен единственный атрибут `aVertexPosition`, который представляет собой массив из двух элементов (тип данных `vec2`), определяющих координаты x и y. Вершинный шейдер всегда должен назначать специальной

переменной gl_Position вершину из четырех частей, даже если ему переданы только две координаты. Шейдер в показанном примере создает новый массив из четырех элементов (vec4) и добавляет в него недостающие значения, преобразуя двухмерные координаты в трехмерные.

Фрагментные шейдеры похожи на вершинные, но передавать им данные можно только как однородные значения. Вот пример фрагментного шейдера:

Листинг WebGLExample02.htm

```
// OpenGL Shading Language
// Шейдер от Бартека Дроздзя (Bartek Drozdz) из статьи по адресу
// http://www.netmagazine.com/tutorials/get-started-webgl-draw-square
uniform vec4 uColor;

void main() {
        gl_FragColor = uColor;
}
```

Фрагментный шейдер должен присваивать значение переменной gl_FragColor, которая задает цвет, используемый при рисовании. Для шейдера в показанном примере определяется однородный (uniform) цвет из четырех частей (vec4) с именем uColor, а сам шейдер только назначает полученное значение переменной gl_FragColor. Значение uColor изменить в шейдере нельзя.

GLSL — непростой язык с множеством нюансов, которому посвящены целые книги. В этом разделе приведены лишь поверхностные сведения о нем, касающиеся WebGL. Дополнительные сведения о GLSL см. в книге «OpenGL Shading Language» (Randi J. Rost, Addison-Wesley, 2006).

Создание программ с шейдерами

Изначально браузеры не понимают GLSL-код, поэтому для создания программы с шейдерами необходима GLSL-строка, готовая к компиляции и компоновке. Ради простоты и удобства шейдеры обычно включают в код страницы с помощью элементов <script> с пользовательским атрибутом type. Применение неправильного атрибута type не позволяет браузеру интерпретировать содержимое элемента <script>, при этом обеспечивается легкий доступ к шейдеру, например:

Листинг WebGLExample02.htm


```
<script type="x-webgl/x-vertex-shader" id="vertexShader">
attribute vec2 aVertexPosition;

void main() {
        gl_Position = vec4(aVertexPosition, 0.0, 1.0);
}
</script>
<script type="x-webgl/x-fragment-shader" id="fragmentShader">
uniform vec4 uColor;
```

```
void main() {
        gl_FragColor = uColor;
}
</script>
```

Содержимое такого элемента `<script>` можно извлечь с помощью свойства `text`:

```
var vertexGlsl = document.getElementById("vertexShader").text,
    fragmentGlsl = document.getElementById("fragmentShader").text;
```

Более сложные WebGL-приложения могут загружать шейдеры динамически с помощью технологии Ajax (см. главу 21), здесь же важно понять, что для использования шейдера требуется GLSL-строка.

Когда GLSL-строка получена, нужно создать объект шейдера, вызвав метод `gl.createShader()` и передав ему в качестве аргумента тип создаваемого шейдера (`gl.VERTEX_SHADER` или `gl.FRAGMENT_SHADER`). После этого следует назначить шейдеру исходный GLSL-код с помощью метода `gl.shaderSource()` и скомпилировать его методом `gl.compileShader()`, например:

Листинг WebGLExample02.htm

```
var vertexShader = gl.createShader(gl.VERTEX_SHADER);
gl.shaderSource(vertexShader, vertexGlsl);
gl.compileShader(vertexShader);

var fragmentShader = gl.createShader(gl.FRAGMENT_SHADER);
gl.shaderSource(fragmentShader, fragmentGlsl);
gl.compileShader(fragmentShader);
```

Этот код создает два шейдера и сохраняет их в переменных `vertexShader` и `fragmentShader`, которые затем можно скомпоновать в программе следующим образом:

Листинг WebGLExample02.htm

```
var program = gl.createProgram();
gl.attachShader(program, vertexShader);
gl.attachShader(program, fragmentShader);
gl.linkProgram(program);
```

Этот код создает переменную `program`, добавляет к программе шейдеры методом `attachShader()`, а затем инкапсулирует их в ней методом `gl.linkProgram()`. После компоновки программы можно дать WebGL-контексту команду использовать ее, вызвав метод `gl.useProgram()`:

```
gl.useProgram(program);
```

После вызова метода `gl.useProgram()` указанная программа будет использоваться во всех операциях рисования.

Передача значений шейдерам

Каждый из определенных нами шейдеров нуждается в данных для выполнения своей работы. Для передачи значений в шейдер нужно сначала найти переменную, которую необходимо заполнить. В случае однородной переменной это можно сделать с помощью метода `gl.getUniformLocation()`, который возвращает объект, представляющий расположение переменной в памяти. Затем это расположение можно задействовать для назначения данных, например:

Листинг WebGLExample02.htm

```
var uColor = gl.getUniformLocation(program, "uColor");
gl.uniform4fv(uColor, [0, 0, 0, 1]);
```

Этот код находит однородную переменную `uColor` в объекте `program` и возвращает ее расположение в памяти. Во второй строке метод `gl.univorm4fv()` задает переменной `uColor` значение.

Похожая процедура соблюдается и при работе с переменными-атрибутами в вершинных шейдерах. Расположение переменной-атрибута можно получить методом `gl.getAttribLocation()`, а когда оно получено, его можно использовать следующим образом:

Листинг WebGLExample02.htm

```
var aVertexPosition = gl.getAttribLocation(program, "aVertexPosition");
gl.enableVertexAttribArray(aVertexPosition);
gl.vertexAttribPointer(aVertexPosition, itemSize, gl.FLOAT, false, 0, 0);
```

Этот код получает расположение атрибута `aVertexPosition` и инициирует его использование с помощью метода `gl.enableVertexAttribArray()`. Последняя строка создает указатель на последний буфер, заданный с помощью метода `gl.bindBuffer()`, и сохраняет его в переменной `aVertexPosition`, чтобы его мог использовать вершинный шейдер.

Отладка шейдеров и программ

Как и другие действия в WebGL, операции с шейдерами могут завершаться ошибками без уведомления об этом. Если вы считаете, что могла произойти ошибка, нужно вручную запросить сведения о шейдере или программе у WebGL-контекста.

Чтобы получить статус шейдера после попытки его компиляции, вызовите метод `gl.getShaderParameter()`:

Листинг WebGLExample02.htm

```
if (!gl.getShaderParameter(vertexShader, gl.COMPILE_STATUS)){
    alert(gl.getShaderInfoLog(vertexShader));
}
```

Этот код проверяет статус компиляции шейдера `vertexShader`. Если он был скомпилирован успешно, метод `gl.getShaderParameter()` возвращает `true`. Если он возвращает `false`, это означает, что при компиляции шейдера произошла ошибка. Получить сведения о ней можно методом `gl.getShaderInfoLog()`, который принимает шейдер и возвращает строку с описанием проблемы. Методы `gl.getShaderParameter()` и `gl.getShaderInfoLog()` можно использовать и с вершинными, и с фрагментными шейдерами.

Для проверки статуса программы служит похожий метод `gl.getProgramParameter()`. Чаще всего сбои программ происходят во время компоновки, статус которой можно проверить с помощью следующего кода:

Листинг WebGLExample02.htm

```
if (!gl.getProgramParameter(program, gl.LINK_STATUS)){
    alert(gl.getProgramInfoLog(program));
}
```

Как и `gl.getShaderParameter()`, метод `gl.getProgramParameter()` возвращает `true`, если компоновка выполнена успешно, и `false` в противном случае. Кроме того, доступен метод `gl.getProgramInfoLog()`, с помощью которого можно получить сведения о сбое программы.

Эти методы используются преимущественно во время отладки. Если от них не зависит другой код, их можно удалить из окончательного продукта.

Рисование

WebGL позволяет рисовать только точки, линии и треугольники, а все остальные фигуры нужно составлять из этих трех базовых компонентов, рисуемых в трехмерном пространстве. Рисование выполняется методами `drawArrays()` и `drawElements()`, первый из которых работает с буферами массивов, а второй — с буферами массивов элементов.

Первым аргументом методов `gl.drawArrays()` и `drawElements()` является константа, задающая тип фигуры, которую нужно нарисовать. Поддерживаемые значения:

- `gl.POINTS` — указывает, что каждую вершину нужно нарисовать как точку.
- `gl.LINES` — указывает, что массив содержит последовательность вершин, которые нужно соединить линиями. Каждое множество вершин содержит начальную и конечную точки, так что количество вершин в массиве должно быть четным, чтобы были нарисованы все линии.
- `gl.LINE_LOOP` — указывает, что массив содержит последовательность вершин, которые нужно соединить линиями. Линии рисуются от первой вершины ко второй, от второй к третьей и так далее, пока не будет достигнута последняя вершина. После этого последняя вершина соединяется с первой. В результате получается контур фигуры.

- `gl.LINE_STRIP` — то же, что и `gl.LINE_LOOP`, но без рисования линии от последней вершины к первой.
- `gl.TRIANGLES` — указывает, что массив содержит последовательность вершин треугольников. Каждый треугольник рисуется отдельно от предыдущего без общих вершин, если иное не указано явно.
- `gl.TRIANGLES_STRIP` — то же, что и `gl.TRIANGLES`, но каждая последующая вершина после первых трех определяет новый треугольник, включающий кроме нее две предыдущие вершины. Например, если массив содержит вершины *A*, *B*, *C* и *D*, первым треугольником будет *ABC*, а вторым — *BCD*.
- `gl.TRIANGLES_FAN` — то же, что и `gl.TRIANGLES`, но каждая последующая вершина после первых трех определяет новый треугольник, включающий кроме нее предыдущую и первую вершины. Например, если массив содержит вершины *A*, *B*, *C* и *D*, первым треугольником будет *ABC*, а вторым — *ACD*.

Метод `gl.drawArrays()` принимает одно из этих значений в качестве первого аргумента, начальный индекс в буфере массива — в качестве второго и количество вершин в буфере массива — в качестве третьего. В следующем примере он рисует на холсте один треугольник:

Листинг WebGLExample02.htm


```
// предполагается, что область просмотра очищена с помощью шейдеров

// определение координат трех вершин
var vertices = new Float32Array([ 0, 1, 1, -1, -1, -1 ]),
    buffer = gl.createBuffer(),
    vertexSetSize = 2,
    vertexSetCount = vertices.length/vertexSetSize,
    uColor, aVertexPosition;

// запись данных в буфер
gl.bindBuffer(gl.ARRAY_BUFFER, buffer);
gl.bufferData(gl.ARRAY_BUFFER, vertices, gl.STATIC_DRAW);

// передача цвета фрагментному шейдеру
uColor = gl.getUniformLocation(program, "uColor");
gl.uniform4fv(uColor, [ 0, 0, 0, 1 ]);

// передача информации о вершинах вершинному шейдеру
aVertexPosition = gl.getAttribLocation(program, "aVertexPosition");
gl.enableVertexAttribArray(aVertexPosition);
gl.vertexAttribPointer(aVertexPosition, vertexSetSize, gl.FLOAT,
                       false, 0, 0);

// рисование треугольника
gl.drawArrays(gl.TRIANGLES, 0, vertexSetCount);
```

Массив `Float32Array` в этом примере содержит три вершины, каждая из которых имеет две координаты. Размер и количество вершин важно отслеживать для использования в дальнейших вычислениях. Переменной `vertexSetSize`, которая определяет

количество координат вершины, присваивается значение 2, тогда как количество вершин `vertexSetCount` вычисляется. Затем информация о вершинах сохраняется в буфере и фрагментному шейдеру передаются сведения о цвете.

Вершинному шейдеру передается количество координат вершины, при этом указывается, что они представлены числами с плавающей точкой (`gl.FLOAT`). Четвертым аргументом метода `gl.vertexAttribPointer` является логическое значение, указывающее, что координаты не нормализованы. Пятый аргумент — это *значение шага* (stride value), которое показывает, сколько элементов массива нужно пропустить для получения следующего значения. Если вы не уверены, каким оно должно быть, используйте значение 0. Последний аргумент метода определяет начальное смещение в массиве. Значение 0 соответствует первому элементу.

Наконец, метод `gl.drawArrays()` рисует треугольник. Аргумент `gl.TRIANGLES` предписывает ему нарисовать треугольник с вершинами (0, 1), (1, –1) и (–1, –1), залив его цветом, переданным фрагментному шейдеру. Второй аргумент метода указывает начальное смещение в буфере, а последний — общее количество вершин. Результат выполнения этого кода показан на рис. 15.17.

Рис. 15.17

Передав другой первый аргумент методу `gl.drawArrays()`, можно изменить способ рисования треугольника. Два возможных варианта показаны на рис. 15.18.

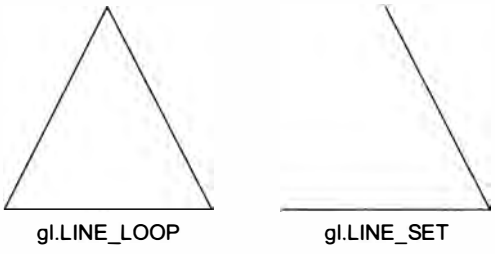

Рис. 15.18

Текстуры

С DOM-изображениями можно использовать WebGL-текстуры. Для этого нужно создать текстуру методом `gl.createTexture()`, а затем связать с ней изображение. Если изображение еще не загружено, для его динамической загрузки можно создать экземпляр типа `Image`. Текстура не инициализируется, пока изображение не загружено полностью, так что настраивать ее нужно после события `load`, например:

```
var image = new Image(),
    texture;
image.src = "smile.gif";
image.onload = function(){
    texture = gl.createTexture();
    gl.bindTexture(gl.TEXTURE_2D, texture);
    gl.pixelStorei(gl.UNPACK_FLIP_Y_WEBGL, true);

    gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, gl.RGBA, gl.UNSIGNED_BYTE,
                  image);
    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.NEAREST);
    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.NEAREST);

    // очистка текущей текстуры
    gl.bindTexture(gl.TEXTURE_2D, null);
}
```

Если забыть об использовании DOM-изображения, эти действия почти не отличаются от создания OpenGL-текстуры. Самое главное отличие — настройка формата хранения пикселя с помощью метода `gl.pixelStorei()`. Константа `gl.UNPACK_FLIP_Y_WEBGL` уникальна для WebGL и ее нужно использовать в большинстве ситуаций при загрузке изображений одного из популярных в Интернете форматов. Это объясняется тем, что системы координат GIF-, JPEG- и PNG-изображений отличаются от внутренней системы координат в WebGL. Без этой константы изображение будет показано вверх ногами.

Текстуры должны находиться в том же домене, что и страница-контейнер, или храниться на сервере, на котором для изображений включен обмен ресурсами с запросом происхождения (Cross-Origin Resource Sharing, CORS). CORS обсуждается в главе 21.

Текстуры могут быть основаны на изображениях, на видеороликах, загруженных в элемент <video>, и даже на других элементах <canvas>. На видео распространяются те же ограничения относительно исходных доменов.

Чтение пикселей

Как и при работе с двухмерным контекстом, вы можете читать пиксели из WebGL-контекста. Метод `readPixels()` принимает те же аргументы, что и в OpenGL, только его последним аргументом должен быть типизированный массив. Данные пикселей считываются из буфера кадров в типизированный массив. Аргументами метода

`readPixels()` являются координаты x и y, ширина, высота, формат изображения, тип и типизированный массив. Первые четыре аргумента показывают расположение считываемых пикселей. Форматом изображения почти всегда является `gl.RGBA`. Тип определяет тип данных, которые будут сохранены в типизированном массиве, и имеет следующие ограничения:

- если типом является `gl.UNSIGNED_BYTE`, типизированным массивом должен быть `Uint8Array`;
- если типом является `gl.UNSIGNED_SHORT_5_6_5`, `gl.UNSIGNED_SHORT_4_4_4_4` или `gl.UNSIGNED_SHORT_5_5_5_1`, типизированным массивом должен быть `Uint16Array`.

Вот простой пример:

```
var pixels = new Uint8Array(25*25);
gl.readPixels(0, 0, 25, 25, gl.RGBA, gl.UNSIGNED_BYTE, pixels);
```

Этот код читает из буфера кадров область размерами 25×25, сохраняя данные пикселей в массиве `pixels`. Цвет каждого пикселя представляется массивом из четырех элементов — по одному для красного, зеленого, синего компонентов и прозрачности. Значениями элементов могут быть числа от 0 до 255 включительно. Не забудьте инициализировать типизированный массив с учетом ожидаемого объема данных.

Вызов метода `readPixels()` до рисования обновленного WebGL-изображения выполняется без сюрпризов. По завершении рисования кадровый буфер переходит в первоначальное очищенное состояние; если после этого вызвать метод `readPixels()`, он возвратит данные очищенного буфера. Если требуется считать данные пикселей после рисования, при инициализации WebGL-контекста нужно указать уже упоминавшийся параметр `preserveDrawingBuffer`:

```
var gl = drawing.getContext("experimental-webgl",
                            { preserveDrawingBuffer: true; });
```

Этот параметр предписывает буферу кадров оставаться в последнем состоянии до следующей операции рисования. Это снижает быстродействие, так что лучше не использовать его без необходимости.

Поддержка

WebGL API реализован в Firefox 4+ и Chrome. В Safari 5.1 он также доступен, но по умолчанию отключен. Имейте в виду, что использование одного из этих браузеров не гарантирует поддержку WebGL. Кроме реализации API в браузере для этого также необходимо, чтобы его поддерживали драйверы видеокарты. На старых компьютерах, например с системой Windows XP, часто применяются устаревшие драйверы, из-за чего браузеры отключают WebGL. По этой причине нужно проверять именно поддержку WebGL, а не конкретные версии браузеров.

Спецификация WebGL все еще разрабатывается, поэтому многое в ней может измениться, в том числе имена и сигнатуры функций и типы данных. Поэкспериментировать с WebGL интересно, но использовать эту технологию в окончательном коде пока преждевременно.

Резюме

Элемент `<canvas>` из HTML5 предоставляет API, предназначенный для динамического создания графики в одном из двух специальных контекстов. Двухмерный контекст поддерживает следующие примитивные операции:

- настройка цветов и узоров заливки и контуров;
- рисование прямоугольников;
- рисование путей;
- рисование текста;
- создание градиентов и узоров.

Второй контекст, трехмерный, называется WebGL. WebGL — это браузерная версия языка OpenGL ES 2.0, который разработчики игр часто используют для программирования компьютерной графики. WebGL-контекст поддерживает гораздо более мощную функциональность, чем двухмерный контекст, в том числе:

- вершинные и фрагментные шейдеры, создаваемые на языке GLSL (OpenGL Shading Language);
- типизированные массивы, которые могут содержать только числа определенных типов;
- текстуры и различные операции над ними.

Тег `<canvas>` и двухмерный контекст доступны в последних версиях всех основных браузеров, тогда как WebGL пока поддерживается только в Firefox 4+ и Chrome.

16 HTML5

- Передача сообщений между документами
- Drag-and-Drop API
- Работа со звуком и видео

Спецификация HTML5 определяет не только новые элементы HTML-разметки, но и API для работы с ними. Предназначение этих API — облегчить решение трудных задач и в конечном итоге обеспечить возможность создания динамических веб-интерфейсов.

Передача сообщений между документами

Передача сообщений между документами (cross-document messaging, XDM) — это технология обмена информацией между страницами из разных источников. Предположим, например, что странице на сайте `www.wrox.com` нужно взаимодействовать со страницей `p2p.wrox.com`, которая загружена во встроенный фрейм. Раньше для безопасной реализации такого взаимодействия нужно было приложить немалые усилия, но теперь благодаря XDM сделать это стало гораздо проще.

В основе XDM лежит метод `postMessage()`. Он используется в HTML5 в контексте XDM всегда с одной и той же целью: для передачи данных в другое место. В случае XDM этим местом является элемент `<iframe>` или всплывающее окно, принадлежащее странице.

Метод `postMessage()` принимает два аргумента: сообщение и строку, указывающую источник предполагаемого получателя. Второй аргумент очень важен по причине безопасности, потому что он ограничивает возможности доставки сообщения. Рассмотрим пример:

```
// примечание: все браузеры, поддерживающие XDM, также поддерживают
// свойство contentWindow встроенного фрейма
var iframeWindow = document.getElementById("myframe").contentWindow;
iframeWindow.postMessage("A secret", "http://www.wrox.com");
```

Последняя строка кода пытается отправить сообщение встроенному фрейму с источником `"http://www.wrox.com"`. Если источник фрейма совпадает с указанным, сообщение доставляется, в противном случае метод `postMessage()` не делает ничего. Это ограничение защитит ваши данные, если источник окна изменится без вашего ведома. Передав в метод `postMessage()` значение `"*"` в качестве второго аргумента, можно разрешить отправку сообщений любым получателям, но делать это не рекомендуется.

Когда XDM-сообщение доставляется объекту `window`, для него генерируется событие `message`. Оно генерируется асинхронно, так что между отправкой сообщения и этим событием в окне-получателе возможна задержка. Объект `event`, передаваемый в обработчик события `message`, содержит три важных значения:

- `data` — строковые данные, переданные методу `postMessage()` в качестве первого аргумента;
- `origin` — источник документа, отправившего сообщение, например `"http://www.wrox.com"`;
- `source` — прокси-объект `window` документа, отправившего сообщение. Такой прокси обычно используется, если нужно вызвать метод `postMessage()` для окна, отправившего последнее сообщение. Если окно-отправитель имеет тот же источник, этим значением может быть фактический объект `window`.

При получении сообщения очень важно проверить источник окна-отправителя. Подобно тому как второй аргумент метода `postMessage()` исключает случайную отправку данных неизвестной странице, проверка источника в обработчике сообщения `message` позволяет убедиться, что данные получены от надежного отправителя. Базовая схема такова:

```
EventUtil.addHandler(window, "message", function(event){

    // проверка отправителя
    if (event.origin == "http://www.wrox.com"){

        // обработка данных
        processMessage(event.data);

        // необязательно: отправка сообщения исходному окну
        event.source.postMessage("Received!", "http://p2p.wrox.com");
    }
});
```

Имейте в виду, что в большинстве случаев объект `event.source` является не фактическим объектом `window`, а его прокси, так что вы не можете получить доступ ко

всем данным window. Лучше просто использовать метод postMessage(), который доступен всегда.

У технологии XDM есть несколько особенностей. Так, сначала первый аргумент метода postMessage() мог быть только строкой. Теперь им могут быть любые структурированные данные, но это изменение реализовано не во всех браузерах, поэтому лучше всегда передавать в метод postMessage() строку в качестве первого аргумента. Если нужно отправить структурированные данные, вызовите для них метод JSON.stringify(), передайте полученную строку в метод postMessage(), а затем вызовите для нее метод JSON.parse() в обработчике события message.

Технология XDM очень полезна для изоляции контента из другого домена во встроенном фрейме. Этот подход часто используется в гибридных веб-приложениях и приложениях для социальных сетей. В этом случае применение XDM для взаимодействия со встроенным фреймом защищает страницу-контейнер от вредоносного контента. XDM также можно использовать со страницами из того же домена.

Технология XDM поддерживается в Internet Explorer 8+, Firefox 3.5+, Safari 4+, Opera, Chrome, Safari для iOS и WebKit для Android. Она была вынесена в отдельную спецификацию Web Messaging, которая доступна по адресу http://dev.w3.org/html5/postmsg/.

Встроенная поддержка перетаскивания

JavaScript-механизм перетаскивания элементов веб-страниц впервые был реализован в Internet Explorer 4. Сначала перетаскивать можно было только изображения и текст. Для этого нужно было просто нажать кнопку мыши на изображении или уже выделенном фрагменте текста и, удерживая ее нажатой, переместить мышь. В Internet Explorer 4 единственным допустимым целевым элементом перетаскиваемых данных было текстовое поле. В Internet Explorer 5 добавились новые события перетаскивания, а целевыми элементами могли быть почти все элементы веб-страницы. В Internet Explorer 5.5 и 6 перетаскивание поддерживали почти все элементы. Реализация перетаскивания в Internet Explorer легла в основу соответствующего раздела спецификации HTML5, который был встроен в Firefox 3.5, Safari 3+ и Chrome.

Пожалуй, наиболее интересно в перетаскивании то, что элементы можно перетаскивать между разными фреймами и окнами браузера, а иногда даже между браузером и другими приложениями. Вы тоже можете использовать эти возможности в своем коде.

События перетаскивания

События перетаскивания позволяют контролировать практически каждый его аспект, но при этом нужно четко понимать, что одни события генерируются для

перетаскиваемого объекта, а другие — для его целевого элемента. При перетаскивании объекта возникают следующие события:

1. `dragstart`.
2. `drag`.
3. `dragend`.

Когда пользователь нажимает кнопку мыши и начинает перемещать мышь, для перетаскиваемого объекта генерируется событие `dragstart`. Указатель мыши при этом изменяется на перечеркнутую окружность, которая выводится на экран, если объект находится не на потенциальном целевом элементе. Для выполнения JavaScript-кода в начале перетаскивания можно использовать обработчик события `dragstart`.

После события `dragstart` в течение перетаскивания многократно генерируется событие `drag`. В этом смысле оно похоже на событие `mousemove`, которое тоже генерируется раз за разом, пока мышь перемещается. При завершении перетаскивания (в результате отпускания объекта на допустимом целевом элементе или любом другом месте) возникает событие `dragend`.

Целевым элементом всех трех событий является перетаскиваемый объект. По умолчанию его вид во время перетаскивания не меняется, так что при необходимости вы должны сделать это сами. Вместо этого большинство браузеров показывают полупрозрачную копию перетаскиваемого объекта, которая всегда перемещается вместе с указателем мыши.

При перетаскивании объекта на допустимый целевой элемент возникают следующие события:

1. `dragenter`.
2. `dragover`.
3. `dragleave` или `drop`.

Подобно событию `mouseover`, событие `dragenter` возникает, как только перетаскиваемый объект появляется на целевом элементе. Сразу же после этого возникает событие `dragover`, которое продолжает генерироваться, пока объект находится на целевом элементе. Когда объект покидает границы целевого элемента, оно прекращает генерироваться и возникает событие `dragleave` (похожее на `mouseout`). Если отпустить перетаскиваемый объект на целевой элемент, вместо `dragleave` будет сгенерировано событие `drop`. Целевым элементом этих событий является получатель перетаскиваемого объекта.

Пользовательские целевые элементы для данных

Если перетаскиваемый объект не находится на допустимом целевом элементе, указатель мыши превращается в перечеркнутую окружность. Если отпустить объект, когда указатель имеет такой вид, ничего не произойдет. Несмотря на то

что все элементы поддерживают события целевого элемента, по умолчанию они не принимают перетаскиваемые данные. При перетаскивании чего-либо на такой элемент событие `drop` не возникает, что бы ни делал пользователь. Чтобы сделать элемент допустимым целевым элементом перетаскиваемых данных, следует переопределить действия, предлагаемые по умолчанию для событий `dragenter` и `dragover`. Например, следующий код делает целевым элемент `<div>` с идентификатором `"droptarget"`:

```
var droptarget = document.getElementById("droptarget");

EventUtil.addHandler(droptarget, "dragover", function(event){
    EventUtil.preventDefault(event);
});

EventUtil.addHandler(droptarget, "dragenter", function(event){
    EventUtil.preventDefault(event);
});
```

После этих изменений при перетаскивании объекта на элемент `<div>` указатель мыши своим видом проинформирует о возможности отпустить объект, а целевой элемент начнет генерировать событие `drop`.

В Firefox 3.5+ событие `drop` вызывает по умолчанию переход по URL-адресу целевого элемента перетаскивания. Иначе говоря, если целевым элементом перетаскивания является изображение, загружается файл изображения, если текст — выводится сообщение о том, что URL-адрес недействителен. Чтобы обеспечить поддержку Firefox, нужно отменить действие, предлагаемое по умолчанию для события `drop`:

```
EventUtil.addHandler(droptarget, "drop", function(event){
    EventUtil.preventDefault(event);
});
```

Объект dataTransfer

Перетаскивание бесполезно, если оно никак не влияет на данные, поэтому в Internet Explorer 5 был добавлен объект `dataTransfer`, который используется для передачи строковых данных от перетаскиваемого к целевому элементу. Он является свойством объекта `event`, а потому доступен только в области видимости обработчика события перетаскивания. В обработчике можно использовать свойства и методы объекта `dataTransfer` для работы с различными аспектами перетаскивания. Этот объект входит в рабочий проект HTML5.

Главные методы объекта `dataTransfer` называются `getData()` и `setData()`. Как нетрудно догадаться, метод `getData()` получает значение, сохраненное методом `setData()`. Первым аргументом `setData()` и единственным аргументом `getData()` является строка `"text"` или `"URL"`, задающая тип данных, например:

```
// работа с текстом
event.dataTransfer.setData("text", "some text");
var text = event.dataTransfer.getData("text");

// работа с URL-адресом
event.dataTransfer.setData("URL", "http://www.wrox.com/");
var url = event.dataTransfer.getData("URL");
```

Хотя браузер Internet Explorer изначально поддерживал только типы `"text"` и `"URL"`, в HTML5 разрешено передавать в эти методы любой MIME-тип. Значения `"text"` и `"URL"` заменены в HTML5 типами `"text/plain"` и `"text/uri-list"`, но тоже поддерживаются ради обратной совместимости.

Объект `dataTransfer` может содержать одно значение каждого MIME-типа, то есть в нем можно хранить одновременно текст и URL-адрес. Данные в объекте доступны только до события `drop`. Если не получить их в обработчике `ondrop`, объект `dataTransfer` уничтожается и данные теряются.

При перетаскивании текста из текстового поля браузер вызывает метод `setData()` и сохраняет данные в формате `"text"`, а при перетаскивании ссылки или изображения методу `setData()` передается URL-адрес. При отпускании данных на целевом элементе эти значения можно получить методом `getData()`. Вы также можете вызвать метод `setData()` вручную в обработчике события `dragstart`, чтобы сохранить собственные данные, которые могут потребоваться позже.

Данные, сохраненные как текст, не подвергаются специальной обработке. В то же время данные, сохраненные как URL-адрес, интерпретируются как ссылка: если перетащить их в другое окно браузера, загружается соответствующая страница.

Firefox до версии 5 включительно не сопоставляет значение `"url"` с типом `"text/uri-list"`, а `"text"` — с `"text/plain"`. Однако он сопоставляет с типом `"text/plain"` значение `"Text"` (с прописной буквой «T»). Таким образом, в кроссбраузерном коде для чтения URL-адресов из объекта `dataTransfer` следует использовать оба варианта, а для чтения обычного текста — значение `"Text"`:

DataTransferExample01.htm


```
var dataTransfer = event.dataTransfer;

// чтение URL-адреса
var url = dataTransfer.getData("url") ||
          dataTransfer.getData("text/uri-list");

// чтение текста
var text = dataTransfer.getData("Text");
```

Первым нужно указывать сокращенное имя типа данных, потому что Internet Explorer до версии 10 включительно не поддерживает расширенные имена и генерирует ошибку, если не удается распознать имя типа.

Свойства dropEffect и effectAllowed

Объект `dataTransfer` позволяет не только передать данные, но и узнать, какие действия можно выполнить с перетаскиваемым объектом и целевым элементом. Эти сведения можно получить с помощью свойств `dropEffect` и `effectAllowed`.

Свойство `dropEffect` информирует браузер о разрешенных действиях при отпускании объекта и поддерживает четыре значения:

- `"none"` — перетаскиваемый объект не может быть принят текущим целевым элементом (это значение, предлагаемое по умолчанию для всех элементов, кроме текстовых полей);
- `"move"` — перетаскиваемый объект должен быть перемещен в целевой элемент;
- `"copy"` — перетаскиваемый объект должен быть скопирован в целевой элемент;
- `"link"` — целевой элемент должен загрузить перетаскиваемый объект (только если это URL-адрес).

Каждое из этих значений изменяет вид указателя мыши при перетаскивании объекта на целевой элемент, но выполнять соответствующие действия вы должны сами. Иначе говоря, ничто автоматически не перемещается, не копируется и не загружается без вашего вмешательства — по умолчанию браузер только изменяет вид указателя мыши. Чтобы использовать свойство `dropEffect`, нужно задать его для целевого элемента в обработчике события `dragenter`.

Свойство `dropEffect` бесполезно, если вместе с ним не используется свойство `effectAllowed`, которое указывает, какие действия `dropEffect` могут быть выполнены для перетаскиваемого объекта при его отпускании:

- `"uninitialized"` — никакое действие не задано для перетаскиваемого элемента;
- `"none"` — никакое действие не разрешено для перетаскиваемого элемента;
- `"copy"` — разрешено только действие `"copy"`;
- `"link"` — разрешено только действие `"link"`;
- `"move"` — разрешено только действие `"move"`;
- `"copyLink"` — разрешены действия `"copy"` и `"link"`;
- `"copyMove"` — разрешены действия `"copy"` и `"move"`;
- `"linkMove"` — разрешены действия `"link"` и `"move"`;
- `"all"` — разрешены все действия `dropEffect`.

Это свойство должно быть задано в обработчике события `dragstart`.

Например, чтобы пользователь мог переместить текст из текстового поля в элемент `<div>`, нужно присвоить свойствам `dropEffect` и `effectAllowed` значение `"move"`. Однако текст не переместится автоматически, потому что по умолчанию обработчик

события drop элемента ничего не делает. Если переопределить его поведение, предлагаемое по умолчанию, текст автоматически будет удален из текстового поля, но для завершения операции мы должны еще вставить его в элемент . Если присвоить свойствам dropEffect и effectAllowed значение "copy", браузер не будет автоматически удалять текст из текстового поля.

В Firefox до версии 5 включительно есть дефект, из-за которого установка свойства effectAllowed в коде может нарушать генерирование события drop.

Возможность перетаскивания

По умолчанию изображения, ссылки и текст можно перетаскивать без какого-либо дополнительного кода. Текст можно перетаскивать только после его выделения, а изображения и ссылки — когда угодно.

Можно также разрешить перетаскивание других элементов. В HTML5 у всех элементов есть свойство draggable, которое указывает, можно ли их перетаскивать. У изображений и ссылок оно по умолчанию равно true, а у всех остальных элементов — false. Изменив это значение, можно разрешить перетаскивание любого элемента или запретить перетаскивание изображения или ссылки, например:

```
<!-- отключение перетаскивания для изображения -->
<img src="smile.gif" draggable="false" alt="Smiley face">

<!-- включение перетаскивания для элемента -->
<div draggable="true">...</div>
```

Атрибут draggable поддерживается в Internet Explorer 10+, Firefox 4+, Safari 5+ и Chrome. Opera до версии 11.5 включительно не поддерживает HTML5-перетаскивание. Чтобы инициировать перетаскивание в Firefox, нужно добавить обработчик события dragstart для настройки объекта dataTransfer.

В Internet Explorer 9 и более ранних версий можно разрешить перетаскивание любого элемента, вызвав для него метод dragDrop() в обработчике события mousedown. В Safari 4 и более ранних версий для включения режима перетаскивания элемента требовалось добавить CSS-стиль -khtml-user-drag: element.

Дополнительные члены

В спецификации HTML5 определены также методы для объекта dataTransfer:

- addElement(*элемент*) — добавляет элемент в операцию перетаскивания. Это выполняется исключительно для информирования и не влияет на вид перетаскиваемого элемента. На день написания этого текста ни один браузер не поддерживает этот метод.

- `clearData(`*формат*`)` — очищает данные, хранящиеся в конкретном формате. Этот метод реализован в Internet Explorer, Firefox 3.5+, Chrome и Safari 4+.
- `setDragImage(`*элемент*`,`*x*`,`*y*`)` — позволяет указать элемент, показываемый под указателем мыши во время перетаскивания. Этот метод принимает три аргумента: HTML-элемент, который нужно показать, и координаты указателя мыши относительно этого элемента. HTML-элемент может быть изображением или любым другим элементом. Метод поддерживается в Firefox 3.5+, Safari 4+ и Chrome.
- `types` — список типов данных, хранящихся в объекте в текущий момент. Эта коллекция действует как массив, а типы данных хранятся в нем как строки (например, `"text"`). Свойство реализовано в Internet Explorer 10+, Firefox 3.5+ и Chrome.

Элементы для медиафайлов

Чтобы обеспечить доступ к аудио- и видеофайлам на веб-сайтах для максимально широкой аудитории, большинство производителей контента ранее использовали технологию Flash, но в HTML5 представлены элементы `<audio>` и `<video>` для кроссбраузерного внедрения медиафайлов без подключаемых модулей.

Эти элементы позволяют легко встраивать медиафайлы в страницы и создавать для них на JavaScript пользовательские элементы управления. Вот примеры разметки с ними:

```
<!-- встраивание видеофайла -->
<video src="conference.mpg" id="myVideo">Video player not available.</video>

<!-- встраивание аудиофайла -->
<audio src="song.mp3" id="myAudio">Audio player not available.</audio>
```

Каждый из этих элементов должен содержать как минимум атрибут `src`, задающий файл, который нужно загрузить. Также можно задать атрибуты `width` и `height`, определяющие размеры видеоплеера, и атрибут `poster` с URI изображения, которое должно присутствовать на экране во время загрузки видео. Если указан атрибут `controls`, браузер отображает пользовательский интерфейс, чтобы пользователь мог взаимодействовать с медиафайлом. Любое содержимое между открывающим и закрывающим тегами этих элементов выводится не экран, если плеер недоступен.

Можно указать несколько источников файла на тот случай, если браузер не поддерживает конкретный формат. Для этого атрибут `src` следует заменить одним или несколькими элементами `<source>`, например:

```
<!-- встраивание видеофайла -->
<video id="myVideo">
  <source src="conference.webm" type="video/webm; codecs='vp8, vorbis'">
  <source src="conference.ogv" type="video/ogg; codecs='theora, vorbis'">
```

```
  <source src="conference.mpg">
  Video player not available.
</video>

<!-- встраивание аудиофайла -->
<audio id="myAudio">
  <source src="song.ogg" type="audio/ogg">
  <source src="song.mp3" type="audio/mpeg">
  Audio player not available.
</audio>
```

Обсуждать различные аудио- и видеокодеки в этой книге мы не будем, но помните, что браузеры поддерживают разные наборы кодеков, поэтому обычно лучше использовать несколько форматов. Мультимедийные элементы поддерживаются в Internet Explorer 9+, Firefox 3.5+, Safari 4+, Opera 10.5+, Chrome, Safari для iOS и WebKit для Android.

Свойства

Элементы `<video>` и `<audio>` предоставляют достаточно развитые API. Общие для них свойства, с помощью которых можно определить текущее состояние медиафайла, указаны в таблице.

Имя свойства	Тип данных	Описание
autoplay	Boolean	Получает или устанавливает флаг autoplay
buffered	TimeRanges	Объект, определяющий уже загруженные буферизованные диапазоны времени
bufferedBytes	ByteRanges	Объект, определяющий уже загруженные буферизованные диапазоны байтов
bufferingRate	Integer	Средняя скорость загрузки файла в битах в секунду
bufferingThrottled	Boolean	Указывает, регулировал ли браузер буферизацию
controls	Boolean	Получает или устанавливает атрибут controls, который показывает или скрывает встроенные элементы управления браузера
currentLoop	Integer	Количество циклов воспроизведения медиафайла на текущий момент
currentSrc	String	URL-адрес воспроизводимого медиафайла
currentTime	Float	Текущая позиция в медиафайле в секундах
defaultPlaybackRate	Float	Получает или устанавливает скорость воспроизведения, предлагаемую по умолчанию (значение по умолчанию — 1.0)
duration	Float	Длительность медиафайла в секундах
ended	Boolean	Указывает, воспроизведен ли медиафайл полностью

Имя свойства	Тип данных	Описание
loop	Boolean	Получает или устанавливает параметр запуска повторного воспроизведения медиафайла при его завершении
muted	Boolean	Получает или устанавливает параметр отключения звука
networkState	Integer	Указывает текущее состояние сетевого подключения для медиафайла: 0 — элемент пуст; 1 — идет загрузка; 2 — загружаются метаданные; 3 — загружается первый кадр; 4 — файл загружен
paused	Boolean	Указывает, приостановлено ли воспроизведение
playbackRate	Float	Получает или устанавливает текущую скорость воспроизведения. В отличие от свойства defaultPlaybackRate, которое может изменить только разработчик, этим значением может управлять пользователь
played	TimeRanges	Воспроизведенные диапазоны времени
readyState	Integer	Указывает, готов ли медиафайл к воспроизведению. Возможные значения: 0 — данные недоступны; 1 — можно отобразить текущий кадр; 2 — можно начать воспроизведение медиафайла; 3 — можно воспроизвести медиафайл от начала до конца
seekable	TimeRanges	Диапазоны времени, доступные для поиска
seeking	Boolean	Указывает, что плеер перемещается к новой позиции в медиафайле
src	String	Источник медиафайла. Его можно перезаписать в любое время
start	Float	Получает или задает место в медиафайле, где должно начаться воспроизведение (в секундах)
totalBytes	Integer	Общее количество байтов, необходимых для ресурса (если известно)
videoHeight	Integer	Возвращает высоту видео, которая может не совпадать с высотой элемента (только для элемента <video>)
videoWidth	Integer	Возвращает ширину видео, которая может не совпадать с шириной элемента (только для элемента <video>)
volume	Float	Получает или задает текущую громкость (от 0.0 до 1.0)

Многие из этих свойств можно также задать в качестве атрибутов элемента `<audio>` или `<video>`.

События

Элементы <audio> и <video> поддерживают также целый ряд событий, которые уведомляют об изменениях свойств при воспроизведении файлов и взаимодействии пользователя с плеером. Эти события перечислены в таблице.

Имя события	Условие возникновения
abort	Загрузка файла из Интернета была прервана
canplay	Воспроизведение может быть начато; readyState = 2
canplaythrough	Воспроизведение может быть продолжено без пауз; readyState = 3
canshowcurrentframe	Текущий кадр загружен из Интернета; readyState = 1
dataunavailable	Воспроизведение невозможно из-за отсутствия данных; readyState = 0
durationchange	Свойство duration изменилось
emptied	Сетевое подключение закрыто
empty	Произошла ошибка, препятствующая загрузке медиафайла
ended	Воспроизведение медиафайла завершено
error	Во время загрузки файла из Интернета произошла сетевая ошибка
load	Медиафайл полностью загружен. Это событие устарело, используйте вместо него событие canplaythrough
loadeddata	Загружен первый кадр медиафайла
loadedmetadata	Загружены метаданные медиафайла
loadstart	Началась загрузка медиафайла из Интернета
pause	Воспроизведение приостановлено
play	Запрошено начало воспроизведения медиафайла
playing	Началось воспроизведение медиафайла
progress	Выполняется загрузка медиафайла из Интернета
ratechange	Изменилась скорость воспроизведения медиафайла
seeked	Поиск завершен
seeking	Позиция воспроизведения изменяется
stalled	Браузер пытается загрузить медиафайл из Интернета, но не получает данные
timeupdate	Значение currentTime обновлено нестандартным или непредвиденным образом
volumechange	Значение свойства volume или muted изменилось
waiting	Воспроизведение приостановлено для загрузки данных

Эти события сделаны максимально конкретными, чтобы веб-разработчики могли создавать собственные аудио- и видеоплееры, используя почти исключительно HTML и JavaScript (в отличие от Flash-роликов).

Пользовательские плееры

Для элементов `<audio>` и `<video>` доступны методы `play()` и `pause()`, позволяющие вручную контролировать воспроизведение медиафайла. Используя описанные ранее свойства и события, а также эти методы, можно легко создать собственный медиаплеер. Его разметка может быть следующей:

VideoPlayerExample01.htm


```
<div class="mediaplayer">
    <div class="video">
        <video id="player" src="movie.mov" poster="mymovie.jpg"
               width="300" height="200">
            Video player not available.
        </video>
    </div>
    <div class="controls">
        <input type="button" value="Пуск" id="video-btn">
        <span id="curtime">0</span>/<span id="duration">0</span>
    </div>
</div>
```

Для реализации простого плеера можно использовать следующий JavaScript-код:

VideoPlayerExample01.htm

```
// получение ссылок на элементы
var player = document.getElementById("player"),
    btn = document.getElementById("video-btn"),
    curtime = document.getElementById("curtime"),
    duration = document.getElementById("duration");

// обновление длительности
duration.innerHTML = player.duration;

// подключение обработчика события к кнопке
EventUtil.addHandler(btn, "click", function(event){
    if (player.paused){
        player.play();
        btn.value = "Pause";
    } else {
        player.pause();
        btn.value = "Play";
    }
});

// периодическое обновление текущего времени
setInterval(function(){
    curtime.innerHTML = player.currentTime;
}, 250);
```

Этот код просто подключает к кнопке обработчик события, который приостанавливает или возобновляет воспроизведение видео в зависимости от текущего состояния плеера. При загрузке элемента `<video>` отображается длительность видео, а в конце фрагмента настраивается таймер для периодического обновления текущего времени. Вы можете расширить возможности этого плеера, создав обработчики других событий и задействовав больше свойств. Точно такой же код можно использовать с элементом `<audio>` для создания аудиоплеера.

Распознавание кодеков

Как уже отмечалось, из-за того что браузеры поддерживают не все аудио- и видеокодеки, желательно указывать несколько вариантов медиафайлов. Кроме того, с помощью JavaScript-кода можно определить, поддерживает ли браузер конкретные формат и кодек. У обоих элементов для медиафайлов есть метод `canPlayType()`, который принимает формат/кодек в строковом виде и возвращает строку `"probably"`, («скорее всего») `"maybe"` («возможно») или `""` (пустая строка). Пустая строка при необходимости приводится к значению `false`, что позволяет использовать метод `canPlayType()` в условии `if`:

```
if (audio.canPlayType("audio/mpeg")){
    // какие-то действия
}
```

Строки `"probably"` и `"maybe"` в контексте инструкции `if` приводятся к значению `true`.

Если в метод `canPlayType()` передается только MIME-тип, метод обычно возвращает `"maybe"` или пустую строку. Это объясняется тем, что файл на самом деле является лишь контейнером аудио- или видеоданных, а возможность его воспроизведения зависит от кодировки. Если указаны и MIME-тип, и кодек, повышается вероятность возвращения строки `"probably"`, например:

```
var audio = document.getElementById("audio-player");

// наиболее вероятный результат - "maybe"
if (audio.canPlayType("audio/mpeg")){
    // какие-то действия
}

// может быть возвращено значение "probably"
if (audio.canPlayType("audio/ogg; codecs=\"vorbis\"")){
    // какие-то действия
}
```

Чтобы метод работал правильно, список кодеков должен быть заключен в кавычки. Некоторые из поддерживаемых аудиоформатов и кодеков указаны в таблице.

Имя	Строка	Поддерживающие браузеры
AAC	audio/mp4; codecs="mp4a.40.2"	Internet Explorer 9+, Safari 4+, Safari для iOS
MP3	audio/mpeg	Internet Explorer 9+, Chrome
Vorbis	audio/ogg; codecs="vorbis"	Firefox 3.5+, Chrome, Opera 10.5+
WAV	audio/wav; codecs="1"	Firefox 3.5+, Opera 10.5+, Chrome

Метод `canPlayType()` можно также использовать с элементами `<video>`. Некоторые из поддерживаемых видеоформатов и кодеков указаны в таблице.

Имя	Строка	Поддерживающие браузеры
H.264	video/mp4; codecs="avc1.42E01E, mp4a.40.2"	Internet Explorer 9+, Safari 4+, Safari для iOS, WebKit для Android
Theora	video/ogg; codecs="theora"	Firefox 3.5+, Opera 10.5, Chrome
WebM	video/webm; codecs="vp8, vorbis"	Firefox 4+, Opera 10.6, Chrome

Тип Audio

У элемента `<audio>` есть встроенный JavaScript-конструктор `Audio`, позволяющий воспроизвести аудиофайл в любое время. Тип `Audio` похож на `Image` тем, что он эквивалентен DOM-элементу, но не требует вставки в документ. Достаточно просто создать его экземпляр, передав в конструктор аудиофайл:

```
var audio = new Audio("sound.mp3");
EventUtil.addHandler(audio, "canplaythrough", function(event){
    audio.play();
});
```

При создании экземпляра типа `Audio` начинается загрузка указанного файла из Интернета. Как только она завершается, можно вызвать метод `play()`, чтобы начать воспроизведение звука.

В iOS при вызове метода `play()` появляется всплывающее диалоговое окно с запросом разрешения на воспроизведение звука. Если нужно воспроизвести один аудиофайл после другого, вызовите метод `play()` в обработчике события `finish`.

Управление состоянием журнала

Управление журналом — один из наиболее сложных аспектов программирования современных веб-приложений. Времена, когда каждое действие приводило к загрузке новой страницы, прошли, а это значит, что привычный способ изменения состояния

с помощью кнопок Назад и Вперед отчасти устарел. Первой мерой по решению этой проблемы стало добавление события hashchange (см. главу 13), а чтобы управлять состояниями было проще, в HTML5 был обновлен объект history.

В то время как событие hashchange просто уведомляет об изменении хэша URL-адреса, API управления состоянием позволяет изменить URL-адрес в браузере без загрузки новой страницы. Для этого используется метод history.pushState(), который принимает объект data, название нового состояния и необязательный относительный URL-адрес, например:

```
history.pushState({name: "Nicholas"}, "Nicholas' page", "nicholas.html");
```

При вызове метода pushState() сведения о состоянии записываются в стек журнала, а содержимое адресной строки браузера изменяется в соответствии с новым относительным URL-адресом. Несмотря на это изменение, браузер не отправляет запрос серверу, хотя если обратиться к свойству location.href, оно возвратит содержимое адресной строки. Второй аргумент в настоящее время не используется ни в одном браузере, так что можно указать пустую строку или короткое название состояния. Все сведения, требуемые для правильной инициализации состояния страницы, в случае надобности должен содержать первый аргумент.

Поскольку метод pushState() создает запись в журнале, при этом включается кнопка Назад. Если щелкнуть на ней, для объекта window генерируется событие popstate. Его объект event имеет свойство state, которое содержит объект, переданный в метод pushState() в качестве первого аргумента:

```
EventUtil.addHandler(window, "popstate", function(event){
    var state = event.state;
    if (state){    // при первой загрузке страницы свойство state равно null
        processState(state);
    }
});
```

Использовав эти данные, вы должны затем вернуть страницу в состояние, представленное объектом state (так как браузер не делает это автоматически). При первой загрузке страницы у нее нет состояния, так что при возврате к начальной странице с помощью кнопки Назад свойство event.state получает значение null.

Текущее состояние можно обновить методом replaceState(), аргументы которого аналогичны первым двум аргументам pushState(). Он не создает новую запись в журнале, а просто перезаписывает текущее состояние:

```
history.replaceState({name: "Greg"}, "Greg's page");
```

Управление состоянием журнала в стиле HTML5 поддерживается в Firefox 4+, Safari 5+, Opera 11.5+ и Chrome. В Safari и Chrome объект state, передаваемый в метод pushState() или replaceState(), не должен содержать DOM-элементы,

а в Firefox это разрешено. Opera также поддерживает свойство `history.state`, которое возвращает объект `state` текущего состояния.

При управлении состоянием журнала с помощью HTML5 любые «поддельные» URL-адреса, которые вы создаете методом pushState(), должны соответствовать реальным URL-адресам на веб-сервере, в противном случае при щелчке на кнопке обновления страницы возникнет ошибка 404.

Резюме

HTML5 определяет не только новые правила разметки, но и несколько JavaScript API, которые упрощают разработку улучшенных веб-интерфейсов, напоминающих приложения для настольных компьютеров. Вот краткий обзор этих API:

- Передача сообщений между документами обеспечивает взаимодействие документов из разных доменов с соблюдением принципа одинакового источника.
- Drag-and-Drop API позволяет включить для элемента режим перетаскивания и имитировать действия операционной системы при отпускании элемента. Вы также можете создавать собственные перетаскиваемые и целевые элементы.
- Новые элементы `<audio>` и `<video>` имеют собственные API для взаимодействия с аудио- и видеофайлами. С помощью метода `canPlayType()` можно узнать, поддерживает ли браузер конкретный формат медиафайла.
- Управление состоянием журнала позволяет изменять стек журнала браузера без выгрузки текущей страницы. Благодаря этому перемещения между состояниями страниц с помощью кнопок Назад и Вперед можно обрабатывать на чистом JavaScript.

17 Обработка ошибок и отладка

- Уведомления об ошибках
- Обработка ошибок
- Отладка JavaScript-кода

Из-за динамической типизации языка и многолетнего отсутствия полноценных средств разработки отладка JavaScript-кода традиционно требовала немалых усилий. Когда возникали ошибки, браузеры обычно отделывались туманными сообщениями, и хорошо, если при этом удавалось получить хоть какие-то сведения о контексте проблемы. Чтобы помочь разработчикам обрабатывать ошибки при их возникновении, в третью редакцию ECMAScript были добавлены инструкции `try-catch` и `throw`, а также разные типы ошибок. Несколькими годами позже начали появляться отладчики и отладочные средства JavaScript для браузеров, и к 2008 году большинство браузеров поддерживали те или иные возможности отладки JavaScript-кода.

Благодаря адекватной поддержке со стороны языка и адекватным средствам разработки теперь можно правильно обрабатывать ошибки и эффективно искать источники проблем.

Уведомления об ошибках

Все основные веб-браузеры[1] — Internet Explorer, Firefox, Safari, Chrome и Opera — поддерживают тот или иной способ уведомления о JavaScript-ошибках. По

[1] Обратите внимание, что интерфейс многих браузеров к моменту выхода русского перевода изменился. — *Примеч. пер.*

умолчанию эта информация скрывается, потому что она полезна только разработчикам. При создании собственных JavaScript-решений не забудьте включить режим вывода этих уведомлений.

Internet Explorer

Internet Explorer — единственный браузер, который по умолчанию выводит на экран индикатор JavaScript-ошибки. Когда она возникает, в левом нижнем углу окна браузера рядом с уведомлением об ошибке появляется желтый значок. Этот значок легко не заметить, если вы его не ожидаете. При двойном щелчке на значке появляется диалоговое окно с сообщением об ошибке и другими сведениями о ней, такими как номер строки, позиция знака, код ошибки и имя файла, которым всегда является URL-адрес текущей страницы (рис. 17.1).

Рис. 17.1

Такое стандартное уведомление вполне сгодится для обычных пользователей, но разработчикам лучше включить вывод диалогового окна при каждой ошибке. Чтобы внести это изменение, выберите в меню Tools команду Internet Options (Сервис ▸ Свойства браузера), в появившемся диалоговом окне перейдите на вкладку Advanced (Дополнительно) и установите флажок Display a notification about every script error (Показывать уведомление о каждой ошибке сценария), как показано на рис. 17.2. Щелкните на кнопке OK, чтобы сохранить изменения.

После обновления этого параметра диалоговое окно, которое обычно появляется при двойном щелчке на желтом значке, по умолчанию будет появляться при ошибке.

Если включен режим отладки сценариев (по умолчанию он отключен), а в браузере включен режим уведомления обо всех ошибках, вы можете увидеть альтернативное диалоговое окно с предложением отладить веб-страницу (рис. 17.3).

Чтобы включить режим отладки сценариев, нужно сначала установить отладчик, совместимый с Internet Explorer (в версии 8 и 9 он входит по умолчанию). Об отладчиках мы поговорим позже.

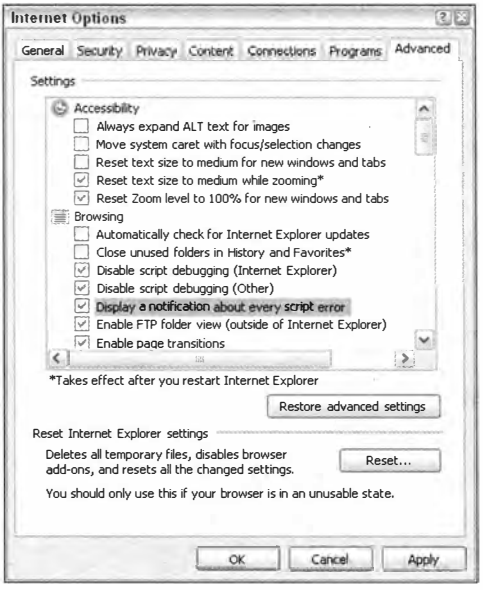

Рис. 17.2

Рис. 17.3

Если в Internet Explorer 7 и более ранних версий возникает ошибка во внешнем сценарии, номер строки в сообщении об ошибке обычно искажается на единицу. При ошибке во встроенных сценариях номер строки выводится правильно. В Internet Explorer 8 и более поздних версий все номера строк правильные.

Firefox

По умолчанию при JavaScript-ошибках пользовательский интерфейс Firefox не меняется. Вместо этого браузер просто незаметно выводит информацию об ошибке на консоль ошибок (рис. 17.4). Чтобы увидеть ее, выберите в меню Tools команду

Error Console (Инструменты ▸ Консоль ошибок). В консоли ошибок также выводятся предупреждения и информационные сообщения, касающиеся JavaScript-, CSS- и HTML-кода, так что иногда полезно фильтровать результаты.

Рис. 17.4

Когда возникает JavaScript-ошибка, она регистрируется вместе с сообщением об ошибке, URL-адресом соответствующей страницы и номером строки. При щелчке на файле открывается сценарий с ошибкой, который доступен только для чтения, при этом строка с ошибкой выделяется.

Firebug, популярное среди разработчиков дополнение браузера Firefox, меняет его режим работы, предлагаемый по умолчанию в случае JavaScript-ошибок. При установке Firebug с сайта www.getfirebug.com это дополнение включает область вывода JavaScript-информации в правую нижнюю часть строки состояния браузера Firefox. По умолчанию в этом месте выводится зеленый флажок, который при JavaScript-ошибке изменяется на красный крестик с количеством ошибок. Если щелкнуть на нем, открывается Firebug-консоль с сообщением об ошибке, строкой кода, в которой возникла ошибка (без контекста), URL-адресом страницы с ошибкой и номером строки (рис. 17.5).

При щелчке на строке с ошибкой в Firebug открывается новое Firebug-представление с этой строкой, выделенной в контексте всего файла сценария.

Вывод сообщений об ошибках — всего лишь одна из множества возможностей Firebug. Это полнофункциональная отладочная среда для Firefox, позволяющая отлаживать JavaScript- и CSS-код, модель DOM и сетевые протоколы.

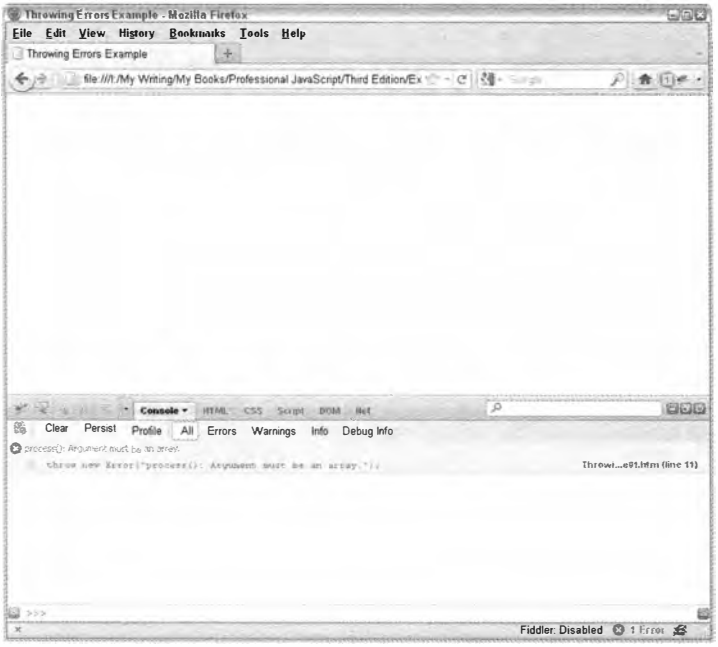

Рис. 17.5

Safari

По умолчанию Safari в Windows и Mac OS скрывает всю информацию о JavaScript-ошибках. Чтобы получить к ней доступ, нужно перейти режим разработчика, выведя меню Develop (Разработка) в строке меню. Для этого выберите в меню Edit команду Preferences (Правка ▸ Настройки), перейдите на вкладку Advanced (Дополнения) и установите флажок Show develop menu in menu bar (Показывать меню «Разработка» в строке меню). Как только это будет сделано, в строке меню браузера Safari появится меню Develop (рис. 17.6).

Меню Develop содержит несколько команд для отладки и выполнения других действий с текущей страницей. Выбрав команду Show Error Console (Показать консоль ошибок), можно вывести на экран список JavaScript-ошибок и не только. Кроме сообщения об ошибке, Safari выводит на консоль URL-адрес соответствующей страницы и номер строки с ошибкой (рис. 17.7).

При щелчке на сообщении об ошибке отображается исходный код с ней. Исключая вывод на консоль, JavaScript-ошибки никак не проявляют себя в Safari.

Opera

Opera также по умолчанию скрывает JavaScript-ошибки. Все ошибки выводятся на консоль ошибок, которую можно открыть, выбрав в меню Page команду

Рис. 17.6

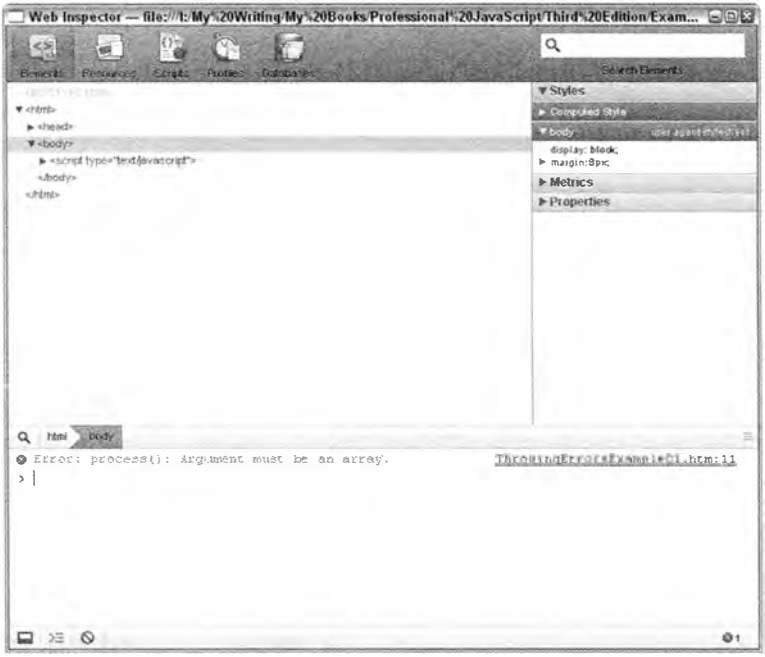

Рис. 17.7

Page ▸ Developer Tools ▸ Error Console. Как и в Firefox, на консоли браузера Opera выводятся не только JavaScript-ошибки, но и ошибки и предупреждения, относящиеся к HTML-, CSS-, XML-, XSLT-коду и ряду других источников. С помощью раскрывающихся списков в левом нижнем углу консоли можно фильтровать сообщения по типу (рис. 17.8).

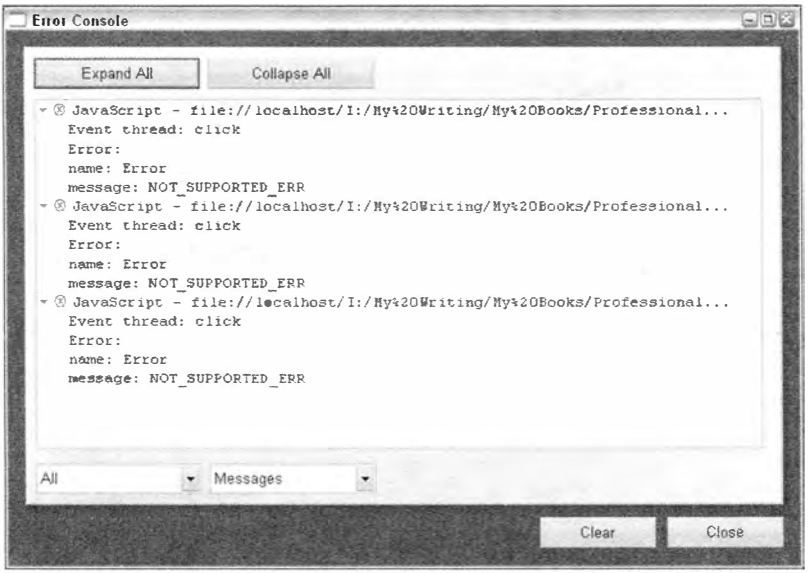

Рис. 17.8

Вместе с сообщением об ошибке выводятся URL-адрес соответствующей страницы и программный поток, в котором возникла ошибка. Иногда к ним прилагается трасса стека. Получить другие сведения об ошибке нельзя.

Можно включить режим вывода консоли при возникновении ошибок. Для этого выберите в меню Settings команду Preferences, перейдите на вкладку Advanced и выберите в левом меню пункт Content. Затем щелкните на кнопке JavaScript Options, чтобы открыть диалоговое окно JavaScript Options (рис. 17.9).

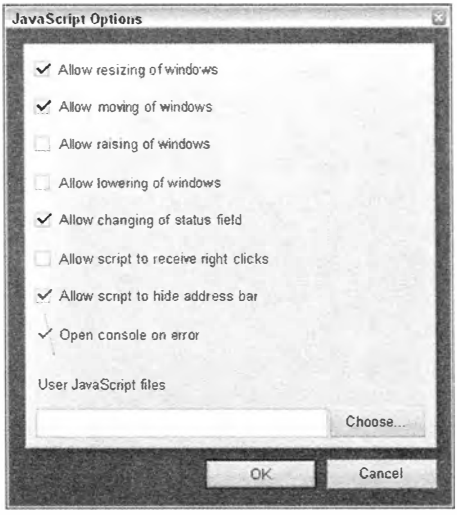

Рис. 17.9

Установите флажок Open console on error и щелкните на кнопке OK. После этого консоль ошибок будет появляться при

каждой JavaScript-ошибке. Этот режим можно также включить для отдельного сайта, для этого выберите команду Tools ▸ Quick Preferences ▸ Edit Site Preferences, перейдите на вкладку Scripting и установите флажок Open console on error.

Chrome

Подобно Safari и Opera, Chrome скрывает JavaScript-ошибки. Все ошибки регистрируются на консоли Web Inspector. Чтобы получить доступ к этой информации, нужно вручную открыть Web Inspector. Для этого щелкните на кнопке Control this page справа от адресной строки и выберите команду Developer ▸ JavaScript console (рис. 17.10).

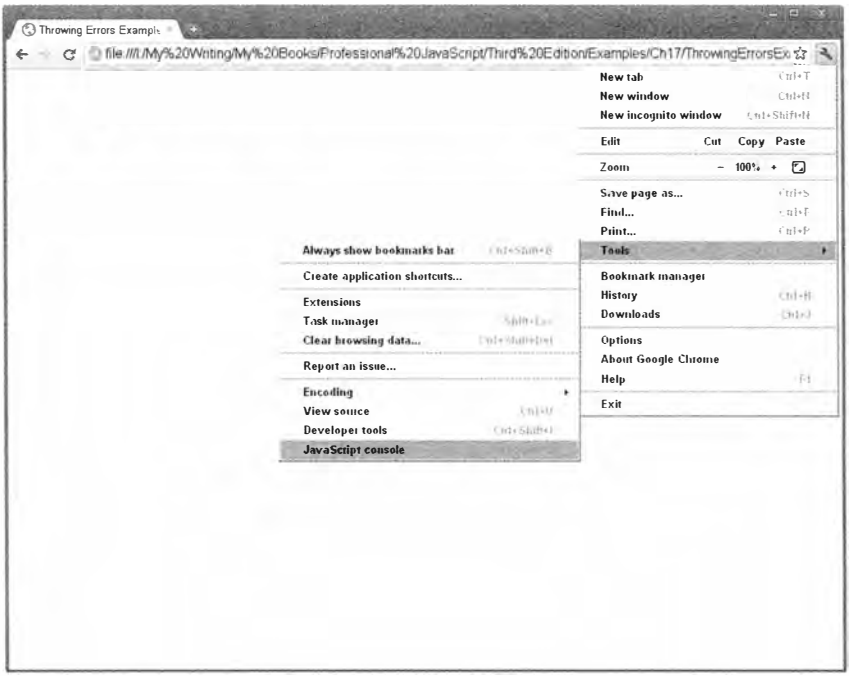

Рис. 17.10

Web Inspector содержит сведения о странице и JavaScript-консоль. Chrome выводит на консоли сообщение об ошибке, URL-адрес страницы и номер строки с ошибкой (рис. 17.11).

При щелчке на ошибке в JavaScript-консоли выполняется переход к исходному коду, который вызвал ошибку.

Рис. 17.11

Обработка ошибок

Важность обработки ошибок в программировании несомненна. Каждое серьезное веб-приложение должно следовать строгому протоколу обработки ошибок, который обычно реализуют на стороне сервера. Как правило, разработчики серверной части приложения уделяют большое внимание механизму регистрации ошибок, реализуя в нем сортировку ошибок по типу, частоте возникновения и любым другим важным критериям. Это позволяет быстро проверить работу приложения, сгенерировав отчет или выполнив запрос к базе данных.

Обработка ошибок в браузере не менее важна, хотя на осознание этого потребовалось больше времени. Нельзя забывать, что большинство посетителей веб-сайтов мало что понимают в технологиях — многие из них даже не знают, что такое веб-браузер, не говоря уж о том, какой браузер они используют. Как уже отмечалось, браузеры реагируют на JavaScript-ошибки по-разному. Небольшой значок в углу окна вполне может остаться незамеченным, ведь некоторые браузеры вообще не уведомляют

пользователей об ошибках! Столкнувшись с проблемой, типичный пользователь в лучшем случае просто обновит страницу, а в худшем — никогда больше не вернется на ваш веб-сайт. Грамотная стратегия обработки ошибок должна информировать пользователей о том, что происходит, не отпугивая их, но для реализации такой процедуры нужно уметь применять разные способы обработки JavaScript-ошибок по мере их возникновения.

Инструкция try-catch

В третьей редакции ECMA-262 для обработки исключений в JavaScript была представлена инструкция `try-catch` с таким же синтаксисом, как в Java:

```
try {
    // код, который может привести к ошибке
} catch (error) {
    // действия при возникновении ошибки
}
```

Любой код, который может привести к ошибке, следует помещать в раздел `try`, а код обработки ошибки — в раздел `catch`, например:

```
try {
    window.someNonexistentFunction();
} catch (error){
    alert("An error happened!"); // Оповещение об ошибке
}
```

Если в разделе `try` происходит ошибка, выполнение кода немедленно прекращается и возобновляется с раздела `catch`, в который передается объект со сведениями об ошибке. В отличие от других языков, вы должны назначить объекту ошибки имя, даже если не собираетесь его использовать. Содержимое этого объекта зависит от браузера, но включает как минимум свойство `message` с сообщением об ошибке. В ECMA-262 также определено свойство `name`, которое задает тип ошибки и доступно во всех современных браузерах. При необходимости можно показать фактическое сообщение браузера об ошибке:

Листинг TryCatchExample01.htm


```
try {
    window.someNonexistentFunction();
} catch (error){
    alert(error.message);
}
```

Этот код выводит в качестве сообщения об ошибке значение свойства `message`. Оно является единственным, которое гарантированно имеется в Internet Explorer, Firefox, Safari, Chrome и Opera, хотя каждый браузер добавляет к нему другие сведения. Internet Explorer добавляет свойство `description`, которое всегда совпадает с `message`,

а также свойство `number`, содержащее внутренний номер ошибки. Firefox добавляет свойства `fileName`, `lineNumber` и `stack` (которое содержит трассу стека), а Safari — `line` (номер строки), `sourceId` (внутренний код ошибки) и `sourceURL`. В кроссбраузерном коде лучше использовать только свойство `message`.

Предложение finally

Необязательное предложение `finally` инструкции `try-catch` выполняется после нее в любом случае независимо от того, возникла ли ошибка. Ничто в разделе `try` или `catch`, даже инструкция `return`, не может предотвратить выполнение кода в разделе `finally`. Рассмотрим следующую функцию:

Листинг TryCatchExample02.htm


```
function testFinally(){
    try {
        return 2;
    } catch (error){
        return 1;
    } finally {
        return 0;
    }
}
```

Эта функция содержит только инструкцию `try-catch`, каждый раздел которой возвращает число. На первый взгляд, функция должна возвратить значение 2, потому что в разделе `try` нет никаких ошибок. Однако из-за наличия предложения `finally` эта инструкция `return` игнорируется, и функция всегда возвращает 0. Если удалить предложение `finally`, функция возвратит 2.

При наличии предложения `finally` раздел `catch` не обязателен (достаточно чего-то одного). В Internet Explorer 7 и более ранних версий имеется дефект, из-за которого код в разделе `finally` никогда не выполняется, если отсутствует раздел `catch`. Если вам нужно поддерживать ранние версии Internet Explorer, просто всегда создавайте раздел `catch`, хотя бы пустой. В Internet Explorer 8 эта ошибка устранена.

Не забывайте, что при наличии предложения finally любые инструкции return в разделах try и catch игнорируются. Проверяйте такие фрагменты кода с особым вниманием.

Типы ошибок

JavaScript-ошибки делятся на несколько категорий, каждой из которых соответствует особый объект, генерируемый при ошибке. В ECMA-262 определены семь типов ошибок:

- `Error`;
- `EvalError`;
- `RangeError`;

- `ReferenceError`;
- `SyntaxError`;
- `TypeError`;
- `URIError`.

Тип `Error` — это базовый тип, от которого наследуются все остальные типы ошибок. Таким образом, все они имеют набор общих свойств (что касается методов, то у них есть только стандартные методы объекта). Ошибки типа `Error` почти никогда не генерируются браузерами; этот тип предоставляется в основном для того, чтобы разработчики могли генерировать собственные пользовательские ошибки.

Ошибка `EvalError` генерируется, когда возникает исключение при использовании функции `eval()`. В ECMA-262 сказано, что она генерируется, «если значение свойства `eval` используется каким-либо образом, отличным от непосредственного вызова (то есть отличным от явного использования имени в качестве идентификатора, который представляет собой `MemberExpression` в `CallExpression`), или если свойству `eval` присваивается значение». По сути, речь идет о ситуациях, когда `eval()` используется не как вызов функции, например:

```
new eval();         // генерирует ошибку EvalError
eval = foo;         // генерирует ошибку EvalError
```

На практике браузеры генерируют ошибку `EvalError` не всегда, когда того требует спецификация. Например, в Firefox 4+ и Internet Explorer 8 в первом случае возникает ошибка `TypeError`, а вторая строка выполняется без ошибок. По этой причине и из-за того, что такой код используется редко, едва ли вы столкнетесь с ошибками данного типа.

Ошибка `RangeError` генерируется, если число не попадает в диапазон. Например, она может возникнуть при попытке определить массив с недопустимым количеством элементов, таким как –20 или `Number.MAX_VALUE`:

```
var items1 = new Array(-20);                // генерирует ошибку RangeError
var items2 = new Array(Number.MAX_VALUE);   // генерирует ошибку RangeError
```

Ошибки диапазонов редко встречаются в JavaScript.

Ошибка `ReferenceError` генерируется, если не удается получить ожидаемый объект (это причина знаменитой ошибки `"object expected"`). Она обычно возникает при попытке доступа к несуществующей переменной, например:

```
var obj = x;  // генерирует ошибку ReferenceError, если
              // переменная x не объявлена
```

Наиболее частая причина ошибки `SyntaxError` — передача строки кода с неправильным синтаксисом в функцию `eval()`, например:

```
eval("a ++ b");    // генерирует ошибку SyntaxError
```

Вне функции `eval()` этот тип используется редко, потому что синтаксические ошибки в JavaScript-коде немедленно останавливают его выполнение.

Чаще всего JavaScript-ошибки имеют тип `TypeError`. Такие ошибки возникают, если переменная имеет недопустимый тип или предпринимается попытка вызвать несуществующий метод. Наиболее частая причина этого — использование переменной неправильного типа в типизированной операции, например:

```
var o = new 10;                            // генерирует ошибку TypeError
alert("name" in true);                     // генерирует ошибку TypeError
Function.prototype.toString.call("name");  // генерирует ошибку TypeError
```

Очень много ошибок типа `TypeError` возникает при использовании аргументов функций без предварительной проверки их типа.

Ошибки последнего типа, `URIError`, возникают только при использовании функции `encodeURI()` или `decodeURI()` с URI неправильного формата. Наверное, эти JavaScript-ошибки встречаются реже всего, потому что эти функции очень надежны.

В объектах ошибок разных типов можно передавать дополнительные сведения для дифференцированной обработки исключений. Определить тип ошибки в разделе `catch` можно с помощью оператора `instance`:

```
try {
    someFunction();
} catch (error){
    if (error instanceof TypeError){
        // обработка ошибок типа TypeError
    } else if (error instanceof ReferenceError){
        // обработка ошибок типа ReferenceError
    } else {
        // обработка всех остальных ошибок
    }
}
```

Проверка типа ошибки — самый простой кроссбраузерный способ выбора алгоритма ее обработки. Свойство `message` плохо подходит для этого, потому что его значение зависит от браузера.

Использование инструкции try-catch

Если ошибка возникает в инструкции `try-catch`, браузер не уведомляет о ней, потому что считает, что она обрабатывается должным образом. Это идеально подходит для веб-приложений, ориентированных на пользователей без технической подготовки, которым не имеет смысла сообщать об ошибках. С помощью `try-catch` можно реализовать собственный механизм обработки ошибок конкретных типов.

Инструкцию `try-catch` лучше всего использовать, когда у вас нет контроля над возможными ошибками, например если код включает вызов функции из

JavaScript-библиотеки, которую вы не можете изменить. В этом случае уместно заключить вызов функции в блок `try` и обработать возможную ошибку в блоке `catch`.

Использовать инструкцию `try-catch` для обработки ошибок в собственном коде не следует. Например, если функция принимает число и выдает ошибку, когда получает вместо него строку, нужно просто проверить в ней тип данных аргумента. Инструкция `try-catch` в этой ситуации не требуется.

Генерирование ошибок

Инструкцию `try-catch` дополняет оператор `throw`, с помощью которого в любое время можно сгенерировать собственную ошибку. Он применяется со значением, но не налагает никаких ограничений на тип значения. Все следующие варианты допустимы:

```
throw 12345;
throw "Hello world!";
throw true;
throw { name: "JavaScript"};
```

Оператор `throw` немедленно останавливает выполнение кода, которое возобновляется только в том случае, если инструкция `try-catch` перехватывает сгенерированное им значение.

Ошибки браузера можно имитировать, используя один из встроенных типов. Конструктор каждого типа ошибки принимает как единственный аргумент сообщение об ошибке, например:

```
throw new Error("Something bad happened.");
```

Этот код генерирует обобщенную ошибку, но с нашим собственным сообщением. Она обрабатывается браузером, как если бы была сгенерирована им самим, то есть браузер уведомляет о ней обычным способом, выводя указанное сообщение. То же самое возможно и с ошибками других типов, например:

```
throw new SyntaxError("I don't like your syntax.");
throw new TypeError("What type of variable do you take me for?");
throw new RangeError("Sorry, you just don't have the range.");
throw new EvalError("That doesn't evaluate.");
throw new URIError("Uri, is that you?");
throw new ReferenceError("You didn't cite your references properly.");
```

Чаще всего разработчики используют собственные сообщения об ошибках `Error`, `RangeError`, `ReferenceError` и `TypeError`.

Вы также можете создавать собственные типы ошибок, наследуя их от типа `Error` с помощью цепочек прототипов (см. главу 6). Для собственного типа ошибки нужно задать свойства `name` и `message`, например:

Листинг ThrowingErrorsExample01.htm


```
function CustomError(message){
    this.name = "CustomError";
    this.message = message;
}

CustomError.prototype = new Error();

throw new CustomError("My message");  // Нестандартное сообщение об ошибке
```

Ошибки собственных типов, унаследованных от типа `Error`, обрабатываются браузером, как и любые другие ошибки. Собственные типы ошибок полезны, если вам нужно отличать свои ошибки от ошибок, сгенерированных браузером.

Internet Explorer отображает сообщения об ошибках, заданные разработчиками, только для ошибок типа Error. При ошибках любых других типов он просто выводит сообщение «exception thrown and not caught» (необработанное исключение).

Ситуации для генерирования ошибок

Генерирование собственных ошибок — прекрасный способ предоставить дополнительные сведения о том, почему функция завершилась сбоем. Ошибки следует генерировать, если известно, что при определенных условиях функция может выполняться неправильно. Например, следующая функция не сможет решить свою задачу, если получит аргумент, отличный от массива:

Листинг ThrowingErrorsExample02.htm


```
function process(values){
    values.sort();

    for (var i=0, len=values.length; i < len; i++){
        if (values[i] > 100){
            return values[i];
        }
    }

    return -1;
}
```

Если передать этой функции строку, вызов `sort()` приведет к ошибке. Как указано далее, каждый браузер сообщит о ней по-своему:

- **Internet Explorer** — «property or method doesn't exist» (свойство или метод не существует);
- **Firefox** — «values.sort() is not a function» (values.sort() не является функцией);
- **Safari** — «value undefined (result of expression values.sort) is not an object» (неопределенное значение (результат выражения values.sort) не является объектом);

- **Chrome** — «object name has no method 'sort'» (у объекта нет метода 'sort');
- **Opera** — «type mismatch (usually a non-object value used where an object is required)» (несоответствие типов (скорее всего, там, где требуется объект, используется значение, отличное от объекта)).

Хотя Firefox, Chrome и Safari указывают, в каком месте кода произошла ошибка, ни одно из этих сообщений не описывает достаточно ясно, что случилось или как можно устранить проблему. Если нужно отладить всего одну функцию, как в этом примере, это не беда, но при работе над сложными веб-приложениями с тысячами строк кода выяснить причину ошибки по таким сообщениям гораздо труднее.

В подобной ситуации генерирование собственной ошибки с точным описанием проблемы может значительно упростить обслуживание кода, например:

Листинг ThrowingErrorsExample02.htm


```
function process(values){

    if (!(values instanceof Array)){
        throw new Error("process(): Argument must be an array.");
    }

    values.sort();

    for (var i=0, len=values.length; i < len; i++){
        if (values[i] > 100){
            return values[i];
        }
    }

    return -1;
}
```

Эта версия функции явно генерирует ошибку, если аргумент `values` не является массивом. Сообщение об ошибке содержит имя функции и ясно описывает проблему: «Argument must be an array» (аргумент должен быть массивом). Если бы эта ошибка произошла в сложном веб-приложении, вам было бы гораздо проще понять, что пошло не так.

При написании каждой JavaScript-функции нужно учитывать условия, в которых она может завершиться ошибкой. Грамотная организация обработки ошибок позволит вам работать только с теми ошибками, которые вы генерируете сами.

Использование оператора instanceof для распознавания массивов связано с некоторыми проблемами при наличии нескольких фреймов. Подробности см. в главе 22.

Генерирование и перехват ошибок

Часто спрашивают, когда следует генерировать ошибки, а когда перехватывать их с помощью инструкции `try-catch`. Вообще говоря, ошибки генерируются на низких

уровнях архитектуры приложения, где мало известно о происходящем и адекватно обработать ошибку трудно. Если вы пишете JavaScript-библиотеку, предполагая, что ее будут использовать в самых разных приложениях, или вспомогательную функцию, которая будет вызываться в разных местах одного приложения, следует всерьез задуматься о генерировании ошибок с подробными описаниями проблем. Тогда при необходимости можно будет перехватывать ошибки в приложении и обрабатывать их должным образом.

Различие между генерированием и перехватом ошибок лучше всего выразить следующим образом: перехватывать ошибки следует, только если вы точно знаете, что делать дальше. Цель перехвата ошибки — блокировать действия, которые браузер выполняет по умолчанию. Цель генерирования ошибки — предоставить сведения о том, почему она произошла.

Событие error

Любая ошибка, не обработанная с помощью инструкции `try-catch`, генерирует событие `error` для объекта `window`. Это событие было реализовано в браузерах одним из первых, и ради обратной совместимости его формат был оставлен неизменным в Internet Explorer, Firefox и Chrome (в Opera и Safari оно не поддерживается). Вместо объекта `event` в обработчик события `error` передаются три аргумента: сообщение об ошибке, URL-адрес страницы и номер строки с ошибкой. В большинстве случаев важно только сообщение об ошибке, потому что URL-адрес совпадает с расположением документа, а номер строки может указывать место во встроенном JavaScript-сценарии или во внешнем файле. Обработчик события `error` не соответствует стандартной спецификации DOM Level 2 Events, поэтому его нужно назначать, используя DOM Level 0:

```
window.onerror = function(message, url, line){
    alert(message);
};
```

Любая ошибка (независимо от того, генерирует ли ее браузер или JavaScript-код) инициирует событие `error` и запускает этот обработчик. Затем браузер выполняет действия, предлагаемые по умолчанию, выводя сообщение об ошибке. Эти действия можно заблокировать, возвратив из обработчика значение `false`:

Листинг OnErrorExample01.htm


```
window.onerror = function(message, url, line){
    alert(message);
    return false;
};
```

По сути, возврат значения `false` превращает эту функцию в глобальную инструкцию `try-catch`, которая перехватывает в документе все необработанные ошибки времени

выполнения. Такой обработчик — последнее средство блокирования уведомлений браузера, которое в идеале никогда не должно использоваться. При грамотном применении инструкции `try-catch` никакие ошибки не должны достигать уровня браузера, а значит, и событие `error` никогда не будет генерироваться.

Браузеры обрабатывают событие error по-разному. Internet Explorer при ошибке продолжает обычное выполнение кода, при этом все переменные и данные доступны в обработчике onerror. В то же время в Firefox обычное выполнение кода при этом событии завершается, а все переменные и данные уничтожаются, что затрудняет анализ ошибки.

Изображения также поддерживают событие `error`, которое генерируется, если при доступе к атрибуту `src` не удается получить изображение в известном формате. Это событие соответствует формату DOM и возвращает в качестве целевого элемента событий объект `event` с изображением, например:

Листинг OnErrorExample02.htm


```
var image = new Image();
EventUtil.addHandler(image, "load", function(event){
    alert("Image loaded!");
});
EventUtil.addHandler(image, "error", function(event){
    alert("Image not loaded!");
});
image.src = "smilex.gif";    // изображение не существует
```

Этот код выводит оповещение, если загрузка изображения завершается ошибкой. К тому времени, когда генерируется событие `error`, загрузка изображения уже завершена и возобновить ее нельзя.

Стратегии обработки ошибок

Стратегию обработки ошибок в веб-приложениях традиционно реализуют на сервере, при этом часто тратят много времени и усилий на разработку систем регистрации и мониторинга ошибок. Эти системы нужны для анализа паттернов возникновения ошибок, чтобы можно было отследить их причины и понять, как ошибки влияют на обслуживание пользователей.

Не менее важно продумать аналогичную стратегию и для уровня JavaScript-сценариев. Любая ошибка в них может сделать работу с веб-страницей невозможной, поэтому необходимо ясно понимать, когда и почему возникают ошибки. Большинство пользователей веб-приложений не имеют технической подготовки и быстро теряются, если что-то не работает. По сути, все, что они могут сделать, — это перезагрузить страницу. Как разработчик, вы должны хорошо понимать, где и как могут возникнуть сбои, и эффективно их отслеживать.

Идентификация потенциальных источников ошибок

Самое важное при обработке ошибок — это определить места, где они могут возникнуть. Поскольку JavaScript слабо типизирован и аргументы функций в нем не проверяются, ошибки часто обнаруживают себя только при выполнении кода. Можно выделить три категории ошибок:

- ошибки приведения типов;
- ошибки типов данных;
- ошибки обмена данными.

Любые из этих ошибок чаще всего возникают в специфичных ситуациях, которые можно отслеживать.

Ошибки приведения типов

Ошибки приведения типов (type coercion errors) возникают при использовании операторов или других конструкций языка, которые автоматически изменяют тип данных значения. Две наиболее частых ошибки приведения типов возникают при сравнении значений с помощью оператора равенства (`==`) или неравенства (`!=`) и использовании неуместных значений в условиях управляющих инструкций, таких как `if`, `for` и `while`.

Операторы равенства и неравенства, которые были описаны в главе 3, перед сравнением автоматически преобразуют типы значений. Поскольку во многих нединамических языках такие же операторы выполняют обычное сравнение, разработчики часто ошибочно используют их в JavaScript аналогичным образом. Чтобы избежать приведения типов, вместо них лучше использовать операторы строгого равенства (`===`) и строгого неравенства (`!==`):

```
alert(5 == "5");     // true
alert(5 === "5");    // false
alert(1 == true);    // true
alert(1 === true);   // false
```

Этот код сначала сравнивает число 5 и строку `"5"`, используя операторы равенства и строгого равенства. Оператор равенства преобразует строку `"5"` в число 5, а затем сравнивает его с другим числом 5, что дает в результате `true`. Оператор строгого равенства учитывает, что типы двух значений различаются, и возвращает `false`. То же самое происходит со значениями 1 и `true`: оператор строгого равенства не признает их равными из-за разницы типов. Операторы строгого равенства и строгого неравенства предотвращают ошибки приведения типов, а потому предпочтительнее, чем обычные операторы равенства и неравенства.

Ошибки приведения типов также часто возникают в управляющих инструкциях. Дело в том, что инструкции вроде `if` перед выбором следующего действия автоматически преобразуют любое значение в логическое, например:

```
function concat(str1, str2, str3){
    var result = str1 + str2;
    if (str3){    // не делайте так!!!
        result += str3;
    }
    return result;
}
```

Предполагаемое назначение этой функции — конкатенация двух или трех строк. Третья строка является необязательным аргументом, поэтому инструкция `if` проверяет, передан ли он в функцию. Как отмечалось в главе 3, если именованная переменная не используется, ей автоматически присваивается значение `undefined`. Ему соответствует логическое значение `false`, так что инструкция `if` присоединяет третий аргумент к строке, только если он задан. Проблема в том, что `undefined` — не единственное значение, которое преобразуется в `false`, а строка — не единственное значение, которое преобразуется в `true`. Например, если третьим аргументом будет число 0, условие `if` окажется ложным, а если 1 — истинным.

Очень много ошибок возникает из-за того, что в условиях управляющих инструкций используются значения, типы которых отличаются от логического. Чтобы таких ошибок не было, условия должны иметь логический тип, чего можно добиться с помощью операторов равенства. Например, предыдущую функцию можно переписать следующим образом:

```
function concat(str1, str2, str3){
    var result = str1 + str2;
    if (typeof str3 == "string"){    // правильное сравнение
        result += str3;
    }
    return result;
}
```

В этой версии функции условие `if` возвращает логическое значение. Эта функция гораздо безопаснее и меньше зависит от неправильных значений.

Ошибки типов данных

Поскольку JavaScript типизирован слабо, типы переменных и аргументов функций не проверяются автоматически — ответственность за это лежит на разработчике. Ошибки типов данных чаще всего возникают при передаче неправильных значений в функции.

В предыдущем примере проверяется тип данных третьего аргумента, но не первых двух. Если функция должна возвратить строку, то передача двух чисел без третьего аргумента вызовет ошибку. Похожая проблема имеет место в следующей функции:

```
// небезопасная функция - значение, отличное от строки, приводит к ошибке
function getQueryString(url){
    var pos = url.indexOf("?");
```

```
    if (pos > -1){
        return url.substring(pos +1);
    }
    return "";
}
```

Эта функция извлекает строку запроса из полученного URL-адреса. Для этого она сначала ищет в адресе вопросительный знак с помощью метода `indexOf()`, и если его удается обнаружить, возвращает часть адреса после него методом `substring()`. Два метода, которые используются в этом примере, специфичны для строк, так что передача значения любого другого типа в функцию приведет к ошибке. Следующая простая проверка типа делает функцию более устойчивой к ошибкам:

```
function getQueryString(url){
    if (typeof url == "string"){    // более безопасный вариант
        var pos = url.indexOf("?");
        if (pos > -1){
            return url.substring(pos +1);
        }
    }
    return "";
}
```

Эта версия функции первым делом проверяет, действительно ли в нее передана строка, что предотвращает ошибки при передаче значений других типов.

Вы уже знаете, что использование значений, отличных от логических, в условиях управляющих инструкций может приводить к ошибкам приведения типов. Это плохо еще и потому, что может вызывать ошибки типов данных. Рассмотрим следующую функцию:

```
// небезопасная функция - значение, отличное от массива, приводит к ошибке
function reverseSort(values){
    if (values){    // не делайте так!!!
        values.sort();
        values.reverse();
    }
}
```

Функция `reverseSort()` сортирует массив в обратном порядке, используя методы `sort()` и `reverse()`. К сожалению, при текущем условии инструкции `if` любое значение, которое не является массивом, но преобразуется в `true`, приведет к ошибке. Другой частой ошибкой является сравнение аргумента со значением `null`:

```
// небезопасная функция - значение, отличное от массива, приводит к ошибке
function reverseSort(values){
    if (values != null){    // не делайте так!!!
        values.sort();
        values.reverse();
    }
}
```

Сравнение аргумента с `null` защищает функцию только от значений `null` и `undefined` (что эквивалентно использованию операторов равенства и неравенства). Однако это не гарантирует, что значение приемлемо в других отношениях, поэтому использовать этот прием не следует. Сравнивать переменную со значением `undefined` не рекомендуется по той же причине.

Еще один источник ошибок — распознавание только одной из нужных возможностей, например:

```
// небезопасная функция - значение, отличное от массива, приводит к ошибке
function reverseSort(values){
    if (typeof values.sort == "function"){    // не делайте так!!!
        values.sort();
        values.reverse();
    }
}
```

Этот код проверяет, есть ли у аргумента метод `sort()`, однако в функцию может быть передан объект, который содержит метод `sort()`, но не является массивом. В этом случае вызов `reverse()` приведет к ошибке. Если вы точно знаете, какого типа объект вам нужен, лучше использовать для его проверки оператор `instanceof`:

```
// безопасная функция - значение, отличное от массива, игнорируется
function reverseSort(values){
    if (values instanceof Array){    // исправлено
        values.sort();
        values.reverse();
    }
}
```

Эта версия функции `reverseSort()` безопасна. Она проверяет, является ли аргумент `values` экземпляром `Array`, и игнорирует любые значения, отличные от массива.

Вообще говоря, если значение должно быть примитивным типом, его следует проверять с помощью оператора `typeof`, а если объектом — с помощью оператора `instanceof`. В зависимости от использования функции проверка типа данных всех аргументов может быть излишней, но и пренебрегать ею не следует, особенно в общедоступных API.

Ошибки обмена данными

Благодаря Ajax (см. главу 21) веб-приложения смогли динамически загружать информацию или код в течение всего жизненного цикла, но любой обмен данными между браузером и сервером — это еще один потенциальный источник ошибок.

Ошибки обмена данными могут возникать из-за неправильного формата URL-адресов или отправляемых данных. Это обычно происходит, если данные перед отправкой серверу не кодируются с помощью метода `encodeURIComponent()`. Например, формат следующего URL-адреса недопустим:

```
http://www.yourdomain.com/?redir=http://www.someotherdomain.com?a=b&c=d
```

Этот URL-адрес можно исправить, вызвав метод `encodeURIComponent()` для части после `"redir="`. В результате получается следующее:

```
http://www.yourdomain.com/?redir=http%3A%2F%2Fwww.someotherdomain.
com%3Fa%3Db%26c%3Dd
```

Аргументы строки запроса всегда нужно обрабатывать методом `encodeURIComponent()`. Чтобы гарантировать это, можно создать для составления строки запроса специальную функцию:

```
function addQueryStringArg(url, name, value){
    if (url.indexOf("?") == -1){
        url += "?";
    } else {
        url += "&";
    }

    url += encodeURIComponent(name) + "=" + encodeURIComponent(value);
    return url;
}
```

Эта функция принимает три аргумента: URL-адрес, к которому нужно добавить аргумент строки запроса, имя аргумента и его значение. Если в URL-адресе нет вопросительного знака, функция добавляет его; если он есть, это означает, что строка запроса уже содержит аргументы, так что добавляется амперсанд. После этого функция кодирует имя и значение аргумента и добавляет их к URL-адресу. Использовать ее можно следующим образом:

```
var url = "http://www.somedomain.com";
var newUrl = addQueryStringArg(url, "redir",
                               "http://www.someotherdomain.com?a=b&c=d");
alert(newUrl);
```

Использование этой функции вместо составления URL-адресов вручную обеспечивает правильное кодирование запросов, предотвращая возможные ошибки.

Ошибки обмена данными также происходят, если сервер возвращает неожиданный ответ. Например, при динамической загрузке сценариев или стилей (см. главу 10) запрошенный ресурс иногда оказывается недоступен. В этом случае Firefox, Chrome и Safari не делают ничего, а Internet Explorer и Opera возвращают ошибку. К сожалению, при использовании таких приемов узнать об ошибке почти невозможно, хотя иногда с помощью Ajax можно получить сведения о проблеме.

Ошибки обмена данными могут также возникать при использовании технологии Ajax (см. главу 21).

Различение критичных и некритичных ошибок

Любая стратегия обработки ошибок должна давать ответ на вопрос, критична ли та или иная ошибка. Для некритичной ошибки характерно следующее:

- она не мешает пользователю решать основные задачи;
- она затрагивает только часть страницы;
- восстановление возможно;
- повторение действия может решить проблему.

Некритичные ошибки не являются серьезным поводом для беспокойства. Например, в Yahoo! Mail (http://mail.yahoo.com) есть инструмент, позволяющий пользователям отправлять SMS-сообщения. Если по какой-либо причине он не работает, это некритичная ошибка, потому что отправка SMS не входит в число основных функций приложения Yahoo! Mail. Оно предназначено для чтения и написания сообщений электронной почты, и пока пользователь может это делать, незачем его беспокоить. Некритичная ошибка — не повод отвлекать его от работы, хотя при желании вы можете заменить нерабочую область страницы каким-либо уведомлением.

В то же время для критичной ошибки характерно следующее:

- приложение не может продолжить работу;
- ошибка мешает пользователю решать основные задачи;
- ошибка влечет за собой другие ошибки.

Важно понимать, когда в JavaScript происходит критичная ошибка, чтобы можно было принять адекватные меры. Следует сразу же отправить пользователям уведомление об ошибке, чтобы дать им понять, что продолжение работы невозможно. Если для восстановления работоспособности приложения нужно перезагрузить страницу, следует сообщить об этом и предоставить кнопку перезагрузки.

Код не должен диктовать, что является, а что не является критичной ошибкой — серьезность ошибки зависит в основном от того, как она влияет на работу пользователя. В грамотно спроектированном приложении ошибка в одной части не должна влиять на части приложения, которые напрямую с ней не связаны. Например, персонализированная домашняя страница вроде My Yahoo! (http://my.yahoo.com) может содержать много независимых модулей. Для инициализации JavaScript-модулей иногда используют такой код:

```
for (var i=0, len=mods.length; i < len; i++){
    mods[i].init();        // возможна критичная ошибка
}
```

На первый взгляд, все в порядке: метод `init()` просто вызывается для каждого модуля. Проблема в том, что ошибка в методе `init()` любого из модулей помешает инициализировать все последующие модули в массиве. Если ошибка произойдет на первой же итерации, ни один модуль на странице не будет инициализирован. Это

не имеет никакого смысла, потому что все модули не зависят друг от друга. Чтобы эти ошибки инициализации перестали быть критичными, можно переписать код следующим образом:

```
for (var i=0, len=mods.length; i < len; i++){
    try {
        mods[i].init();
    } catch (ex){
        // обработка ошибки
    }
}
```

Теперь ошибка в каком-либо из модулей не помешает инициализировать другие модули. Ее можно будет обработать отдельно, не прерывая работу пользователя.

Протоколирование ошибок на сервере

Для записи и отслеживания ошибок в веб-приложениях часто используют централизованный журнал. Протоколирование и систематизация ошибок базы данных и сервера в таком журнале с помощью некоего API — уже общепринятая практика, но в сложных веб-приложениях JavaScript-ошибки также следует протоколировать на сервере. Идея в том, чтобы сохранять сведения о них в той же системе, где хранятся ошибки серверной части приложения, но помечать, что они возникли на стороне клиента. Такой подход позволяет унифицировать анализ ошибок.

Для протоколирования JavaScript-ошибок на сервере нужна страница, которая будет принимать данные из строки запроса и сохранять их в журнале ошибок. Для отправки сведений об ошибке можно использовать следующую функцию:

```
function logError(sev, msg){
    var img = new Image();
    img.src = "log.php?sev=" + encodeURIComponent(sev) + "&msg=" +
              encodeURIComponent(msg);
}
```

Функция `logError()` принимает индикатор серьезности ошибки и сообщение об ошибке. Серьезность ошибки можно обозначать самыми разными строками в зависимости от того, какую систему вы используете. Объект `Image` выбран для отправки запроса из следующих соображений:

- Он доступен во всех браузерах, даже тех, которые не поддерживают объект `XMLHttpRequest`.
- На него не распространяются ограничения взаимодействия между доменами. Если один сервер должен принимать сведения об ошибках от многих других серверов, объект `XMLHttpRequest` не будет работать.
- Это снижает вероятность сбоя во время протоколирования ошибки. Как правило, обмен данными по технологии Ajax осуществляется с помощью оболочек из

JavaScript-библиотек. Если в коде библиотеки, которую вы пытаетесь использовать для протоколирования ошибки, случится сбой, сообщение об ошибке может быть утеряно.

Если в коде используется инструкция `try-catch`, вероятно, ошибку следует запротоколировать. Это можно сделать следующим образом:

```
for (var i=0, len=mods.length; i < len; i++){
    try {
        mods[i].init();
    } catch (ex){
        logError("nonfatal", "Module init failed: " + ex.message);
    }
}
```

При неудачной инициализации модуля здесь вызывается функция `logError()`. Для указания серьезности ошибки в функцию передается значение `"nonfatal"`, а ее второй аргумент состоит из описания контекста и подлинного сообщения о JavaScript-ошибке. В сообщения, протоколируемые на сервере, желательно добавлять максимально подробные сведения о контексте ошибки, чтобы было проще определить ее причину.

Приемы отладки

Когда JavaScript-отладчиков еще не было, разработчики были вынуждены добавлять в свой код фрагменты, специально предназначенные для вывода отладочной информации. Чаще всего для этого использовались оповещения с присущими им недостатками. Так, чтобы удалить их из кода после отладки, приходилось тратить драгоценное время, а оповещения, по недосмотру оставленные в коде, раздражали пользователей. К счастью, теперь доступны другие более элегантные решения.

Вывод сообщений на консоль

В Internet Explorer 8+, Firefox, Opera, Chrome и Safari доступна JavaScript-консоль, которую можно использовать для просмотра ошибок, а сведения об ошибках можно выводить на консоль из кода. Чтобы этот подход работал в Firefox 3.6 или более ранней версии, нужно установить дополнение Firebug (www.getfirebug.com). В Internet Explorer 8+, Firefox, Chrome, Safari и Opera 10.5 для вывода данных на JavaScript-консоль используется объект `console`, у которого есть следующие методы:

- `error(`*сообщение*`)` — выводит на консоль сообщение об ошибке;
- `info(`*сообщение*`)` — выводит на консоль информационное сообщение;
- `log(`*сообщение*`)` — выводит на консоль сообщение общего характера;
- `warn(`*сообщение*`)` — выводит на консоль предупреждение.

В Internet Explorer 8, Firebug, Chrome, Safari и Opera вид сообщения на консоли ошибок зависит от того, какой метод был использован для его вывода. Сообщения об ошибках содержат красный значок, а предупреждения — желтый. Выводить сообщения на консоль можно следующим образом:

```
function sum(num1, num2){
    console.log("Entering sum(), arguments are " + num1 + "," + num2);

    console.log("Before calculation");
    var result = num1 + num2;
    console.log("After calculation");

    console.log("Exiting sum()");
    return result;
}
```

При вызове этой функции `sum()` на JavaScript-консоль выводятся несколько отладочных сообщений. В Safari консоль можно открыть в меню Develop, а в Chrome для этого нужно щелкнуть на кнопке Control this page и выбрать команду Developer ▸ JavaScript console. Для доступа к консоли дополнения Firebug нужно щелкнуть на значке в правом нижнем углу строки состояния Firefox. Консоль Internet Explorer 8 является частью расширения Developer Tools; оно доступно в меню Tools, а сама консоль находится на вкладке Script.

В Opera до версии 10.5 JavaScript-консоль была доступна только с помощью метода `opera.postError()`. Он принимает сообщение, которое нужно вывести на консоль, и используется так:

```
function sum(num1, num2){
    opera.postError("Entering sum(), arguments are " + num1 + "," + num2);

    opera.postError("Before calculation");
    var result = num1 + num2;
    opera.postError("After calculation");

    opera.postError("Exiting sum()");
    return result;
}
```

Несмотря на имя метод `opera.postError()` позволяет выводить на JavaScript-консоль любую информацию.

Также можно использовать средство LiveConnect, которое позволяет в JavaScript-коде выполнять Java-код и поддерживается в Firefox, Safari и Opera. Вывести сообщение на Java-консоль из JavaScript-кода можно так:

```
java.lang.System.out.println("Your message");
```

Таким образом, вместо метода `console.log()` или `opera.postError()` можно использовать следующую функцию:

```
function sum(num1, num2){
    java.lang.System.out.println("Entering sum(), arguments are " +
                                 num1 + "," + num2);

    java.lang.System.out.println("Before calculation");
    var result = num1 + num2;
    java.lang.System.out.println("After calculation");

    java.lang.System.out.println("Exiting sum()");
    return result;
}
```

В зависимости от системных настроек при вызове LiveConnect сразу может появиться Java-консоль. Она доступна в Firefox в меню Tools, а в Opera — в меню Tools ▸ Advanced. В интерфейсе Safari нет команды открытия Java-консоли, так что ее нужно запускать отдельно.

Для вывода данных на консоль независимо от браузера можно использовать следующую функцию:

Листинг ConsoleLoggingExample01.htm


```
function log(message){
    if (typeof console == "object"){
        console.log(message);
    } else if (typeof opera == "object"){
        opera.postError(message);
    } else if (typeof java == "object" && typeof java.lang == "object"){
        java.lang.System.out.println(message);
    }
}
```

Функция `log()` просто выбирает доступный интерфейс JavaScript-консоли. Использовать ее можно в любых браузерах следующим образом:

Листинг ConsoleLoggingExample01.htm

```
function sum(num1, num2){
    log("Entering sum(), arguments are " + num1 + "," + num2);

    log("Before calculation");
    var result = num1 + num2;
    log("After calculation");

    log("Exiting sum()");
    return result;
}
```

Вывод сообщений на JavaScript-консоль помогает отлаживать код, но в окончательном коде никаких сообщений быть не должно. При развертывании приложения их можно удалить автоматически или вручную.

Вывод сообщений на консоль предпочтительнее для отладки, чем вывод оповещений, так как оповещения прерывают выполнение программы. Это может влиять на результаты, если используются асинхронные процессы.

Вывод сообщений на страницу

Другой популярный прием отладки — это вывод отладочных сообщений на страницу. Можно использовать для этого отдельную область, всегда доступную на странице, или создавать соответствующий элемент только при необходимости. Например, функцию `log()` можно изменить так:

Листинг PageLoggingExample01.htm


```
function log(message){
    var console = document.getElementById("debuginfo");
    if (console === null){
        console = document.createElement("div");
        console.id = "debuginfo";
        console.style.background = "#dedede";
        console.style.border = "1px solid silver";
        console.style.padding = "5px";
        console.style.width = "400px";
        console.style.position = "absolute";
        console.style.right = "0px";
        console.style.top = "0px";
        document.body.appendChild(console);
    }
    console.innerHTML += "<p>" + message + "</p>";
}
```

Эта версия функции `log()` сначала проверяет, существует ли элемент для вывода отладочных сведений. Если нет, она создает элемент `<div>` и назначает ему специфический стиль, отличный от остальной страницы. После этого переданное в функцию сообщение присваивается свойству `innerHTML` элемента `<div>`. В результате оно выводится на странице в небольшой специальной области. Этот подход может быть полезен при отладке кода в Internet Explorer 7 и более ранних версий или в других браузерах, не поддерживающих JavaScript-консоль.

В окончательной программе отладочные данные не должны выводиться ни на консоль, ни на страницу.

Генерирование ошибок

Как уже отмечалось, генерирование ошибок — прекрасный способ отладки кода. Если сообщения об ошибках достаточно конкретны, беглого взгляда на уведомление может хватить для определения источника ошибки. Хорошие сообщения об ошибках должны предоставлять точные сведения о проблеме, чтобы дополнительные

действия по отладке можно было свести к минимуму. Возьмем для примера следующую функцию:

```
function divide(num1, num2){
    return num1 / num2;
}
```

Эта простая функция делит одно число на другое, но возвращает `NaN`, если один из аргументов не является числом. Неожиданное получение `NaN` в результате простых вычислений часто приводит к проблемам в веб-приложениях. Чтобы подстраховаться, можно проверять перед вычислением тип каждого аргумента:

```
function divide(num1, num2){
    if (typeof num1 != "number" || typeof num2 != "number"){
        throw new Error("divide(): Both arguments must be numbers.");
    }
    return num1 / num2;
}
```

Если какой-либо из аргументов имеет нечисловой тип, этот код генерирует ошибку, указывая в сообщении имя функции и точную причину ошибки. Увидев это сообщение, вы сразу узнаете, где начать поиск, и получите общее представление о проблеме. Это гораздо лучше, чем получить от браузера неконкретное сообщение об ошибке.

В сложных приложениях для генерирования пользовательских ошибок разработчики обычно задействуют функцию `assert()`. Она принимает условие, которое должно быть истинным, и генерирует ошибку, если оно ложно. Вот совсем простой пример:

Листинг AssertExample01.htm


```
function assert(condition, message){
    if (!condition){
        throw new Error(message);
    }
}
```

Функция `assert()` позволяет заменить несколько инструкций `if` и хорошо подходит для вывода сведений об ошибках, например:

Листинг AssertExample01.htm

```
function divide(num1, num2){
    assert(typeof num1 == "number" && typeof num2 == "number",
           "divide(): Both arguments must be numbers.");
    return num1 / num2;
}
```

Применение функции `assert()` сокращает объем кода, требуемого для генерирования пользовательских ошибок, и упрощает его чтение в сравнении с предыдущим примером.

Частые ошибки Internet Explorer

Отлаживать JavaScript-ошибки в браузере Internet Explorer традиционно непросто. Многие сообщения об ошибках в нем коротки, непонятны и ничего не говорят о контексте. В то же время Internet Explorer — самый популярный браузер, поэтому особенности обработки ошибок в нем вызывают наибольший интерес у разработчиков. В последующих разделах описано несколько часто встречающихся нетривиальных JavaScript-ошибок.

Операция прервана

В Internet Explorer до версии 8 встречается, пожалуй, одна из самых непонятных и трудных для отладки ошибок — «операция прервана». Она возникает при изменении части страницы, которая еще не полностью загружена, и проявляется в том, что браузер выводит модальное диалоговое окно с сообщением «operation aborted» (операция прервана). При щелчке на кнопке OK вся веб-страница выгружается и заменяется пустым экраном, что очень затрудняет отладку. Например, подобная ошибка возникает при обработке следующей страницы:

Листинг OperationAbortedExample01.htm

```
<!DOCTYPE html>
<html>
<head>
    <title>Operation Aborted Example</title>
</head>
<body>
    <p>The following code should cause an Operation Aborted error
       in IE versions prior to 8.</p>
    <div>
        <script type="text/javascript">
            document.body.appendChild(document.createElement("div"));
        </script>
    </div>
</body>
</html>
```

Проблемы здесь заключаются в том, что JavaScript-код пытается изменить объект `document.body` до его полной загрузки и что элемент `<script>` не является непосредственным дочерним узлом элемента `<body>`. Если говорить точнее, эта ошибка возникает, когда узел `<script>` содержится в элементе, а JavaScript-код пытается изменить родительский узел или других предков этого элемента с помощью метода `appendChild()`, свойства `innerHTML` или любого другого DOM-метода, который предполагает, что элемент полностью загружен.

Эту проблему можно обойти, дождавшись полной загрузки элемента или выбрав другой метод манипулирования DOM. Например, к объекту `document.body` часто добавляют наложения с абсолютным позиционированием на странице. Это обычно

делают с помощью метода `appendChild()`, но его можно легко заменить методом `insertBefore()`. Изменив в предыдущем примере всего одну строку, можно предотвратить ошибку прерванной операции:

Листинг OperationAbortedExample02.htm


```
<!DOCTYPE html>
<html>
<head>
    <title>Operation Aborted Example</title>
</head>
<body>
    <p> The following code should not cause an Operation Aborted error in IE
        versions prior to 8.</p>
    <div>
        <script type="text/javascript">
            document.body.insertBefore(document.createElement("div"),
                                       document.body.firstChild);
        </script>
    </div>
</body>
</html>
```

В этом примере новый элемент `<div>` добавляется в начало элемента `document.body`, а не в конец. Это не приводит к ошибке, потому что вся информация, нужная для выполнения операции, доступна при запуске сценария.

Другое решение — сделать элемент `<script>` непосредственным дочерним узлом элемента `<body>`:

Листинг OperationAbortedExample03.htm


```
<!DOCTYPE html>
<html>
<head>
    <title>Operation Aborted Example</title>
</head>
<body>
    <p> The following code should not cause an Operation Aborted error in IE
        versions prior to 8.</p>
    <div>
    </div>
    <script type="text/javascript">
        document.body.appendChild(document.createElement("div"));
    </script>
</body>
</html>
```

Здесь ошибка также не возникает, потому что сценарий изменяет свой непосредственный родительский элемент, а не элемент-предок.

В Internet Explorer 8 и более поздних версий ошибка прерванной операции не возникает. Вместо нее генерируется обычная JavaScript-ошибка с таким сообщением:

«HTML Parsing Error: Unable to modify the parent container element before the child element is closed (KB927917)» (HTML-ошибка синтаксического разбора. Не удалось изменить родительский элемент-контейнер до закрытия дочернего элемента (KB927917)).

Несмотря на изменение решается эта проблема точно так же.

Недопустимый символ

Синтаксис JavaScript содержит не все символы. Если при обработке JavaScript-файла обнаруживается неподдерживаемый символ, IE генерирует ошибку «invalid character» (недопустимый символ). Недопустимым считается любой символ, который не используется в синтаксисе. Например, символ, который выглядит как «минус», но представляется в Юникоде значением 8211 (`\u2013`), нельзя применять вместо обычного минуса (ASCII-код 45). Этот специальный символ часто автоматически вставляется в документы Microsoft Word, и если запустить код с ним в Internet Explorer, возникнет ошибка недопустимого символа. Другие браузеры ведут себя подобным образом. Firefox генерирует ошибку «illegal character», Safari сообщает о синтаксической ошибке, а Opera возвращает ошибку `ReferenceError`, потому что интерпретирует недопустимый символ как неопределенный идентификатор.

Член группы не найден

Как уже отмечалось, все DOM-объекты в Internet Explorer реализованы с помощью модели COM, а не как встроенные JavaScript-объекты. При сборке мусора это может вызывать очень странные эффекты. Ошибка «member not found» (член группы не найден) — прямое следствие несоответствия алгоритмов сборки мусора в Internet Explorer.

Эта ошибка обычно возникает при попытке присвоить значение свойству уже уничтоженного объекта. Чтобы появилось это сообщение об ошибке, объект должен быть COM-объектом. Самый показательный пример этого имеет место при работе с объектом `event`. В Internet Explorer он является свойством объекта `window`, которое создается при возникновении события и уничтожается после выполнения последнего обработчика события. Если объект `event` используется в замыкании, которое должно быть выполнено позднее, любая попытка задать свойство этого объекта приведет к этой ошибке, например:

```
document.onclick = function(){
    var event = window.event;
    setTimeout(function(){
        event.returnValue = false;    // ошибка "member not found"
    }, 1000);
};
```

Этот код назначает документу обработчик события `click`, который сохраняет ссылку на объект `window.event` в локальной переменной `event`. Затем эта переменная

используется в замыкании, которое передается в функцию `setTimeout()`. При завершении обработчика события `click` объект `event` уничтожается, в результате замыкание ссылается на объект, члены которого больше не существуют. Присвоение значения свойству `returnValue` вызывает ошибку, потому что запись в COM-объект, члены которого уже уничтожены, невозможна.

Неизвестная ошибка выполнения

Неизвестная ошибка выполнения возникает при установке свойства `innerHTML` или `outerHTML`, если блочный элемент вставляется во встроенный элемент или доступ к любому из этих свойств осуществляется в какой-либо части таблицы (`<table>`, `<tbody>` и т. д.). Например, тег `<p>` технически не может содержать элемент блочного уровня вроде `<div>`, так что следующий код вызывает неизвестную ошибку выполнения:

```
p.innerHTML = "<div>Привет</div>";    // где p содержит элемент <p>
```

Другие браузеры пытаются исправлять ошибки при вставке блочных элементов в недопустимые места, но Internet Explorer гораздо строже в этом отношении.

Синтаксическая ошибка

Большинство синтаксических ошибок в Internet Explorer возникают по банальным причинам вроде отсутствия точки с запятой или скобки. Однако иногда причина ошибки далеко не так очевидна.

Если запрошенный внешний JavaScript-файл по какой-либо причине возвращает не JavaScript-код, а что-то другое, Internet Explorer генерирует синтаксическую ошибку. Например, она возникает, если атрибут `src` элемента `<script>` ссылается на HTML-файл. В качестве источника синтаксической ошибки обычно указывается первый символ в первой строке сценария. Opera и Safari при этом также указывают на файл с ошибкой. Internet Explorer этого не делает, так что вы сами должны проверить все используемые в коде внешние JavaScript-файлы. Firefox просто игнорирует любые ошибки синтаксического анализа файлов, добавленных как JavaScript-файлы, но содержащих что-то другое.

Ошибки этого типа обычно возникают, если JavaScript-код динамически генерируется серверным компонентом. Многие серверные языки при ошибке времени выполнения автоматически вставляют HTML-код в выводимые данные, что нарушает синтаксис JavaScript. Если вам трудно отследить синтаксическую ошибку, проверьте каждый внешний JavaScript-файл на предмет того, нет ли в нем HTML-кода, добавленного сервером из-за ошибки.

Не удается найти указанный ресурс

Иногда при ошибке выводится практически бесполезное сообщение «the system cannot locate the resource specified» (системе не удается найти указанный ресурс).

Эта ошибка возникает, если JavaScript-код запрашивает ресурс по URL-адресу, длина которого превышает 2083 символа (максимум в Internet Explorer). Это ограничение длины URL-адреса распространяется не только на JavaScript-код, но и вообще на браузер Internet Explorer (другие браузеры не ограничивают длину URL-адреса так строго). Дополнительно путь, указанный в URL-адресе, ограничен 2048 символами. Эта ошибка возникает в следующем примере:

Листинг LongURLErrorExample01.htm


```
function createLongUrl(url){
    var s = "?";
    for (var i=0, len=2500; i < len; i++){
        s += "a";
    }

    return url + s;
}

var x = new XMLHttpRequest();
x.open("get", createLongUrl("http://www.somedomain.com/"), true);
x.send(null);
```

Здесь объект `XMLHttpRequest` пытается запросить URL-адрес, длина которого превышает максимально допустимую, из-за чего при вызове метода `open()` возникает ошибка. Одно из обходных решений этой проблемы — сократить именованные аргументы в строке запроса или удалить из нее ненужные данные. Другое решение — изменить метод запроса на `POST` и отправлять данные в теле запроса, а не в URL-адресе. Технология Ajax, объект `XMLHttpRequest` и проблемы вроде этой подробно обсуждаются в главе 21.

Резюме

Обработка JavaScript-ошибок — обязательное требование к современным сложным веб-приложениям. Если приложение не может предвосхитить возможные ошибки и восстановиться после них, пользователи могут остаться разочарованными. По умолчанию большинство браузеров не уведомляют о JavaScript-ошибках, так что при разработке и отладке вы должны вручную включать режим протоколирования ошибок. Тем не менее в окончательном коде никакие отладочные сообщения об ошибках выводить не следует.

Чтобы не заставлять браузер отвлекаться на JavaScript-ошибки, можно использовать следующие способы:

- В предполагаемых местах возникновения ошибок можно задействовать инструкцию `try-catch`, которая позволяет обработать ошибку надлежащим образом вместо того, чтобы доверять это браузеру.

- Другой вариант — использовать обработчик `window.onerror`, который получает все ошибки, не обработанные с помощью инструкций `try-catch` (относится только к Internet Explorer, Firefox и Chrome).

В каждом веб-приложении следует анализировать источники ошибок и выбирать в зависимости от этого оптимальные способы их обработки.

- Необходимо заблаговременно решить, какие ошибки считать критичными и некритичными.
- После этого можно определить в коде наиболее вероятные места возникновения ошибок. Некоторые частые причины JavaScript-ошибок таковы:
  - приведение типов;
  - неадекватная проверка типов данных;
  - отправка или получение неправильных данных при взаимодействии с сервером.

Для Internet Explorer, Firefox, Chrome, Opera и Safari доступны отладчики JavaScript-кода, интегрированные в браузер или загружаемые в виде дополнений. Все они поддерживают установку точек прерывания, контроль выполнения кода и просмотр значений переменных во время выполнения.

18 XML в JavaScript

- Поддержка XML DOM в браузерах
- XPath в JavaScript
- Использование XSLT-процессоров

Когда-то XML был стандартным форматом хранения структурированных данных и их передачи через Интернет. Развитие XML близко соответствовало эволюции веб-технологий, в частности DOM, которая использовалась для работы со структурами XML-данных не только в веб-браузерах, но и в приложениях для настольных компьютеров и серверов. Из-за отсутствия готовых решений многие разработчики писали собственные синтаксические XML-анализаторы на JavaScript, но с тех пор поддержка XML, XML DOM и многих связанных технологий была встроена во все браузеры.

Поддержка XML DOM в браузерах

Поскольку производители браузеров начали работать над поддержкой XML еще до создания формальных стандартов, XML-реализации в браузерах различаются. Так, в DOM Level 2 была представлена концепция динамического создания DOM-документа с XML-кодом, которая в DOM Level 3 была дополнена синтаксическим анализом и сериализацией. Однако ко времени окончания работы над DOM Level 3 большинство производителей браузеров уже реализовали в них собственные решения.

DOM Level 2 Core

Как отмечено в главе 12, у объекта `document.implementation` есть метод `createDocument()`, который поддерживается в Internet Explorer 9+, Firefox, Opera, Chrome

и Safari. Вы можете создать пустой XML-документ, используя следующий синтаксис:

```
var xmldom = document.implementation.createDocument(
                                                namespaceUri, root, doctype);
```

При работе с XML в JavaScript обычно используется только аргумент `root`, который определяет имя тега DOM-элемента `document` с XML-кодом. Аргумент `namespaceUri` применяется не часто, потому что управлять пространствами имен в JavaScript-коде сложно. Что касается аргумента `doctype`, то он требуется совсем редко.

Создать XML-документ с тегом `<root>` в качестве элемента `document` можно следующим образом:

Листинг DOMLevel2CoreExample01.htm


```
var xmldom = document.implementation.createDocument("", "root", null);

alert(xmldom.documentElement.tagName);    // "root"

var child = xmldom.createElement("child");
xmldom.documentElement.appendChild(child);
```

Этот код создает документ XML DOM без пространства имен, предлагаемого по умолчанию, и без типа документа. Даже если пространство имен и тип документа не требуются, аргументы все же должны быть переданы в метод `createDocument()`. В качестве URI пространства имен в него передается пустая строка, а в качестве типа документа — значение `null`. Переменной `xmldom` назначается экземпляр типа `Document` согласноDOM Level 2 со всеми методами и свойствами DOM, описанными в главе 12. Далее этот код выводит имя тега элемента `document`, а затем создает и добавляет к нему дочерний элемент.

Проверить, включена ли в браузере поддержка DOM Level 2 XML, можно следующим образом:

```
var hasXmlDom = document.implementation.hasFeature("XML", "2.0");
```

На практике разработчики редко создают XML-документ с нуля, наращивая его с помощью DOM-методов. Гораздо чаще приходится преобразовывать XML-документ в структуру DOM или наоборот. DOM Level 2 не предлагает такого функционала, но со временем некоторые способы решения этой задачи стали стандартами де-факто.

Тип DOMParser

В Firefox для синтаксического анализа XML-кода с преобразованием его в DOM-документ был введен тип `DOMParser`, который позднее был реализован в Internet Explorer 9, Safari, Chrome и Opera. Чтобы использовать его, нужно создать экземпляр

DOMParser и вызвать метод parseFromString(). Он принимает исходную строку XML-кода и тип содержимого, которым всегда должен быть "text/xml", а возвращает экземпляр типа Document, например:

Листинг DOMParserExample01.htm

```
var parser = new DOMParser();
var xmldom = parser.parseFromString("<root><child/></root>", "text/xml");

alert(xmldom.documentElement.tagName);    // "root"
alert(xmldom.documentElement.firstChild.tagName);    // "child"

var anotherChild = xmldom.createElement("child");
xmldom.documentElement.appendChild(anotherChild);

var children = xmldom.getElementsByTagName("child");
alert(children.length);        // 2
```

Этот код выполняет синтаксический анализ простой строки XML-кода, преобразуя ее в DOM-документ с узлом <root> в качестве элемента document и единственным дочерним элементом <child>. С возвращенным документом можно работать, используя DOM-методы.

Тип DOMParser выполняет синтаксический анализ XML-кода только правильного формата, а потому не способен преобразовать HTML-код в HTML-документ. При ошибке синтаксического анализа браузеры ведут себя по-разному. В Firefox, Opera, Safari и Chrome метод parseFromString() при ошибке все же возвращает объект Document, но его элементом document является <parsererror>, а содержимым элемента — описание ошибки, например:

```
<parsererror xmlns="http://www.mozilla.org/newlayout/xml/parsererror.xml">
Parsing Error: no element found Location
Адрес: file:///I:/My%20Writing/My%20Books/Professional%20JavaScript/
Second%20Edition/Examples/Ch15/DOMParserExample2.htm
Строка 1, символ 7:<sourcetext>&lt;root&gt;
------^</sourcetext></parsererror>
```

Firefox и Opera возвращают документы в этом формате, а в Safari и Chrome возвращаемый документ содержит элемент <parsererror> там, где произошла ошибка. Internet Explorer 9 генерирует ошибку синтаксического анализа при вызове метода parseFromString(). Из-за этих различий для выявления ошибки синтаксического анализа лучше всего воспользоваться блоком try-catch, а при отсутствии ошибки попытаться найти элемент <parsererror> в документе методом getElementsByTagName():

Листинг DOMParserExample02.htm


```
var parser = new DOMParser(),
    xmldom,
    errors;
try {
    xmldom = parser.parseFromString("<root>", "text/xml");
```

```
    errors = xmldom.getElementsByTagName("parsererror");
    if (errors.length > 0){
        throw new Error("Parsing error!"); // ошибка синтаксического анализа
    }
} catch (ex) {
    alert("Parsing error!");
}
```

В этом примере в исходной строке отсутствует закрывающий тег `</root>`. При ее обработке Internet Explorer 9+ генерирует ошибку. В Firefox и Opera элемент `<parsererror>` становится элементом `document`, а в Chrome и Safari — первым дочерним элементом узла `<root>`. Оба эти случая распознаются при вызове метода `getElementsByTagName("parsererror")`. Если он возвращает какие-либо элементы, значит, произошла ошибка, поэтому отображается оповещение о ней. При желании можно извлечь из элемента сведения об ошибке.

Тип XMLSerializer

Тип `XMLSerializer` противоположен типу `DOMParser`: он сериализует DOM-документ в XML-строку. Этот тип появился в Firefox и позднее был реализован в Internet Explorer 9+, Opera, Chrome и Safari.

Чтобы сериализовать DOM-документ, нужно создать экземпляр `XMLSerializer`, а затем вызвать метод `serializeToString()`, передав в него документ:

Листинг XMLSerializerExample01.htm


```
var serializer = new XMLSerializer();
var xml = serializer.serializeToString(xmldom);
alert(xml);
```

Метод `serializeToString()` возвращает строку без какого-либо форматирования, так что читать ее трудно.

Тип `XMLSerializer` поддерживает сериализацию любых допустимых DOM-объектов, в том числе отдельных узлов и HTML-документов. HTML-документ, переданный в метод `serializeToString()`, обрабатывается как XML-документ, так что формат итогового кода получается правильным.

Если передать в метод serializeToString() что-либо отличное от DOM-объекта, произойдет ошибка.

XML в Internet Explorer 8 и более ранних версий

На самом деле встроенная поддержка XML-кода впервые появилась в Internet Explorer и была основана на ActiveX-объектах. Корпорация Майкрософт создала MSXML-библиотеку для обработки XML-кода в приложениях для настольных

компьютеров, и вместо того чтобы создавать другие объекты для JavaScript-кода, она просто предоставила доступ из браузера к тем же объектам.

В главе 8 мы познакомились с типом `ActiveXObject`, который используется для создания ActiveX-объектов в JavaScript-коде. Экземпляр XML-документа можно создать с помощью конструктора `ActiveXObject`, передав ему строковый идентификатор версии XML-документа. Доступны шесть объектов XML-документов:

- `Microsoft.XmlDom` — первоначальный объект в IE, который не следует использовать;
- `MSXML2.DOMDocument` — обновленная версия для сценариев, используемая для страховки в крайних случаях;
- `MSXML2.DOMDocument.3.0` — наименьшая версия из рекомендованных для использования в JavaScript;
- `MSXML2.DOMDocument.4.0` — этот объект не считается безопасным в сценариях (попытка его использовать может привести к предупреждению системы безопасности);
- `MSXML2.DOMDocument 5.0` — этот объект также не считается безопасным в сценариях и может вызывать предупреждение системы безопасности;
- `MSXML2.DOMDocument.6.0` — самая свежая версия, отмеченная как безопасная в сценариях.

Корпорация Майкрософт рекомендует использовать только версию `MSXML2.DOMDocument.6.0`, которая является новейшей и самой надежной, или `MSXML2.DOMDocument.3.0`, которая доступна на большинстве компьютеров с Windows. В крайнем случае можно использовать версию `MSXML2.DOMDocument`, что может потребоваться для поддержки старых браузеров (до Internet Explorer 5.5).

Чтобы узнать, какие версии доступны, можно попробовать создать каждую из них, отслеживая ошибки, например:

Листинг IEXmlDomExample01.htm


```
function createDocument(){
    if (typeof arguments.callee.activeXString != "string"){
        var versions = ["MSXML2.DOMDocument.6.0", "MSXML2.DOMDocument.3.0",
                        "MSXML2.DOMDocument"],
            i, len;

        for (i=0,len=versions.length; i < len; i++){
            try {
                new ActiveXObject(versions[i]);
                arguments.callee.activeXString = versions[i];
                break;
            } catch (ex){
                // ничего делать не требуется
            }
        }
```

```
    }

    return new ActiveXObject(arguments.callee.activeXString);
}
```

В цикле `for` мы перебираем возможные версии ActiveX-объекта. Если версия недоступна, конструктор `ActiveXObject` генерирует ошибку, которая перехватывается инструкцией `catch`, и цикл продолжается. Если ошибка не возникает, версия сохраняется в свойстве `activeXString` функции, чтобы этот процесс не нужно было повторять при каждом вызове функции, и созданный объект возвращается.

Для преобразования XML-строки в DOM-документ нужно создать его и вызвать его метод `loadXML()` с исходной строкой в качестве аргумента, например:

Листинг IEXmlDomExample01.htm

```
var xmldom = createDocument();
xmldom.loadXML("<root><child/></root>");

alert(xmldom.documentElement.tagName);    // "root"
alert(xmldom.documentElement.firstChild.tagName);    // "child"

var anotherChild = xmldom.createElement("child");
xmldom.documentElement.appendChild(anotherChild);

var children = xmldom.getElementsByTagName("child");
alert(children.length);        // 2
```

Когда DOM-документ заполнен XML-контентом, с ним можно работать так же, как с любым другим DOM-документом, и использовать все его методы и свойства.

Ошибки преобразования представляются объектом `parseError`, который содержит несколько свойств, описывающих проблему:

- `errorCode` — числовой код, указывающий тип ошибки (0, если ошибок нет);
- `filePos` — позиция в файле, где произошла ошибка;
- `line` — номер строки с ошибкой;
- `linepos` — символ в строке, где произошла ошибка;
- `reason` — текстовое описание ошибки;
- `srcText` — код, вызвавший ошибку;
- `url` — URL-адрес файла с ошибкой (если доступен).

Метод `valueOf()` объекта `parseError` возвращает значение `errorCode`, так что проверить, произошла ли ошибка синтаксического анализа, можно следующим образом:

```
if (xmldom.parseError != 0){
    alert("Parsing error."); // оповещение об ошибке
}
```

Код ошибки может быть положительным или отрицательным числом, поэтому достаточно проверить, не равен ли он нулю. После этого можно легко получить дополнительные сведения об ошибке, например:

Листинг IEXmlDomExample02.htm


```
if (xmldom.parseError != 0){
    // Оповещение об ошибке
    alert("An error occurred:\nError Code: "
          // Код ошибки
          + xmldom.parseError.errorCode + "\n"
          // Строка с ошибкой
          + "Line: " + xmldom.parseError.line + "\n"
          // Позиция ошибки в строке
          + "Line Pos: " + xmldom.parseError.linepos + "\n"
          // Причина ошибки
          + "Reason: " + xmldom.parseError.reason);
}
```

Наличие ошибок синтаксического анализа следует проверять сразу после вызова метода `loadXML()` до попыток получить дополнительные данные из XML-документа.

Сериализация в формат XML

В Internet Explorer средства сериализации в формат XML встроены в DOM-документ. У каждого узла есть свойство `xml`, с помощью которого можно получить XML-строку, представляющую этот узел:

```
alert(xmldom.xml);
```

Этот простой способ сериализации доступен для каждого узла документа, что позволяет сериализовать весь документ или его поддерево.

Загрузка XML-файлов

С помощью объекта XML-документа в Internet Explorer можно также загружать файлы с сервера. Как и в DOM Level 3, загружаемые XML-документы должны находиться на том же сервере, что и страница с JavaScript-кодом, а загружать их можно синхронно или асинхронно. Чтобы указать режим загрузки, нужно присвоить свойству `async` значение `true` или `false` (по умолчанию оно равно `true`), например:

```
var xmldom = createDocument();
xmldom.async = false;
```

Как только режим задан, запустить загрузку можно с помощью метода `load()`, который принимает URL-адрес загружаемого XML-файла. В синхронном режиме сразу после вызова `load()` можно обработать ошибки синтаксического анализа и приступить к работе с XML-файлом, например:

Листинг IEXmlDomExample03.htm


```
var xmldom = createDocument();
xmldom.async = false;
xmldom.load("example.xml");

if (xmldom.parseError != 0){
    // обработка ошибки
} else {
    alert(xmldom.documentElement.tagName);    // "root"
    alert(xmldom.documentElement.firstChild.tagName);    // "child"

    var anotherChild = xmldom.createElement("child");
    xmldom.documentElement.appendChild(anotherChild);

    var children = xmldom.getElementsByTagName("child");
    alert(children.length);        // 2

    alert(xmldom.xml);
}
```

Поскольку XML-файл загружается синхронно, выполнение кода приостанавливается до завершения синтаксического анализа, что упрощает программирование. К сожалению, это может приводить к длительным задержкам, если загрузка занимает много времени, поэтому XML-документы обычно загружают асинхронно.

При асинхронной загрузке XML-файла нужно назначить обработчик события `readystatechange` DOM-документу с XML-кодом. Возможные состояния готовности таковы:

- 1 — DOM-данные загружаются;
- 2 — загрузка DOM-данных завершена;
- 3 — DOM можно использовать, хотя некоторые разделы могут быть недоступны;
- 4 — DOM полностью загружена и готова к использованию.

С практической точки зрения интересно только четвертое состояние, которое указывает, что XML-файл полностью загружен и преобразован в DOM-документ. Получить состояние готовности XML-документа можно с помощью свойства `readyState`. Для асинхронной загрузки XML-файла обычно используется следующая схема:

Листинг IEXmlDomExample04.htm


```
var xmldom = createDocument();
xmldom.async = true;

xmldom.onreadystatechange = function(){
    if (xmldom.readyState == 4){
        if (xmldom.parseError != 0){
        // Оповещение об ошибке
        alert("An error occurred:\nError Code: "
              // Код ошибки
              + xmldom.parseError.errorCode + "\n"
```

```
            // Строка с ошибкой
            + "Line: " + xmldom.parseError.line + "\n"
            // Позиция ошибки в строке
            + "Line Pos: " + xmldom.parseError.linepos + "\n"
            // Причина ошибки
            + "Reason: " + xmldom.parseError.reason);
        } else {
            alert(xmldom.documentElement.tagName);    // "root"
            alert(xmldom.documentElement.firstChild.tagName);    // "child"

            var anotherChild = xmldom.createElement("child");
            xmldom.documentElement.appendChild(anotherChild);

            var children = xmldom.getElementsByTagName("child");
            alert(children.length);        // 2

            alert(xmldom.xml);
        }
    }
};

xmldom.load("example.xml");
```

Имейте в виду, что обработчик события `readystatechange` нужно назначить документу до вызова `load()`, чтобы он наверняка был вызван вовремя. Заметьте также, что в обработчике события нужно использовать имя переменной XML-документа, (`xmldom` в данном случае), а не объект `this`. Использовать `this` с ActiveX-элементами управления запрещено из соображений безопасности. Как только состояние готовности документа становится 4, можно безопасно проверить наличие ошибок синтаксического анализа и начать обработку XML-документа.

Хотя загружать XML-файлы можно с помощью DOM-объекта document с XML-кодом, обычно вместо него используют объект XMLHttpRequest. Этот объект и технология Ajax в общем обсуждаются в главе 21.

Кроссбраузерная обработка XML

Мало кто может позволить себе роскошь писать код только для одного браузера, поэтому часто требуется какое-то универсальное решение для обработки XML. Следующая функция синтаксического анализа XML-кода с его преобразованием в DOM-документ работает во всех основных браузерах:

Листинг CrossBrowserXmlExample01.htm


```
function parseXml(xml){
    var xmldom = null;

    if (typeof DOMParser != "undefined"){
        xmldom = (new DOMParser()).parseFromString(xml, "text/xml");
        var errors = xmldom.getElementsByTagName("parsererror");
        if (errors.length){
```

```
            throw new Error("Parsing error XML: " +
                            errors[0].textContent);
        }

    } else if (typeof ActiveXObject != "undefined"){
        xmldom = createDocument();
        xmldom.loadXML(xml);
        if (xmldom.parseError != 0){
            throw new Error("Parsing error XML: " +
                            xmldom.parseError.reason);
        }

    } else {
        throw new Error("No XML parser available.");
    }

    return xmldom;
}
```

Функция `parseXml()` принимает XML-строку, которую нужно преобразовать в DOM, и распознает возможности браузера, чтобы выбрать те или иные средства преобразования. Наиболее широко поддерживается тип `DOMParser`, поэтому сначала функция проверяет, доступен ли он. Если да, она создает объект `DOMParser` и преобразует XML-строку, сохраняя результат в переменной `xmldom`. Так как объект `DOMParser` генерирует ошибки преобразования только в Internet Explorer 9+, возвращенный документ затем проверяется на предмет ошибок. В случае их обнаружения выполняется инструкция `throw`, которой передается сообщение об ошибке.

В следующей части функция проверяет, поддерживается ли ActiveX-объект, и если да, вызывает определенную ранее функцию `createDocument()` для создания XML-документа с правильной сигнатурой. Как и в первой части, результат проверяется на предмет ошибок преобразования. В случае их обнаружения выполняется инструкция `throw`.

Если синтаксический анализатор XML недоступен, функция просто генерирует ошибку.

Эту функцию можно использовать для преобразования любой XML-строки, при этом ее вызов всегда следует заключать в блок `try` на случай ошибок, например:

Листинг CrossBrowserXmlExample01.htm


```
var xmldom = null;

try {
    xmldom = parseXml("<root><child/></root>");
} catch (ex){
    alert(ex.message);
}

// дальнейшая обработка
```

Для сериализации документа в формат XML можно использовать следующую кроссбраузерную функцию:

Листинг CrossBrowserXmlExample02.htm

```
function serializeXml(xmldom){

    if (typeof XMLSerializer != "undefined"){
        return (new XMLSerializer()).serializeToString(xmldom);

    } else if (typeof xmldom.xml != "undefined"){
        return xmldom.xml;

    } else {
        throw new Error("Could not serialize XML DOM.");
    }
}
```

Функция `serializeXml()` принимает в качестве аргумента документ XML DOM, который требуется сериализовать. Как и функция `parseXml()`, она первым делом проверяет, доступно ли наиболее распространенное средство сериализации — тип `XMLSerializer`. Если да, вызывается его метод сериализации документа в XML-строку. В подходе с ActiveX-объектом используется свойство `xml`, поэтому функция проверяет его напрямую. Если обе попытки оказываются неудачными, генерируется ошибка с сообщением о том, что сериализовать документ не удалось. Вообще говоря, сериализация не должна завершаться ошибкой, если используется правильный объект XML DOM, так что заключать вызов `serializeXml()` в блок `try` не требуется. Вместо этого можно вызывать функцию следующим образом:

```
var xml = serializeXml(xmldom);
```

Имейте в виду, что из-за различий в логике сериализации ее результаты могут зависеть от браузера.

Поддержка XPath в браузерах

XPath был создан для доступа к конкретным узлам в DOM-документе, так что его важность для обработки XML очевидна. API для XPath впервые появился в рекомендации DOM Level 3 XPath. Эта спецификация реализована во всех основных браузерах, кроме Internet Explorer.

DOM Level 3 XPath

Спецификация DOM Level 3 XPath определяет интерфейсы для обработки XPath-выражений в DOM. Определить, поддерживает ли браузер DOM Level 3 XPath, можно следующим образом:

```
var supportsXPath = document.implementation.hasFeature("XPath", "3.0");
```

Из определенных в спецификации типов наиболее важны `XPathEvaluator` и `XPathResult`. Тип `XPathEvaluator` используется для оценки XPath-выражений в конкретном контексте и имеет три метода:

- `createExpression(`*выражение*, *объектРазрешенияПИ*`)` — вычисляет XPath-выражение и сопутствующие сведения о пространстве имен, возвращая объект `XPathExpression` — скомпилированную версию запроса. Этот метод полезен, если один запрос будет выполняться несколько раз.
- `createNSResolver(`*узел*`)` — создает объект `XPathNSResolver` на основе сведений о пространстве имен полученного узла. Объект `XPathNSResolver` требуется при оценке выражений для XML-документа, в котором используются пространства имен.
- `evaluate(`*выражение*, *контекст*, *объектРазрешенияПИ*, *тип*, *результат*`)` — оценивает XPath-выражение в указанном контексте с конкретным пространством имен. Дополнительные аргументы указывают, как должен быть возвращен результат.

В Firefox, Safari, Chrome и Opera тип `Document` обычно реализован с помощью интерфейса `XPathEvaluator`. Таким образом, вы можете либо создать новый экземпляр `XPathEvaluator`, либо использовать методы экземпляра `Document` (и с XML-, и с HTML-документами).

Из этих трех методов более всего востребован `evaluate()`. Он принимает пять аргументов: XPath-выражение, узел контекста, объект разрешения пространства имен, тип возвращаемого результата и объект `XPathResult` для представления результата (он обычно имеет значение `null`, потому что результат также возвращается как значение функции). Третий аргумент, объект разрешения пространства имен, необходим, только если в XML-коде используется XML-пространство имен. В противном случае он должен иметь значение `null`. Типом возвращаемого результата может быть одна из десяти констант.

- `XPathResult.ANY_TYPE` — возвращает тип данных, соответствующий XPath-выражению.
- `XPathResult.NUMBER_TYPE` — возвращает числовое значение,
- `XPathResult.STRING_TYPE` — возвращает строковое значение,
- `XPathResult.BOOLEAN_TYPE` — возвращает логическое значение,
- `XPathResult.UNORDERED_NODE_ITERATOR_TYPE` — возвращает множество узлов, соответствующих XPath-выражению, хотя их порядок может не соответствовать порядку узлов в документе,
- `XPathResult.ORDERED_NODE_ITERATOR_TYPE` — возвращает множество узлов, соответствующих XPath-выражению, в том порядке, в котором они располагаются в документе. Этот тип результата используется чаще всего,
- `XPathResult.UNORDERED_NODE_SNAPSHOT_TYPE` — возвращает снимок множества узлов, сохраняя их вне документа, чтобы последующие изменения документа не влияли на множество узлов. Порядок узлов в множестве может отличаться от их расположения в документе.
- `XPathResult.ORDERED_NODE_SNAPSHOT_TYPE` — возвращает снимок множества узлов, сохраняя их вне документа, чтобы последующие изменения документа не влияли

на множество результатов. Узлы в множестве результатов располагаются в том же порядке, что и в документе.

- `XPathResult.ANY_UNORDERED_NODE_TYPE` — возвращает множество узлов, соответствующих XPath-выражению, хотя их порядок может не соответствовать порядку узлов в документе.
- `XPathResult.FIRST_ORDERED_NODE_TYPE` — возвращает множество только с одним узлом, являющимся первым узлом в документе, соответствующим XPath-выражению.

От указанного типа результата зависит способ его получения. Вот типичный пример:

Листинг DomXPathExample01.htm


```
var result = xmldom.evaluate("employee/name", xmldom.documentElement, null,
                             XPathResult.ORDERED_NODE_ITERATOR_TYPE, null);

if (result !== null) {
    var element = result.iterateNext();
    while(element) {
        alert(element.tagName);
        node = result.iterateNext();
    }
}
```

Этот код запрашивает тип результата `XPathResult.ORDERED_NODE_ITERATOR_TYPE`, который используется чаще всего. Если ни один узел не соответствует XPath-выражению, метод `evaluate()` возвращает `null`, иначе возвращается объект `XPathResult`, у которого есть свойства и методы получения результатов конкретных типов. Если результатом является итератор узлов (неважно, упорядоченных или нет), для получения каждого соответствующего выражению узла нужно использовать метод `iterateNext()`. По исчерпании таких узлов метод `iterateNext()` возвращает `null`.

Если запрошен снимок множества узлов (упорядоченных или неупорядоченных), для доступа к ним требуются метод `snapshotItem()` и свойство `snapshotLength`, например:

Листинг DomXPathExample02.htm

```
var result = xmldom.evaluate("employee/name", xmldom.documentElement, null,
                             XPathResult.ORDERED_NODE_SNAPSHOT_TYPE, null);

if (result !== null) {
    for (var i=0, len=result.snapshotLength; i < len; i++) {
        alert(result.snapshotItem(i).tagName);
    }
}
```

Здесь свойство `snapshotLength` возвращает количество узлов в снимке, а метод `snapshotItem()` — узел в конкретной позиции (подобно свойству `length` и методу `item()` объекта `NodeList`).

Результат из одного узла

Аргумент `XPathResult.FIRST_ORDERED_NODE_TYPE` позволяет запросить первый узел, соответствующий выражению, который доступен в результате через свойство `singleNodeValue`, например:

Листинг DomXPathExample03.htm


```
var result = xmldom.evaluate("employee/name", xmldom.documentElement, null,
                             XPathResult.FIRST_ORDERED_NODE_TYPE, null);

if (result !== null) {
    alert(result.singleNodeValue.tagName);
}
```

Как и в других запросах, при отсутствии узлов, соответствующих выражению, метод `evaluate()` возвращает `null`. Если узел возвращается, он доступен через свойство `singleNodeValue`.

Результаты простых типов

С помощью XPath-выражений можно получать простые типы данных (не узлы), используя логический, числовой и строковый типы `XPathResult`. Если запросить результат одного из этих типов, будет возвращено одно значение в свойстве `booleanValue`, `numberValue` или `stringValue`. В случае логического типа обработка XPath-выражения обычно дает `true`, если хотя бы один узел соответствует выражению, и `false` в противном случае, например:

Листинг DomXPathExample04.htm

```
var result = xmldom.evaluate("employee/name", xmldom.documentElement, null,
                             XPathResult.BOOLEAN_TYPE, null);
alert(result.booleanValue);
```

Если какие-либо узлы в этом примере соответствуют выражению `"employee/name"`, свойство `booleanValue` возвратит `true`.

Если запрашивается числовой тип, XPath-выражением должна быть XPath-функция, возвращающая число, например функция `count()`, которая подсчитывает все узлы, соответствующие указанному шаблону:

Листинг DomXPathExample05.htm

```
var result = xmldom.evaluate("count(employee/name)", xmldom.documentElement,
                             null, XPathResult.NUMBER_TYPE, null);
alert(result.numberValue);
```

Этот код выводит количество узлов, которые соответствуют выражению `"employee/name"`. Если попытаться использовать этот метод без какой-либо из специальных XPath-функций, свойство `numberValue` будет равно `NaN`.

При запросе результата строкового типа метод `evaluate()` находит первый узел, соответствующий XPath-выражению, и возвращает значение его первого дочернего узла, если он является текстовым, или пустую строку в противном случае, например:

Листинг DomXPathExample06.htm


```
var result = xmldom.evaluate("employee/name", xmldom.documentElement, null,
                             XPathResult.STRING_TYPE, null);
alert(result.stringValue);
```

Этот код выводит содержимое первого текстового узла, дочернего по отношению к первому элементу, который соответствует строке `"element/name"`.

Тип результата по умолчанию

Все XPath-выражения автоматически сопоставляются с тем или иным типом результата. Указание конкретного типа ограничивает возможные результаты, но вы можете автоматически получить предлагаемый по умолчанию тип, указав константу `XPathResult.ANY_TYPE`. В этом случае результатом обычно оказывается логическое значение, число, строка или итератор неупорядоченных узлов. Узнать тип возвращенного результата можно с помощью его свойства `resultType`, например:

```
var result = xmldom.evaluate("employee/name", xmldom.documentElement, null,
                             XPathResult.ANY_TYPE, null);

if (result !== null) {
    switch(result.resultType) {
        case XPathResult.STRING_TYPE:
            // обработка результата строкового типа
            break;

        case XPathResult.NUMBER_TYPE:
            // обработка результата числового типа
            break;

        case XPathResult.BOOLEAN_TYPE:
            // обработка результата логического типа
            break;

        case XPathResult.UNORDERED_NODE_ITERATOR_TYPE:
            // обработка итератора неупорядоченных узлов
            break;

        default:
            // обработка результатов других возможных типов

    }
}
```

Константа `XPathResult.ANY_TYPE` позволяет использовать XPath более естественным образом, но может требовать дополнительной обработки результата.

Поддержка пространств имен

Если в XML-документе используются пространства имен, для правильной оценки XPath-выражения нужно сообщить объекту `XPathEvaluator` пространство имен. Есть несколько способов сделать это. Рассмотрим следующий XML-код:

```
<?xml version="1.0" ?>
<wrox:books xmlns:wrox="http://www.wrox.com/">
    <wrox:book>
        <wrox:title>Professional JavaScript for Web Developers</wrox:title>
        <wrox:author>Nicholas C. Zakas</wrox:author>
    </wrox:book>
    <wrox:book>
        <wrox:title>Professional Ajax</wrox:title>
        <wrox:author>Nicholas C. Zakas</wrox:author>
        <wrox:author>Jeremy McPeak</wrox:author>
        <wrox:author>Joe Fawcett</wrox:author>
    </wrox:book>
</wrox:books>
```

В этом XML-документе все элементы относятся к пространству имен `http://www.wrox.com/` с префиксом `wrox`. Чтобы использовать XPath-выражение с этим документом, нужно определить упоминаемые в нем пространства имен; в противном случае оценить выражение не удастся.

Первый способ обработать пространства имен — создать объект `XPathNSResolver` методом `createNSResolver()`, который принимает узел документа, содержащий определение пространства имен. В предыдущем примере это элемент документа `<wrox:books>` с атрибутом `xmlns`, который определяет пространство имен. Вы можете передать этот узел в метод `createNSResolver()` и использовать полученный результат в методе `evaluate()`:

Листинг DomXPathExample07.htm


```
var nsresolver = xmldom.createNSResolver(xmldom.documentElement);

var result = xmldom.evaluate("wrox:book/wrox:author",
                             xmldom.documentElement, nsresolver,
                             XPathResult.ORDERED_NODE_SNAPSHOT_TYPE, null);

alert(result.snapshotLength);
```

Передача объекта `nsresolver` в метод `evaluate()` гарантирует, что префикс `wrox` в XPath-выражении будет обработан правильно. Попытка использовать это же выражение без объекта `XPathNSResolver` приведет к ошибке.

Второй способ обработать пространства имен — определить функцию, которая принимает префикс пространства имен и возвращает соответствующий URI, например:

Листинг DomXPathExample08.htm

```
var nsresolver = function(prefix){
    switch(prefix){
```

```
        case "wrox": return "http://www.wrox.com/";
        // другие варианты
    }
};

var result = xmldom.evaluate("count(wrox:book/wrox:author)",
        xmldom.documentElement, nsresolver, XPathResult.NUMBER_TYPE, null);

alert(result.numberValue);
```

Функция разрешения пространств имен помогает, если вы не знаете, какой узел документа содержит определения пространств имен. Если вам известны префиксы и URI, вы можете определить функцию для возвращения этой информации и передать ее в качестве третьего аргумента в метод `evaluate()`.

XPath в Internet Explorer

В Internet Explorer поддержка XPath встроена в основанный на ActiveX объект `document` из XML DOM, но не в объект DOM, возвращаемый объектом `DOMParser`. Таким образом, для работы с XPath в Internet Explorer до версии 9 включительно необходима ActiveX-реализация. В ней для каждого узла определены дополнительные методы `selectSingleNode()` и `selectNodes()`. Метод `selectSingleNode()` принимает шаблон XPath и возвращает первый соответствующий узел, если он есть, или `null`, если таких узлов нет, например:

Листинг IEXPathExample01.htm


```
var element = xmldom.documentElement.selectSingleNode("employee/name");

if (element !== null){
    alert(element.xml);
}
```

Этот код возвращает первый узел, соответствующий строке `"employee/name"`. Узлом контекста является `xmldom.documentElement`, для которого вызван метод `selectSingleNode()`. Поскольку он может возвратить значение `null`, нужно всегда проверять результат, прежде чем использовать методы узла.

Метод `selectNodes()` также принимает шаблон `XPath` как аргумент, но возвращает коллекцию `NodeList` со всеми узлами, соответствующими шаблону (если узлы ему не соответствуют, возвращается пустая коллекция `NodeList` без элементов). Вот пример:

Листинг IEXPathExample02.htm

```
var elements = xmldom.documentElement.selectNodes("employee/name");
alert(elements.length);
```

В этом примере все элементы, соответствующие выражению `"employee/name"`, возвращаются как коллекция `NodeList`. Значение `null` не может быть возвращено,

поэтому вы можете сразу же использовать результат. Помните, что `NodeList` — это динамическая коллекция, которая обновляется при каждом доступе к ней.

Internet Explorer поддерживает только базовые возможности XPath. Получить результаты, отличные от узла или коллекции `NodeList`, невозможно.

Поддержка пространств имен в Internet Explorer

В Internet Explorer для работы с выражениями XPath, которые содержат пространства имен, нужно узнать, какие пространства имен используются, и создать строку следующего формата:

```
"xmlns:prefix1='uri1' xmlns:prefix2='uri2' xmlns:prefix3='uri3'"
```

Затем эту строку нужно передать в метод `setProperty()` — специальный метод объекта `document` XML DOM в Internet Explorer. Он принимает два аргумента: имя задаваемого свойства и его значение. В приведенном далее примере именем свойства является `"SelectionNamespaces"`, а значением — строка указанного формата. Этот код оценивает XML-документ, использованный в примере с пространствами имен DOM XPath:

Листинг IEXPathExample03.htm


```
xmldom.setProperty("SelectionNamespaces",
                   "xmlns:wrox='http://www.wrox.com/'");

var result = xmldom.documentElement.selectNodes("wrox:book/wrox:author");
alert(result.length);
```

Как и в примере с DOM XPath, отсутствие сведений о разрешении пространств имен приводит к ошибке во время оценки выражения.

Кроссбраузерная обработка XPath

Функционал XPath в Internet Explorer очень ограничен, поэтому кроссбраузерный XPath-код не должен использовать ничего сверх этого. По сути, это означает, что в других браузерах нужно заново создать функции `selectSingleNode()` и `selectNodes()` с помощью объектов DOM Level 3 XPath. Первой функцией, которую мы создадим, будет `selectSingleNode()`. Она принимает три аргумента: узел контекста, выражение XPath и необязательный объект `namespaces`, который должен быть литералом следующего формата:

```
{
    prefix1: "uri1",
    prefix2: "uri2",
    prefix3: "uri3"
}
```

Представление сведений о пространстве имен в таком формате позволяет легко преобразовать их в формат разрешения пространства имен, специфичный для браузера. Полный код функции `selectSingleNode()` таков:

Листинг CrossBrowserXPathExample01.htm

```
function selectSingleNode(context, expression, namespaces){
    var doc = (context.nodeType != 9 ? context.ownerDocument : context);

    if (typeof doc.evaluate != "undefined"){
        var nsresolver = null;
        if (namespaces instanceof Object){
            nsresolver = function(prefix){
                return namespaces[prefix];
            };
        }

        var result = doc.evaluate(expression, context, nsresolver,
                                  XPathResult.FIRST_ORDERED_NODE_TYPE, null);
        return (result !== null ? result.singleNodeValue : null);

    } else if (typeof context.selectSingleNode != "undefined"){

        // создание строки пространств имен
        if (namespaces instanceof Object){
            var ns = "";
            for (var prefix in namespaces){
                if (namespaces.hasOwnProperty(prefix)){
                    ns += "xmlns:" + prefix + "='" +
                            namespaces[prefix] + "' ";
                }
            }
            doc.setProperty("SelectionNamespaces", ns);
        }
        return context.selectSingleNode(expression);
    } else {
        throw new Error("No XPath engine found.");
    }
}
```

В первую очередь эта функция определяет XML-документ, для которого нужно оценить выражение. Поскольку узлом контекста может быть документ, необходимо проверить свойство `nodeType`. После этой проверки переменная `doc` содержит ссылку на XML-документ, что позволяет проверить, есть ли у него метод `evaluate()`, который указывает, что поддерживается спецификация DOM Level 3 XPath. Если она поддерживается, далее следует узнать, был ли передан в функцию объект `namespaces`. Это делается с помощью оператора `instanceof`, потому что `typeof` возвращает `"object"` и для объектов, и для значений `null`. Переменная `nsresolver` инициализируется значением `null` и затем перезаписывается функцией, если доступны сведения о пространстве имен. Эта функция является замыканием и использует объект `namespaces` для возвращения URI пространств имен. После этого мы вызываем метод `evaluate()`, и прежде чем возвратить значение всей функции, проверяем, возвратил ли он узел.

Ветвь функции для Internet Explorer проверяет, есть ли у узла `context` метод `selectSingleNode()`. Если да, то, как и в ветви для DOM, первым делом мы собираем информацию о пространствах имен. Если в функцию передан объект `namespaces`,

мы перебираем его свойства, создавая строку в подходящем формате. Обратите внимание на использование метода `hasOwnProperty()`, который гарантирует, что изменения свойства `Object.prototype` не будут отражены в этой функции. Затем функция вызывает встроенный метод `selectSingleNode()` и возвращает результат.

Если ни один из двух способов не поддерживается, генерируется ошибка с уведомлением о том, что доступных XPath-модулей нет. Функцию `selectSingleNode()` можно использовать так:

Листинг CrossBrowserXPathExample01.htm

```
var result = selectSingleNode(xmldom.documentElement
               "wrox:book/wrox:author", { wrox: "http://www.wrox.com/" });
alert(serializeXml(result));
```

Кроссбраузерную функцию `selectNodes()` можно создать подобным образом. Она принимает те же три аргумента, что и функция `selectSingleNode()`, и имеет во многом схожую логику. Для наглядности различия функций выделены:

Листинг CrossBrowserXPathExample02.htm

```
function selectNodes(context, expression, namespaces){
    var doc = (context.nodeType != 9 ? context.ownerDocument : context);

    if (typeof doc.evaluate != "undefined"){
        var nsresolver = null;
        if (namespaces instanceof Object){
            nsresolver = function(prefix){
                return namespaces[prefix];
            };
        }

        var result = doc.evaluate(expression, context, nsresolver,
                        XPathResult.ORDERED_NODE_SNAPSHOT_TYPE, null);
        var nodes = new Array();

        if (result !== null){
            for (var i=0, len=result.snapshotLength; i < len; i++){
                nodes.push(result.snapshotItem(i));
            }
        }

        return nodes;
    } else if (typeof context.selectNodes != "undefined"){

        // создание строки пространств имен
        if (namespaces instanceof Object){
            var ns = "";
            for (var prefix in namespaces){
                if (namespaces.hasOwnProperty(prefix)){
                    ns += "xmlns:" + prefix + "='" +
                            namespaces[prefix] + "' ";
                }
            }
```

```
            doc.setProperty("SelectionNamespaces", ns);
        }
        var result = context.selectNodes(expression);
        var nodes = new Array();

        for (var i=0,len=result.length; i < len; i++){
            nodes.push(result[i]);
        }

        return nodes;
    } else {
        throw new Error("No XPath engine found.");
    }
}
```

Как видите, большая часть логики в этом коде заимствована у функции `selectSingleNode()`. В ветви DOM в качестве типа результата указан упорядоченный снимок, который сохраняется в массиве. Для совместимости с Internet Explorer функция должна возвращать массив, даже если результатов нет, так что массив `nodes` возвращается всегда. В ветви кода для Internet Explorer мы вызываем метод `selectNodes()`, а результаты копируем в массив. Поскольку Internet Explorer возвращает коллекцию `NodeList`, лучше скопировать узлы в массив, чтобы функция возвращала один и тот же тип независимо от браузера. Использовать ее можно следующим образом:

Листинг CrossBrowserXPathExample02.htm


```
var result = selectNodes(xmldom.documentElement, "wrox:book/wrox:author",
                         { wrox: "http://www.wrox.com/" });
alert(result.length);
```

В кроссбраузерном JavaScript-коде для обработки XPath лучше использовать только эти два метода.

Поддержка XSLT в браузерах

XSLT — это вспомогательная технология для XML, которая использует XPath для преобразования одного документа из одного представления в другое. В отличие от XML и XPath, технология XSLT не имеет формального API и совсем не представлена в формальной DOM, из-за чего производители браузеров могли реализовать ее по своему усмотрению. Впервые поддержка XSLT в JavaScript появилась в Internet Explorer.

XSLT в Internet Explorer

Как и остальной XML-функционал в Internet Explorer, поддержку XSLT обеспечивают объекты ActiveX. Полная поддержка XSLT 1.0 доступна в JavaScript начиная

с MSXML 3.0 (в составе Internet Explorer 6). DOM-документы, созданные с помощью объекта `DOMParser` в Internet Explorer 9, не поддерживают XSLT.

Простые преобразования XSLT

Простейший способ преобразовать XML-документ с помощью таблицы XSLT-стилей — это загрузить их в DOM-документ и воспользоваться методом `transformNode()`, который есть у каждого узла. Он принимает документ, содержащий таблицу XSLT-стилей, и возвращает строку с результатом преобразования, например:

Листинг IEXsltExample01.htm


```
// загрузка XML и XSLT (код специфичен для IE)
xmldom.load("employees.xml");
xsltdom.load("employees.xslt");

// преобразование
var result = xmldom.transformNode(xsltdom);
```

Этот код загружает XML-код и таблицу стилей XSLT в документы DOM, а затем вызывает для узла XML-документа метод `transformNode()`, в который передает XSLT. Переменной `result` присваивается строка, которая получается в результате преобразования. В данном случае преобразование начинается на уровне узла документа, потому что именно для него вызывается метод `transformNode()`. XSLT-преобразование можно также инициировать в любом месте документа, вызвав метод `transformNode()` для узла, с которого нужно начать преобразование, например:

```
result = xmldom.documentElement.transformNode(xsltdom);
result = xmldom.documentElement.childNodes[1].transformNode(xsltdom);
result = xmldom.getElementsByTagName("name")[0].transformNode(xsltdom);
result = xmldom.documentElement.firstChild.lastChild.transformNode(xsltdom);
```

Если вызвать метод `transformNode()` для узла, отличного от элемента `document`, преобразование начинается с этого узла, однако таблице XSLT-стилей все равно доступен весь XML-документ.

Сложные преобразования XSLT

Метод `transformNode()` обеспечивает базовые возможности XSLT-преобразований, но есть и более сложные способы применения этого языка, основанные на использовании XSL-шаблона и XSL-процессора. Первым делом нужно загрузить таблицу стилей XSLT в XML-документ (безопасный в многопоточной среде). Это делается с помощью ActiveX-объекта `MSXML2.FreeThreadedDOMDocument`, который поддерживает те же интерфейсы, что и обычный DOM-документ в Internet Explorer. Нужно создать объект последней версии:

Листинг IEXsltExample02.htm

```
function createThreadSafeDocument(){
    if (typeof arguments.callee.activeXString != "string"){
        var versions = ["MSXML2.FreeThreadedDOMDocument.6.0",
                        "MSXML2.FreeThreadedDOMDocument.3.0",
                        "MSXML2.FreeThreadedDOMDocument"],
            i, len;

        for (i=0,len=versions.length; i < len; i++){
            try {
                new ActiveXObject(versions[i]);
                arguments.callee.activeXString = versions[i];
                break;
            } catch (ex){
                // ничего делать не требуется
            }
        }
    }

    return new ActiveXObject(arguments.callee.activeXString);
}
```

Если не считать отличия имен, документ XML DOM, безопасный в многопоточной среде, используется так же, как обычный:

```
var xsltdom = createThreadSafeDocument();
xsltdom.async = false;
xsltdom.load("employees.xslt");
```

После создания и загрузки DOM-документа его нужно назначить XSL-шаблону, который также является ActiveX-объектом. Шаблон нужен для создания XSL-процессора, который затем можно использовать для преобразования XML-документа. Следующая функция создает шаблон подходящей версии:

Листинг IEXsltExample02.htm


```
function createXSLTemplate(){
    if (typeof arguments.callee.activeXString != "string"){
        var versions = ["MSXML2.XSLTemplate.6.0",
                        "MSXML2.XSLTemplate.3.0",
                        "MSXML2.XSLTemplate"],
            i, len;

        for (i=0,len=versions.length; i < len; i++){
            try {
                new ActiveXObject(versions[i]);
                arguments.callee.activeXString = versions[i];
                break;
            } catch (ex){
                // ничего делать не требуется
            }
        }
    }

    return new ActiveXObject(arguments.callee.activeXString);
}
```

С помощью функции `createXSLTemplate()` можно создать объект шаблона самой последней версии:

Листинг IEXsltExample02.htm

```
var template = createXSLTemplate();
template.stylesheet = xsltdom;

var processor = template.createProcessor();
processor.input = xmldom;
processor.transform();

var result = processor.output;
```

После создания XSL-процессора нужно назначить его свойству `input` узел, который требуется преобразовать. Им может быть документ или любой узел в документе. Вызов `transform()` выполняет преобразования и сохраняет результат в свойстве `output` как строку. Этот код дублирует функционал метода `transformNode()`.

Объекты XSL-шаблона версий 3.0 и 6.0 существенно различаются. В версии 3.0 свойством input должен быть полный документ, а использование узла приводит к ошибке. В версии 6.0 этим свойством может быть любой узел в документе.

Использование XSL-процессора обеспечивает дополнительный контроль над преобразованием и поддержку расширенных возможностей XSLT. Например, таблицы XSLT-стилей могут принимать параметры, используемые как локальные переменные. Рассмотрим следующую таблицу стилей:

Листинг employees.xslt


```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

    <xsl:output method="html" />

    <xsl:param name="message" />

    <xsl:template match="/">
       <ul>
           <xsl:apply-templates select="*" />
       </ul>
       <p>Сообщение: <xsl:value-of select="$message" /></p>
    </xsl:template>

    <xsl:template match="employee">
        <li><xsl:value-of select="name" />,
          <em><xsl:value-of select="@title" /></em></li>
    </xsl:template>

</xsl:stylesheet>
```

Эта таблица стилей определяет параметр `message` и выводит его в результатах преобразования. Чтобы задать значение `message`, перед вызовом `transform()` следует вызвать метод `addParameter()`. Он принимает два аргумента: имя параметра, который нужно задать (значение атрибута `name` элемента `<xsl:param>`), и его значение (чаще всего строка, но может быть числом или логическим значением). Вот пример:

Листинг IEXsltExample03.htm


```
processor.input = xmldom.documentElement;
processor.addParameter("message", "Hello World!");
processor.transform();
```

Если задано значение параметра, оно будет отражено в выводе.

Другая интересная функция XSL-процессора — это возможность задать режим операции. В XSLT это можно сделать для шаблона с помощью атрибута `mode`. Когда определен режим, шаблон не запускается без узла `<xsl:apply-templates>` с соответствующим атрибутом `mode`, например:

Листинг employees3.xslt

```
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

    <xsl:output method="html" />

    <xsl:param name="message" />

    <xsl:template match="/">
       <ul>
           <xsl:apply-templates select="*" />
       </ul>
       <p>Сообщение: <xsl:value-of select="$message" /></p>
    </xsl:template>

    <xsl:template match="employee">
        <li><xsl:value-of select="name" />,
          <em><xsl:value-of select="@title" /></em></li>
    </xsl:template>

    <xsl:template match="employee" mode="title-first">
        <li><em><xsl:value-of select="@title" /></em>,
          <xsl:value-of select="name" /></li>
    </xsl:template>

</xsl:stylesheet>
```

Эта таблица стилей содержит шаблон, у которого атрибут `mode` имеет значение `"title-first"`. В этом шаблоне сначала выводится должность сотрудника, а затем его имя. Чтобы можно было использовать шаблон, для элемента `<xsl:apply-templates>` также должен быть задан режим `"title-first"`. По умолчанию эта таблица стилей выводит такие же данные, что и предыдущая: сначала имя сотрудника, а затем — его должность. Если же при использовании этой таблицы задать в JavaScript режим

"title-first", первой будет выведена должность сотрудника. Это можно сделать с помощью метода setStartMode():

Листинг IEXsltExample05.htm


```
processor.input = xmldom;
processor.addParameter("message", "Hello World!");
processor.setStartMode("title-first");
processor.transform();
```

Метод setStartMode() принимает режим, в который нужно перевести процессор. Как и метод addParameter(), его нужно вызвать раньше метода transform().

Если вы собираетесь использовать одну таблицу стилей для нескольких преобразований, можно сбрасывать процессор после каждого преобразования с помощью метода reset(). Он очищает свойства ввода и вывода, а также начальный режим и любые указанные параметры. Вызывается он так:

```
processor.reset();    // подготовка к следующему преобразованию
```

Благодаря тому что процессор компилирует таблицу XSLT-стилей, он выполняет повторные преобразования быстрее по сравнению с методом transformNode().

MSXML-библиотека поддерживает только XSLT 1.0. Работы над MSXML прекратились с тех пор, как компания Microsoft переключилась на .NET Framework. Ожидается, что в будущем в JavaScript станут доступны XML- и XSLT-объекты, соответствующие технологии .NET.

Тип XSLTProcessor

Разработчики Firefox реализовали поддержку XSLT в JavaScript, создав новый тип XSLTProcessor. Он позволяет преобразовывать XML-документы, используя XSLT подобно XSL-процессору в Internet Explorer. Со временем объект XSLTProcessor был скопирован в Chrome, Safari и Opera, что сделало его стандартом де-факто для XSLT-преобразований в JavaScript.

Как и в Internet Explorer, первым делом нужно загрузить два DOM-документа: один с XML и один с XSLT. После этого следует создать объект XSLTProcessor и назначить ему таблицу XSLT-стилей с помощью метода importStylesheet():

Листинг XsltProcessorExample01.htm

```
var processor = new XSLTProcessor()
processor.importStylesheet(xsltdom);
```

Наконец, нужно выполнить преобразование. Это можно сделать двумя способами. Если требуется возвратить полный DOM-документ, вызовите метод transformToDocument(). Также можно получить объект фрагмента документа, вызвав метод transformToFragment(). Вообще говоря, единственный разумный способ

использования метода `transformToFragment()` — это добавление его результатов в другой DOM-документ.

При вызове метода `transformToDocument()` просто передайте в него документ XML DOM, а возвращенный им результат используйте как совершенно другой DOM-документ, например:

Листинг XsltProcessorExample01.htm

```
var result = processor.transformToDocument(xmldom);
alert(serializeXml(result));
```

Метод `transformToFragment()` принимает два аргумента: документ XML DOM, который нужно преобразовать, и документ, которому должен принадлежать итоговый фрагмент. Это гарантирует, что новый фрагмент документа будет действителен в целевом документе. Например, вы можете создать фрагмент и добавить его к странице, передав в метод объект `document` в качестве второго параметра:

Листинг XsltProcessorExample02.htm

```
var fragment = processor.transformToFragment(xmldom, document);
var div = document.getElementById("divResult");
div.appendChild(fragment);
```

Здесь процессор создает фрагмент, принадлежащий объекту `document`. Это позволяет добавить фрагмент в имеющийся на странице элемент `<div>`.

Если у таблицы XSLT-стилей форматом вывода является `"xml"` или `"html"`, создание документа или фрагмента документа вполне обоснованно, но если форматом вывода является `"text"`, обычно требуется просто текстовый результат преобразования. К сожалению, метода, возвращающего непосредственно текст, не существует. Метод `transformToDocument()` при формате вывода `"text"` возвращает полный XML-документ, но его содержимое зависит от браузера. Например, Safari возвращает весь HTML-документ, а Opera и Firefox — документ с одним элементом, текстом которого является вывод.

Решением этой проблемы является вызов метода `transformToFragment()`, который возвращает фрагмент документа с единственным дочерним узлом, содержащим текст результата. Следовательно, текст можно получить следующим образом:

```
var fragment = processor.transformToFragment(xmldom, document);
var text = fragment.firstChild.nodeValue;
alert(text);
```

Этот код работает одинаково во всех поддерживающих его браузерах, возвращая только текстовый вывод преобразования.

Использование параметров

Объект `XSLTProcessor` позволяет также задать XSLT-параметры с помощью метода `setParameter()`, который принимает три аргумента: URI пространства имен,

локальное имя параметра и задаваемое значение. Обычно URI пространства имен имеет значение `null`, а локальным именем является имя параметра. Этот метод нужно вызывать до метода `transformToDocument()` или `transformToFragment()`, например:

Листинг XsltProcessorExample03.htm

```
var processor = new XSLTProcessor()
processor.importStylesheet(xsltdom);
processor.setParameter(null, "message", "Hello World!");
var result = processor.transformToDocument(xmldom);
```

Для работы с параметрами используются также методы `getParameter()` и `removeParameter()`, которые получают текущее значение параметра и удаляют параметр соответственно. Оба они принимают URI пространства имен (обычно `null`) и локальное имя параметра, например:

```
var processor = new XSLTProcessor()
processor.importStylesheet(xsltdom);
processor.setParameter(null, "message", "Hello World!");

alert(processor.getParameter(null, "message"));    // "Hello World!"
processor.removeParameter(null, "message");

var result = processor.transformToDocument(xmldom);
```

Эти методы применяются не часто и предоставляются в основном для удобства.

Сброс процессора

Каждый экземпляр `XSLTProcessor` можно использовать многократно для преобразований с разными таблицами XSLT-стилей. Метод `reset()` удаляет из процессора все параметры и таблицы стилей, позволяя снова вызвать метод `importStylesheet()` для загрузки другой таблицы XSLT-стилей, например:

```
var processor = new XSLTProcessor()
processor.importStylesheet(xsltdom);

// какие-то преобразования

processor.reset();
processor.importStylesheet(xsltdom2);

// другие преобразования
```

Повторное использование экземпляра `XSLTProcessor` экономит память, если в преобразованиях задействовано несколько таблиц стилей.

Кроссбраузерные XSLT-преобразования

XSLT-преобразования в Internet Explorer во многом отличаются от подхода, основанного на `XSLTProcessor`, так что полностью использовать доступные

возможности в универсальном коде нереалистично. Простейшим кроссбраузерным способом XSLT-преобразований является возвращение строкового результата. В Internet Explorer для этого нужно просто вызвать метод `transformNode()` для узла контекста, а в других браузерах требуется сериализовать результат операции `transformToDocument()`. Следующую функцию можно использовать в Internet Explorer, Firefox, Chrome, Safari и Opera:

Листинг CrossBrowserXsltExample01.htm


```
function transform(context, xslt){
    if (typeof XSLTProcessor != "undefined"){
        var processor = new XSLTProcessor();
        processor.importStylesheet(xslt);

        var result = processor.transformToDocument(context);
        return (new XMLSerializer()).serializeToString(result);

    } else if (typeof context.transformNode != "undefined") {
        return context.transformNode(xslt);
    } else {
        throw new Error("No XSLT processor available.");
    }
}
```

Функция `transform()` принимает два аргумента: узел контекста, для которого нужно выполнить преобразование, и XSLT-объект `document`. Сначала она проверяет, определен ли тип `XSLTProcessor`; если да, он используется для преобразования. Функция вызывает метод `transformToDocument()` и сериализует результат в строку, которую затем возвращает. Если у узла контекста есть метод `transformNode()`, используется он. Если доступного XSLT-процессора нет, то как и другие кроссбраузерные функции в этой главе, функция `transform()` генерирует ошибку. Она вызывается следующим образом:

```
var result = transform(xmldom, xsltdom);
```

Метод `transformNode()` в Internet Explorer позволяет не использовать для преобразования документ DOM, безопасный в многопоточной среде.

Имейте в виду, что из-за использования разных интерпретаторов XSLT результаты преобразований могут зависеть от браузера. Используя XSLT в JavaScript, никогда не полагайтесь на абсолютную точность результата преобразования.

Резюме

В JavaScript доступна развитая поддержка XML и связанных технологий. К сожалению, из-за отсутствия спецификаций она реализована в браузерах по-разному. Спецификация DOM Level 2 определила API для создания пустых XML-документов,

но не для синтаксического анализа или сериализации, поэтому производители браузеров начали изобретать собственные подходы. В Internet Explorer выбран следующий подход:

- Поддержка XML реализована с помощью тех же ActiveX-объектов, которые используются при разработке приложений для настольных компьютеров.
- MSXML-библиотека поставляется с Windows и доступна из JavaScript.
- Эта библиотека поддерживает базовые возможности синтаксического анализа и сериализации XML, а также сопутствующие технологии, такие как XPath и XSLT.

В то же время в Firefox для синтаксического анализа и сериализации XML-кода были добавлены два объекта:

- `DOMParser` — это простой объект, который преобразует строку XML в DOM-документ;
- Объект `XMLSerializer` выполняет противоположную операцию, сериализуя DOM-документ в XML-строку.

Благодаря простоте и популярности эти объекты были дублированы в Internet Explorer 9, Opera, Chrome и Safari и стали стандартами де-факто в веб-разработке.

В DOM Level 3 представлена спецификация XPath API, которая реализована в Firefox, Safari, Chrome и Opera. Этот API позволяет выполнять в JavaScript-коде любые XPath-запросы для DOM-документа и получать результат независимо от его типа данных. В Internet Explorer поддержка XPath реализована в виде методов `selectSingleNode()` и `selectNodes()`. Хотя эти методы не могут сравниться с DOM Level 3, они все же позволяют использовать базовые возможности XPath для поиска узла или множества узлов в DOM-документе.

У XSLT нет API с открытой спецификацией. В Firefox для выполнения преобразований с помощью JavaScript был создан тип `XSLTProcessor`, который затем был скопирован в Safari, Chrome и Opera. Разработчики Internet Explorer реализовали собственное решение, включающее простой метод `transformNode()` и более сложный подход с шаблоном и процессором.

XML хорошо поддерживается в Internet Explorer, Firefox, Chrome, Safari и Opera. Хотя его реализации в Internet Explorer и других браузерах сильно различаются, они достаточно схожи для написания кроссбраузерного кода.

19 ECMAScript для XML

- Дополнительные E4X-типы
- Использование E4X для работы с XML
- Изменения синтаксиса

В 2002 году группа компаний во главе с BEA Systems предложила встроить поддержку XML в ECMAScript. В июне 2004 года под кодом ECMA-357 была выпущена спецификация ECMAScript для XML (E4X), пересмотренная в декабре 2005 года. E4X — это не отдельный язык, а необязательное расширение языка ECMAScript, предоставляющее новый синтаксис для работы с XML и соответствующими объектами.

Разработчики браузеров не торопились поддерживать стандарт E4X, хотя браузеры Firefox 1.5 и более поздних версий поддерживают его почти полностью. Эта глава посвящена реализации E4X в Firefox.

Типы E4X

E4X расширяет ECMAScript следующими глобальными типами:

- `XML` — любая единственная часть XML-структуры;
- `XMLList` — коллекция XML-объектов;
- `Namespace` — соответствие между префиксом и URI пространства имен;
- `QName` — квалифицированное имя, состоящее из локального имени и URI пространства имен.

С помощью этих четырех E4X-типов можно представить все части XML-документа путем проецирования каждого из типов `XML` и `XMLList` на несколько DOM-типов.

Тип XML

E4X-тип `XML` наиболее важен, потому что он может представлять любую часть XML-структуры. Экземпляром `XML` может быть элемент, атрибут, комментарий, инструкция по обработке или текстовый узел. Тип `XML` наследуется от `Object`, то есть обладает всеми свойствами и методами, имеющимися по умолчанию у любых объектов. Создать объект типа `XML` можно несколькими способами. Первый — вызвать конструктор:

```
var x = new XML();
```

Эта строка создает пустой XML-объект, который можно заполнить данными. В конструктор можно передать XML-строку, например:

```
var x = new XML("<employee position=\"Software Engineer\"><name>Nicholas " +
                "Zakas</name></employee>");
```

Переданная в конструктор строка при этом преобразуется в иерархию XML-объектов. Кроме того, в конструктор можно передать DOM-документ или DOM-узел:

```
var x = new XML(xmldom);
```

Эти методы полезны, но наиболее мощным и интересным является непосредственное назначение XML-данных переменной с помощью *XML-литерала* (XML literal). XML-литерал — это просто XML-код, внедренный в JavaScript, например:

Листинг XMLTypeExample01.htm


```
var employee = <employee position="Software Engineer ">
                    <name>Nicholas Zakas</name>
               </employee>;
```

Этот код назначает XML-структуру данных непосредственно переменной `employee`, преобразуя при этом данные в XML-объект.

Реализация E4X в Firefox не поддерживает синтаксический анализ XML-прологов. Если переданный в XML-конструктор текст или XML-литерал содержит элемент <?xml version=»1.0»?>, возникает синтаксическая ошибка.

Метод `toXMLString()` возвращает XML-строку, которая представляет объект и его дочерние элементы. Метод `toString()` работает по-разному в зависимости от содержимого XML-объекта. Если оно простое (обычный текст), возвращается текст, иначе метод `toString()` работает так же, как и `toXMLString()`, например:

```
var data = <name>Nicholas Zakas</name>;
alert(data.toString());        // "Nicholas Zakas"
alert(data.toXMLString());     // "<name>Nicholas Zakas</name>"
```

Этих двух методов достаточно для решения большинства задач XML-сериализации.

Тип XMLList

Тип `XMLList` представляет упорядоченную коллекцию XML-объектов. Его эквивалентом в DOM является тип `NodeList`, хотя объекты типов `XML` и `XMLList` преднамеренно сделаны более схожими, чем `Node` и `NodeList`. Для явного создания XMLList-объекта можно использовать конструктор `XMLList`:

```
var list = new XMLList();
```

Как и в XML-конструктор, в него можно передать XML-строку, причем строка может не содержать ни одного элемента `document`:

Листинг XMLListTypeExample01.htm

```
var list = new XMLList("<item/><item/>");
```

Здесь переменной `list` назначается XMLList-объект, содержащий два XML-объекта — по одному на каждый элемент `<item/>`.

XMLList-объект также можно создать, объединив два или более XML-объектов с помощью оператора «плюс» (+), который перегружен в E4X для создания XMLList-объектов, например:

```
var list = <item/> + <item/>;
```

Этот код объединяет два XML-литерала в XMLList-объект `list`, используя оператор «плюс». Вместо него можно задействовать специальный синтаксис `<>` и `</>`:

```
var list = <><item/><item/></>;
```

Можно создавать и отдельные XMLList-объекты, но обычно их применяют при анализе более крупных XML-структур. Рассмотрим пример:

Листинг XMLListTypeExample02.htm


```
var employees = <employees>
    <employee position="Software Engineer">
        <name>Nicholas Zakas</name>
    </employee>
    <employee position="Salesperson">
        <name>Jim Smith</name>
    </employee>
</employees>;
```

В этом коде определена переменная `employees`, которой назначается XML-объект, представляющий элемент `<employees/>`. Так как дочерних элементов `<employee/>` два, при этом создается XMLList-объект, который сохраняется в свойстве `employees.employee`. После этого обращаться к элементам можно с помощью скобочной нотации и по позиции:

```
var firstEmployee = employees.employee[0];
var secondEmployee = employees.employee[1];
```

У каждого XMLList-объекта есть также метод `length()`, который возвращает количество элементов:

```
alert(employees.employee.length());        // 2
```

Обратите внимание, что `length()` — это метод, а не свойство. Это сделано преднамеренно, чтобы объект типа `XMLList` отличался от массивов и объектов типа `NodeList`.

Интересно, что в E4X грани между типами `XML` и `XMLList` умышленно размыты. На деле нет заметного различия между XML- и XMLList-объектами, если последний содержит один XML-объект. Чтобы эти различия были минимальны, в тип `XML` добавлены метод `length()` и свойство, доступное как `[0]` (его возвращает сам XML-объект).

Совместимость типов `XML` и `XMLList` значительно упрощает работу с E4X, потому что некоторые методы могут возвращать любой из них.

Методы `toString()` и `toXMLString()` XMLList-объекта возвращают одно и то же строковое значение, а именно объединенное сериализованное представление всех содержащихся в нем XML-объектов.

Тип Namespace

Пространства имен представляются в E4X объектами типа `Namespace`. Они обычно используются для проецирования префиксов на URI пространств имен, хотя префикс нужен не всегда. Создать Namespace-объект можно с помощью конструктора:

```
var ns = new Namespace();
```

Кроме того, можно инициализировать Namespace-объект, передав в конструктор только URI либо префикс и URI:

Листинг NamespaceTypeExample01.htm


```
// префикс пространства имен отсутствует
var ns = new Namespace("http://www.wrox.com/");

// пространство имен wrox
var wrox = new Namespace("wrox", "http://www.wrox.com/");
```

Данные объекта `Namespace` можно получить с помощью свойств `prefix` и `uri`:

Листинг NamespaceTypeExample01.htm

```
alert(ns.uri);            // "http://www.wrox.com/"
alert(ns.prefix);         // undefined
alert(wrox.uri);          // "http://www.wrox.com/"
alert(wrox.prefix);       // "wrox"
```

Если префикс не назначен объекту `Namespace`, его свойство `prefix` имеет значение `undefined`. Чтобы создать пространство имен, предлагаемое по умолчанию, следует назначить этому свойству пустую строку.

Если XML-литерал содержит пространство имен или в XML-конструктор передается XML-строка со сведениями о пространстве имен, автоматически создается объект типа `Namespace`. Получить ссылку на него можно, вызвав метод `namespace()` с префиксом в качестве аргумента, например:

Листинг NamespaceTypeExample02.htm


```
var xml = <wrox:root xmlns:wrox="http://www.wrox.com/">
            <wrox:message>Hello World!</wrox:message>
          </wrox:root>;

var wrox = xml.namespace("wrox");
alert(wrox.uri);
alert(wrox.prefix);
```

В этом примере фрагмент XML-кода со сведениями о пространстве имен создается из XML-литерала. Объект `Namespace` из пространства имен `wrox` можно получить, вызвав метод `namespace("wrox")`, после чего становятся доступны свойства `uri` и `prefix`. Если у XML-фрагмента есть пространство имен, предлагаемое по умолчанию, его можно получить, передав в метод `namespace()` пустую строку.

Метод `toString()` объекта `Namespace` всегда возвращает URI пространства имен.

Тип QName

Тип `QName` представляет квалифицированное имя XML-объекта, которое состоит из пространства имен и локального имени. Вы можете создать объект `QName` вручную, передав в его конструктор имя либо объект `Namespace` и имя:

Листинг QNameTypeExample01.htm

```
var wrox = new Namespace("wrox", "http://www.wrox.com/");
var wroxMessage = new QName(wrox, "message");    // "wrox:message"
```

После создания объекта у него доступны свойства `uri` и `localName`. Свойство `uri` возвращает URI пространства имен, указанный при создании объекта (или пустую строку, если пространство имен не указано), а свойство `localName` — локальное имя, извлеченное из квалифицированного имени, например:

Листинг QNameTypeExample01.htm

```
alert(wroxMessage.uri);            // "http://www.wrox.com/"
alert(wroxMessage.localName);      // "message"
```

Эти свойства доступны только для чтения, а попытки изменить их значения приводят к ошибке. Объект `QName` переопределяет метод `toString()` для возвращения строки формата `uri::localName`, такой как `"http://www.wrox.com/::message"` в предыдущем примере.

При синтаксическом анализе XML-структуры объекты `QName` автоматически создаются для объектов `XML`, представляющих элементы или атрибуты. С помощью метода `name()` XML-объекта можно получить ссылку на связанный с ним QName-объект, например:

Листинг QNameTypeExample02.htm


```
var xml = <wrox:root xmlns:wrox="http://www.wrox.com/">
            <wrox:message>Hello World!</wrox:message>
          </wrox:root>;

var wroxRoot = xml.name();
alert(wroxRoot.uri);              // "http://www.wrox.com/"
alert(wroxRoot.localName);        // "root"
```

QName-объект создается для каждого элемента и атрибута в XML-структуре, даже если не указаны сведения о пространстве имен.

Квалифицированное имя XML-объекта можно изменить методом `setName()`, передав ему новый объект типа `QName`:

```
xml.setName(new QName("newroot"));
```

Этот метод обычно используется при изменении имени тега элемента или имени атрибута, который входит в пространство имен. Если имя не является частью пространства имен, можно изменить локальное имя, просто вызвав метод `setLocalName()`:

```
xml.setLocalName("newtagname");
```

Общие принципы использования

Когда в XML объект, элементы, атрибуты и текст собраны в иерархию объектов, вы можете перемещаться по ней, указывая имена атрибутов и тегов в точечной нотации. Каждый дочерний элемент представляется как свойство его родителя, при этом именем свойства является локальное имя дочернего элемента. Если он содержит только текст, свойство возвращает этот текст, например:

```
var employee = <employee position="Software Engineer">
                    <name>Nicholas Zakas</name>
               </employee>;
alert(employee.name);    // "Nicholas Zakas"
```

Элемент `<name/>` в этом коде содержит только текст, для получения которого используется свойство `employee.name`. Поскольку при передаче свойства в функцию `alert()` для него неявно вызывается метод `toString()`, в оповещении выводится текст элемента `<name/>`. Это обеспечивает простой доступ к текстовым данным, содержащимся в XML-документе. Если несколько элементов имеют одно имя тега, возвращается объект `XMLList`. Рассмотрим пример:

```
var employees = <employees>
    <employee position="Software Engineer">
        <name>Nicholas Zakas</name>
    </employee>
    <employee position="Salesperson">
        <name>Jim Smith</name>
    </employee>
</employees>;

alert(employees.employee[0].name);    // "Nicholas Zakas"
alert(employees.employee[1].name);    // "Jim Smith"
```

Этот код обращается к каждому элементу `<employee/>` и выводит значение его элемента `<name/>`. Если вы не знаете локальное имя дочернего элемента или хотите получить все дочерние элементы независимо от их имен, можете использовать звездочку (`*`):

Листинг UsageExample01.htm


```
var allChildren = employees.*;    // возвращение всех дочерних элементов
                                  // независимо от локального имени
alert(employees.*[0].name);       // "Nicholas Zakas"
```

Как и другие свойства, в зависимости от XML-структуры звездочка может возвратить либо один объект `XML`, либо объект `XMLList`.

Метод `child()` работает точно так же, как и доступ к свойству. В него можно передать имя любого свойства или его индекс, и он возвратит то же значение:

```
var firstChild = employees.child(0);            // то же, что employees.*[0]
var employeeList = employees.child("employee"); // то же, что
                                                // employees.employee
var allChildren = employees.child("*");         // то же, что employees.*
```

Для дополнительного удобства предоставляется метод `children()`, который всегда возвращает все дочерние элементы:

```
var allChildren = employees.children();         //то же, что employees.*
```

Есть также элемент `elements()`, который похож на `child()`, но возвращает только XML-объекты, представляющие элементы:

```
var employeeList = employees.elements("employee"); // то же, что
                                                   // employees.employee
var allChildren = employees.elements("*");         // то же, что employees.*
```

Эти методы позволяют использовать для доступа к XML-коду более привычный синтаксис.

Дочерние элементы можно удалять с помощью оператора `delete`:

```
delete employees.employee[0];
alert(employees.employee.length());        // 1
```

Это одно из главных преимуществ доступа к дочерним узлам с помощью свойств.

Доступ к атрибутам

К атрибутам также можно обращаться, используя точечную нотацию, хотя синтаксис для работы с ними слегка расширен. Чтобы имя атрибута отличалось от имени тега дочернего элемента, перед ним указывается знак `@`. Этот синтаксис заимствован из языка XPath, где перед именами атрибутов тоже указывается знак `@`. Выглядит это немного странно, например:

Листинг AttributesExample01.htm


```
var employees = <employees>
    <employee position="Software Engineer ">
        <name>Nicholas Zakas</name>
    </employee>
    <employee position="Salesperson">
        <name>Jim Smith</name>
    </employee>
</employees>;

alert(employees.employee[0].@position);        // "Software Engineer"
```

Как и элементы, каждый атрибут представляется свойством, доступным с помощью этой сокращенной нотации. Свойство возвращает объект `XML`, представляющий атрибут, а его метод `toString()` всегда возвращает значение атрибута. Для получения имени атрибута можно использовать метод `name()` этого объекта.

Вы также можете задействовать для доступа к атрибуту метод `child()`, передав в него имя атрибута с префиксом `@`:

Листинг AttributesExample01.htm

```
alert(employees.employee[0].child("@position"));    // "Software Engineer"
```

Поскольку с методом `child()` можно использовать любое имя свойства XML-объекта, чтобы различать имена тегов и атрибутов, необходимо ставить знак `@`.

Получить доступ к атрибутам можно только с помощью метода `attribute()`, передав в него имя атрибута. В отличие от метода `child()`, префикс `@` перед именем атрибута в этом случае не нужен, например:

Листинг AttributesExample01.htm

```
alert(employees.employee[0].attribute("position"));  // "Software Engineer"
```

Эти три способа доступа к свойствам поддерживаются типами `XML` и `XMLList`. В первом случае возвращается объект `XML`, представляющий атрибут; во втором — `XMLList` с XML-объектами атрибутов для всех элементов в списке. Например, в предыдущем фрагменте выражение `employees.employee.@position` возвращает объект `XMLList` с двумя объектами: для атрибута `position` первого и второго элементов `<employee/>`.

Для получения всех атрибутов в объекте `XML` или `XMLList` можно использовать метод `attributes()`. Он возвращает экземпляр `XMLList` с объектами, представляющими все атрибуты. Это то же самое, что применить шаблон `@*`, например:

```
// обе строки возвращают все атрибуты
var atts1 = employees.employee[0].@*;
var atts2 = employees.employee[0].attributes();
```

Чтобы изменить значение атрибута в E4X, достаточно задать другое значение свойства:

```
employees.employee[0].@position = "Author";  // изменение атрибута position
```

Затем изменение отражается внутри кода, так что при сериализации объекта `XML` значение атрибута обновляется. Эту же методику можно использовать для добавления новых атрибутов, например:

```
// добавление атрибута experience
employees.employee[0].@experience = "8 years";

// добавление атрибута manager
employees.employee[0].@manager = "Jim Smith";
```

Поскольку атрибуты работают так же, как любые другие ECMAScript-свойства, их можно удалять с помощью оператора `delete`:

```
delete employees.employee[0].@position;    // удаление атрибута position
```

Доступ к свойствам, связанным с атрибутами, позволяет без труда взаимодействовать с базовой структурой XML-кода.

Другие типы узлов

E4X может представлять все части XML-документа, включая комментарии и инструкции по обработке. По умолчанию E4X не анализирует комментарии или инструкции по обработке, поэтому они не отражены в иерархии объектов. Чтобы анализатор распознавал их, нужно задать в XML-конструкторе два следующих свойства:

```
XML.ignoreComments = false;
XML.ignoreProcessingInstructions = false;
```

Если эти флаги заданы, E4X выполняет синтаксический анализ комментариев и инструкций по обработке с их преобразованием в XML-структуру.

Поскольку тип `XML` представляет узлы всех типов, необходимо их как-то различать. Метод `nodeKind()` указывает, какой тип узла представляет XML-объект, и возвращает значение `"text"`, `"element"`, `"comment"`, `"processing-instruction"` или `"attribute"`. Рассмотрим следующий XML-объект:

```
var employees = <employees>
    <?Dont forget the donuts?>
    <employee position="Software Engineer">
        <name>Nicholas Zakas</name>
    </employee>
    <!-- только что добавлено -->
    <employee position="Salesperson">
        <name>Jim Smith</name>
    </employee>
</employees>;
```

Для этого кода метод `nodeKind()` возвращает значения, указанные в следующей таблице, в зависимости от того, какой узел находится в области видимости.

Инструкция	Возвращаемое значение
employees.nodeKind()	"element"
employees.*[0].nodeKind()	"processing-instruction"
employees.employee[0].@position.nodeKind()	"attribute"
employees.employee[0].nodeKind()	"element"
employees.*[2].nodeKind()	"comment"
employees.employee[0].name.*[0].nodeKind()	"text"

Метод `nodeKind()` нельзя вызвать для объекта `XMLList`, содержащего более одного объекта типа `XML`; это приводит к ошибке.

Можно возвратить только узлы конкретного типа, используя один из перечисленных методов.

- `attributes()` — возвращает все атрибуты объекта `XML`.
- `comments()` — возвращает все дочерние комментарии объекта `XML`.
- `elements(`*`имяТега`*`)` — возвращает все дочерние элементы объекта `XML`. Вы можете отфильтровать результаты, указав имя тега элементов, которые нужно возвратить.
- `processingInstructions(`*`имя`*`)` — возвращает все дочерние инструкции по обработке объекта типа `XML`. Вы можете отфильтровать результаты, указав имя возвращаемых инструкций по обработке.
- `text()` — возвращает все дочерние текстовые узлы объекта `XML`.

Каждый из этих методов возвращает объект `XMLList`, содержащий соответствующие XML-объекты.

С помощью методов `hasSimpleContent()` и `hasComplexContent()` вы можете определить, содержит ли XML-объект только текст или более сложное содержимое. Первый возвращает `true`, если дочерними узлами являются только текстовые узлы, а второй — если есть дочерние узлы, не являющиеся текстовыми, например:

```
alert(employees.employee[0].hasComplexContent());        // true
alert(employees.employee[0].hasSimpleContent());         // false
alert(employees.employee[0].name.hasComplexContent());   // false
alert(employees.employee[0].name.hasSimpleContent());    // true
```

Эти методы в сочетании с другими помогают запрашивать релевантные данные в XML-структуре.

Запросы

Синтаксис E4X-запросов во многом похож на таковой в языке XPath. Простейшим запросом является получение значения элемента или атрибута. Объекты `XML`, представляющие разные части XML-структуры, не создаются до выполнения запроса. По сути, все свойства объекта `XML` или `XMLList` являются частями запроса. Это означает, что при ссылке на свойство, которое не представляет часть XML-структуры, все равно возвращается объект `XMLList`, просто в нем ничего нет. Например, если запустить следующий код для предыдущего XML-фрагмента, ничто возвращено не будет:

Листинг QueryingExample01.htm

```
var cats = employees.cat;
alert(cats.length());        // 0
```

Этот запрос ищет элементы `<cat/>` в узле `<employees/>`. Первая строка возвращает пустой объект `XMLList`. Это позволяет выполнять запросы, не беспокоясь об исключениях.

В предыдущем примере основная часть кода работала с непосредственными дочерними элементами узлов с помощью точечной нотации. Указав две точки, можно расширить запрос на все узлы-потомки:

```
var allDescendants = employees..*;  // получение всех потомков <employees/>
```

В этом примере возвращаются все потомки элемента `<employees/>`. В результаты включаются элементы, текст, комментарии и инструкции по обработке, при этом последние две категории включаются исключительно на основе флагов, конструктора `XML` (см. предыдущий раздел). Чтобы получить только элементы с определенным именем тега, замените звездочку фактическим именем тега, например:

```
// получение всех узлов <name/>, являющихся потомками <employees/>
var allNames = employees..name;
```

Такие же запросы можно выполнять методом `descendants()`. Без аргументов он возвращает все потомки узла (то же, что и `..*`), а передав имя в качестве аргумента, можно ограничить его результаты. Вот примеры обеих ситуаций:

```
var allDescendants = employees.descendants();  // все потомки
var allNames = employees.descendants("name");  // все потомки узла <name/>
```

Получить все атрибуты всех потомков можно следующими способами:

```
var allAttributes = employees..@*;    // получение всех атрибутов потомков
var allAttributes2 = employees.descendants("@*");    // то же
```

Как и при работе с потомками элементов, вы можете ограничить результаты, указав полное имя атрибута вместо звездочки, например:

```
//получение всех атрибутов position
var allAttributes = employees..@position;

var allAttributes2 = employees.descendants("@position");    // то же
```

Можно потребовать, чтобы при доступе к потомкам выполнялось некоторое условие. Например, чтобы возвратить все элементы `<employee/>`, у которых атрибут `position` имеет значение `"Salesperson"`, можно использовать следующий код:

```
var salespeople = employees.employee.(@position == "Salesperson");
```

Этот синтаксис можно также задействовать для изменения частей XML-структуры. Например, можно изменить атрибут `position` первого продавца на `"Senior Salesperson"`:

```
employees.employee.(@position == "Salesperson")[0].@position=
                                                    "Senior Salesperson";
```

Выражение в обычных скобках возвращает объект `XMLList` с результатами, так что квадратные скобки затем возвращают первый элемент, для которого задается свойство `@position`.

Вы можете вернуться по XML-структуре, используя метод `parent()`, который возвращает родительский объект `XML`. Если метод `parent()` вызывается для объекта `XMLList`, он возвращает общего родителя всех объектов в списке, например:

```
var employees2 = employees.employee.parent();
```

Здесь переменной `employees2` присваивается значение `employees`. Метод `parent()` наиболее полезен при работе с объектами `XML` неизвестного происхождения.

Конструирование и выполнение XML-кода

Есть несколько способов заполнения объекта `XML` данными. Как уже отмечалось, вы можете передать строку XML-кода в конструктор `XML` или использовать XML-литерал. XML-литералы можно сделать более полезными, внедрив в них JavaScript-переменные в фигурных скобках: `{}`. Переменную можно использовать в любом месте XML-литерала, например:

Листинг XMLConstructionExample01.htm


```
var tagName = "color";
var color = "red";
var xml = <{tagName}>{color}</{tagName}>;

alert(xml.toXMLString());    //"<color>red</color>
```

В этом коде имя тега и текстовое значение XML-литерала указаны с помощью переменных в фигурных скобках. Это позволяет легко составлять XML-структуры без конкатенации строк.

В E4X можно также легко создать всю XML-структуру, используя стандартный JavaScript-синтаксис. Как уже отмечалось, большинство операций являются запросами и не генерируют ошибку, даже если элементы или атрибуты не существуют. Если вы присваиваете значение несуществующему элементу или атрибуту, E4X сначала создает базовую структуру, а затем выполняет присваивание, например:

Листинг XMLConstructionExample02.htm

```
var employees = <employees/>;
employees.employee.name = "Nicholas Zakas ";
employees.employee.@position = "Software Engineer";
```

Базовый элемент здесь — `<employees/>`, с которого все начинается. Вторая строка создает элемент `<employee/>`, содержащий элемент `<name/>` с текстовым значением.

В третьей строке к элементу добавляется атрибут `position`, после чего получается такая структура:

```
<employees>
    <employee position="Software Engineer">
        <name>Nicholas Zakas</name>
    </employee>
</employees>
```

Затем с помощью оператора `+` можно добавить второй элемент `<employee/>`:

Листинг XMLConstructionExample02.htm


```
employees.employee += <employee position="Salesperson">
                          <name>Jim Smith </name>
                      </employee>;
```

Итоговая XML-структура такова:

```
<employees>
    <employee position="Software Engineer ">
        <name>Nicholas Zakas</name>
    </employee>
    <employee position="Salesperson">
        <name>Jim Smith</name>
    </employee>
</employees>
```

Кроме этого базового синтаксиса XML-конструирования доступны также следующие методы, похожие на DOM-методы:

- `appendChild(`*дочернийУзел*`)` — добавляет указанный дочерний узел в конец объекта `XMLList`, представляющего дочерние узлы элемента;
- `copy()` — возвращает дубликат объекта `XML`;
- `insertChildAfter(`*опорныйУзел*, *дочернийУзел*`)` — вставляет дочерний узел после опорного узла в объект `XMLList`, представляющий дочерние узлы элемента;
- `insertChildBefore(`*опорныйУзел*, *дочернийУзел*`)` — вставляет дочерний узел перед узлом `refNode` в объект `XMLList`, представляющий дочерние узлы элемента;
- `prependChild(`*дочернийУзел*`)` — вставляет указанный дочерний узел в начало объекта `XMLList`, представляющего дочерние узлы элемента;
- `replace(`*имяСвойства*, *значение*`)` — заменяет указанное свойство, которое может быть элементом или атрибутом, переданным значением;
- `setChildren(`*дочерниеУзлы*`)` — заменяет все текущие дочерние узлы элемента указанными дочерними узлами, которые могут быть представлены объектом `XML` или `XMLList`.

Эти методы невероятно полезны и просты в использовании. Некоторые из них продемонстрированы в следующем примере:

Листинг XMLConstructionExample03.htm


```
var employees = <employees>
    <employee position="Software Engineer">
        <name>Nicholas Zakas</name>
    </employee>
    <employee position="Salesperson">
        <name>Jim Smith</name>
    </employee>
</employees>;

employees.appendChild(<employee position="Vice President">
                        <name>Benjamin Anderson </name>
                      </employee>);

employees.prependChild(<employee position="User Interface Designer">
                        <name>Michael Johnson</name>
                      </employee>);

employees.insertChildBefore(employees.child(2),
                            <employee position="Human Resources Manager">
                                <name>Margaret Jones</name>
                            </employee>);

employees.setChildren(<employee position="President">
                          <name>Richard McMichael</name>
                      </employee> +
                      <employee position="Vice President ">
                          <name>Rebecca Smith</name>
                      </employee>);
```

Сначала этот код добавляет вице-президента (Vice President) по имени Benjamin Anderson в конец списка сотрудников. Затем в начало списка добавляется дизайнер интерфейсов Michael Johnson. После этого непосредственно перед сотрудником в позиции 2, которым является Jim Smith (потому что теперь перед ним находятся Michael Johnson и Nicholas Zakas), добавляется начальник отдела кадров (Human Resources Manager) Margaret Jones. Наконец, все дочерние узлы заменяются узлами президента (President) по имени Richard McMichael и нового вице-президента по имени Rebecca Smith. В результате получается следующий XML-код:

```
<employees>
    <employee position="President">
        <name>Richard McMichael </name>
    </employee>
    <employee position="Vice President ">
        <name>Rebecca Smith</name>
    </employee>
</employees>
```

Используя эти приемы и методы, можно выполнять в E4X любые операции в DOM-стиле.

Параметры синтаксического анализа и сериализации

Синтаксическим анализом и сериализацией данных с помощью E4X можно управлять, используя несколько параметров конструктора `XML`. С синтаксическим анализом XML-кода связаны три следующих параметра:

- `ignoreComments` — указывает, что синтаксический анализатор должен игнорировать комментарии в разметке (по умолчанию `true`);
- `ignoreProcessingInstructions` — указывает, что синтаксический анализатор должен игнорировать в разметке инструкции по обработке (по умолчанию `true`);
- `ignoreWhitespace` — указывает, что синтаксический анализатор должен игнорировать свободные места между элементами, а не создавать для них текстовые узлы (по умолчанию `true`).

Эти параметры влияют на синтаксический анализ XML-строк, передаваемых в конструктор `XML`, а также XML-литералов.

С сериализацией XML-данных связаны следующие два параметра:

- `prettyIndent` — указывает количество пробелов в отступе при сериализации XML-кода (по умолчанию 2);
- `prettyPrinting` — указывает, что XML-код следует вывести в формате, понятном людям, то есть каждый элемент должен находиться в отдельной строке, а дочерние элементы должны выводиться с отступом (по умолчанию `true`).

Эти параметры влияют на результаты вызова методов `toString()` и `toXMLString()`.

Все пять параметров хранятся в объекте `settings`, который можно получить, вызвав метод `settings()` конструктора `XML`, например:

Листинг ParsingAndSerializationExample01.htm


```
var settings = XML.settings();
alert(settings.ignoreWhitespace);      // true
alert(settings.ignoreComments);        // true
```

Можно назначить все пять параметров за раз, передав объект с ними в метод `setSettings()`. Это полезно, если нужно временно изменить параметры, например:

```
var settings = XML.settings();
XML.prettyIndent = 8;
XML.ignoreComments = false;

// какая-то обработка

XML.setSettings(settings);    // возвращение к прежним параметрам
```

Вы всегда можете получить объект с параметрами, предлагаемыми по умолчанию, вызвав метод `defaultSettings()`, что позволяет сбросить параметры в любой момент:

```
XML.setSettings(XML.defaultSettings());
```

Пространства имен

Работать с пространствами имен в E4X довольно легко. Как уже отмечалось, вы можете получить объект `Namespace` для конкретного префикса с помощью метода `namespace()`. Также можно задать пространство имен для конкретного элемента, вызвав метод `setNamespace()` с объектом `Namespace` в качестве аргумента, например:

```
var messages = <messages>
    <message>Hello World!</message>
</messages>;
messages.setNamespace(new Namespace("wrox", "http://www.wrox.com/"));
```

При вызове метода `setNamespace()` пространство имен применяется только к элементу, для которого вызван метод. Сериализация переменной `messages` из приведенного кода дает следующий результат:

```
<wrox:messages xmlns:wrox="http://www.wrox.com/">
    <message>Hello World!</message>
</wrox:messages>
```

Как видите, вызов `setNamespace()` добавил к элементу `<messages/>` префикс пространства имен `wrox`, но элемент `<message/>` остался неизменным.

Чтобы просто добавить объявление пространства имен без изменения элемента, вызовите метод `addNamespace()`, передав в него объект `Namespace`, например:

```
messages.addNamespace(new Namespace("wrox", "http://www.wrox.com/"));
```

При выполнении этого кода для объекта `messages` создается следующая XML-структура:

```
<messages xmlns:wrox="http://www.wrox.com/">
    <message>Hello World!</message>
</messages>
```

Вызвав метод `removeNamespace()` и передав в него объект `Namespace`, можно удалить объявление пространства имен с указанными префиксом и URI (передавать точный объект `Namespace`, представляющий пространство имен, не требуется). Рассмотрим пример:

```
messages.removeNamespace(new Namespace("wrox", "http://www.wrox.com/"));
```

Данный код удаляет пространство имен `wrox`. На квалифицированные имена с указанным префиксом это не влияет.

Два метода возвращают массив объектов `Namespace`, связанных с узлом. Первый называется `namespaceDeclarations()` и возвращает массив всех пространств имен, объявленных для конкретного узла. Второй, `inScopeNamespaces()`, возвращает массив

всех пространств имен, доступных в области видимости конкретного узла; это означает, что они были объявлены для самого узла или его предка, например:

```
var messages = <messages xmlns:wrox="http://www.wrox.com/">
    <message>Hello World!</message>
</messages>;

alert(messages.namespaceDeclarations());          // "http://www.wrox.com"
alert(messages.inScopeNamespaces());              // ",http://www.wrox.com"

alert(messages.message.namespaceDeclarations()); // ""
alert(messages.message.inScopeNamespaces());      // ",http://www.wrox.com"
```

Для элемента `<messages/>` этот код возвращает массив с одним пространством имен при вызове метода `namespaceDEclarations()` и массив с двумя пространствами имен при вызове `inScopeNamespaces()`. В области видимости узла находятся пространство имен, предлагаемое по умолчанию (представленное пустой строкой), и пространство имен `wrox`. Когда эти методы вызываются для элемента `<message/>`, метод `namespaceDeclarations()` возвращает пустой массив, а `inScopeNamespaces()` — тот же результат, что и в предыдущем случае.

Объект `Namespace` можно также использовать с двумя двоеточиями (`::`) для запроса у XML-структуры элементов из указанного пространства имен. Например, получить все элементы `<message/>` из пространства имен `wrox` можно следующим образом:

```
var messages = <messages xmlns:wrox="http://www.wrox.com/">
    <wrox:message>Hello World!</wrox:message>
</messages>;
var wroxNS = new Namespace("wrox", "http://www.wrox.com/");
var wroxMessages = messages.wroxNS::message;
```

Два двоеточия задают пространство имен с элементом, который нужно возвратить. Обратите внимание, что при этом используется имя JavaScript-переменной, а не префикс пространства имен.

Вы также можете задать пространство имен, предлагаемое по умолчанию для всех объектов `XML`, создаваемых в указанной области видимости. Для этого воспользуйтесь инструкцией `default xml namespace`, назначив ей объект `Namespace` или просто URI пространства имен, например:

```
default xml namespace = "http://www.wrox.com/";

    function doSomething(){

        // задание пространства имен по умолчанию только для этой функции
        default xml namespace = new Namespace("your",
                                              "http://www.yourdomain.com");

}
```

XML-пространство имен, предлагаемое по умолчанию, не задается для глобальной области видимости. Эта инструкция позволяет использовать со всеми

XML-данными в указанной области видимости какое-то конкретное пространство имен без ссылок на него.

Другие изменения

Для гибкой интеграции со стандартным ECMAScript в базовый язык E4X были внесены некоторые изменения. Одной из новинок стал цикл `for-each-in`. В отличие от цикла `for-in`, который перебирает свойства, возвращая их имена, цикл `for-each-in` перебирает свойства и возвращает их значения, например:

Листинг ForEachInExample01.htm


```
var employees = <employees>
                   <employee position="Software Engineer ">
                       <name>Nicholas Zakas</name>
                   </employee>
                   <employee position="Salesperson">
                       <name>Jim Smith</name>
                   </employee>
                </employees>;

for each (var child in employees){
    alert(child.toXMLString());
}
```

Цикл `for-each-in` в этом примере по очереди назначает переменной `child` все дочерние узлы элемента `<employees/>`, которыми могут быть комментарии, инструкции по обработке и (или) текстовые узлы. Атрибуты в теле цикла не возвращаются, если не используется объект `XMLList` с атрибутами, например:

```
for each (var attribute in employees.@*){    // перебор атрибутов
    alert(attribute);
}
```

Хотя цикл `for-each-in` определен в E4X, его также можно использовать с обычными массивами и объектами:

Листинг ForEachInExample01.htm

```
var colors = ["red","green","blue"];
for each(var color in colors){
    alert(color);
}
```

При обработке массива цикл `for-each-in` возвращает каждый элемент массива, а при обработке объекта, отличного от `XML`, — каждое его свойство.

В E4X также добавлена глобальная функция `isXMLName()`, которая принимает строку и возвращает `true`, если она является допустимым локальным именем элемента или атрибута. Эта функция предоставлена для удобства разработчиков, которые

используют неизвестные строковые данные для создания XML-структур данных, например:

```
alert(isXMLName("color"));            // true
alert(isXMLName("hello world"));      // false
```

Если вы не уверены в происхождении строки, которую следует использовать как локальное имя, лучше сначала вызвать метод `isXMLName()`, чтобы узнать, допустима ли строка и не вызовет ли она ошибку.

Последнее изменение стандартного ECMAScript связано с оператором `typeof`. Для объекта `XML` или `XMLList` он возвращает строку `"xml"`, но для других объектов — `"object"`:

```
var xml = new XML();
var list = new XMLList();
var object = {};

alert(typeof xml);          // "xml"
alert(typeof list);         // "xml"
alert(typeof object);       // "object"
```

В большинстве случаев различать объекты `XML` и `XMLList` не требуется. Поскольку оба типа считаются в E4X примитивами, оператор `instanceof` также их не различает.

Полная поддержка E4X

Из-за того что в E4X многое делается иначе, чем в стандартном JavaScript, в Firefox реализованы только те части E4X, которые лучше всего работают вместе с другим кодом. Чтобы включить все E4X-возможности, нужно присвоить атрибуту `type` тега `<script>` значение `"text/javascript;e4x=1"`, например:

```
<script type="text/javascript;e4x=1" src="e4x_file.js"></script>
```

Эта команда включает полную поддержку E4X, в том числе правильное преобразование встроенных комментариев и CData-разделов в E4X-литералы. Использование комментариев и (или) CData-разделов без полного доступа к E4X-возможностям приводит к синтаксическим ошибкам.

Резюме

ECMAScript для XML (E4X) — это расширение ECMAScript, определенное в спецификации ECMA-357. Оно разработано для того, чтобы обеспечить возможности для работы с данными XML, похожие на стандартный ECMAScript. Перечислим основные характеристики E4X:

- В отличие от DOM, все типы XML-узлов представлены одним типом.
- Объект `XML` инкапсулирует данные и формы поведения, необходимые всем узлам. Коллекция из нескольких узлов определена в спецификации как объект `XMLList`.
- Другие два типа, `Namespace` и `QName`, представляют пространства имен и квалифицированные имена соответственно.

Чтобы было проще запрашивать XML-структуру, E4X изменяет стандартный синтаксис ECMAScript:

- Две точки (`..`) указывают, что требуются все потомки, тогда как знак `@` предписывает возвратить один или несколько атрибутов.
- Звездочка (`*`) представляет символ подстановки, которому соответствуют все узлы указанного типа.
- Все эти запросы можно также выполнять с помощью соответствующих методов.

На конец 2011 года Firefox был единственным браузером, поддерживающим расширение E4X. Хотя никакие другие производители браузеров не давали обязательств реализовать его, оно получило определенную популярность в серверной среде благодаря платформе BEA Workshop for WebLogic и языку YQL от Yahoo!.

20 JSON

- Общие сведения о синтаксисе JSON
- Синтаксический анализ JSON
- Сериализация JSON

Когда-то XML был стандартом де-факто передачи структурированных данных через Интернет. Большинство веб-сервисов первого поколения были основаны на XML, что подчеркивало их предназначение — обеспечение взаимодействия серверов. Однако XML не лишен недостатков. Многим казалось, что он слишком многословен и избыточен. Было предложено несколько решений этих проблем, но развитие веб-технологий пошло по другому пути.

Дуглас Крокфорд (Douglas Crockford) впервые описал нотацию JavaScript-объектов (JavaScript Object Notation, JSON) в 2006 году в IETF RFC 4627, хотя она использовалась уже в 2001 году. JSON — это строгое подмножество JavaScript, использующее несколько его паттернов для представления структурированных данных. Крокфорд указал, что JSON лучше XML подходит для доступа к структурированным данным в JavaScript, потому что позволяет напрямую передавать их в метод `eval()` и не требует создания DOM.

Важно понимать, что JSON — это формат данных, а не язык программирования. JSON не входит в JavaScript, хотя имеет такой же синтаксис, и будучи форматом данных, используется не только в JavaScript. Синтаксические анализаторы и средства сериализации JSON доступны для многих языков программирования.

Синтаксис

JSON поддерживает значения трех типов:

- **Простые значения** — строки, числа, логические значения и значения `null` можно представлять в JSON, используя тот же синтаксис, что и в JavaScript. Специальное значение `undefined` не поддерживается.
- **Объекты** — первый сложный тип данных, служащий для хранения упорядоченных пар ключей и значений. Каждое значение может быть примитивным или сложным типом.
- **Массивы** — второй сложный тип данных, который представляет упорядоченный список значений, доступных по числовому индексу. Значениями массивов могут быть данные любого типа, в том числе простые значения, объекты и даже другие массивы.

В JSON нет переменных, функций или экземпляров объектов. Этот формат предназначен для работы со структурированными данными, и хотя его синтаксис заимствован из JavaScript, его не следует воспринимать как одну из базовых составляющих языка.

Простые значения

Простейший JSON-код — это примитивное значение, например:

```
5
```

Это просто число 5. А следующий JSON-код представляет строку:

```
"Hello world!"
```

Важное различие между JavaScript- и JSON-строками заключается в том, что последние нужно заключать в двойные кавычки (использование одинарных кавычек приводит к синтаксической ошибке).

Логические значения и значение `null` также допустимы в JSON, но на практике этот формат чаще всего используется для представления более сложных структур данных, в которых простые значения — только часть всей информации.

Объекты

Объекты представляются в JSON с помощью слегка измененной нотации литералов объектов, которые в JavaScript выглядят следующим образом:

```
var person = {
    name: "Nicholas",
    age: 29
};
```

Это стандартный способ создания литералов объектов, но в JSON имена свойств заключаются в кавычки. Следующий код эквивалентен предыдущему:

```
var object = {
    "name": "Nicholas",
    "age": 29
};
```

В JSON тот же объект представляется так:

```
{
    "name": "Nicholas",
    "age": 29
}
```

Можно заметить два отличия этого кода от примера JavaScript-кода. Во-первых, в нем нет объявления переменной (переменные в JSON отсутствуют). Во-вторых, в этом коде отсутствует заключительная точка с запятой (она не требуется, потому что это не JavaScript-инструкция). Чтобы JSON-код был синтаксически правильным, имена свойств должны быть заключены в кавычки. Свойства могут содержать любые простые и сложные значения, что позволяет создавать вложенные объекты, например:

```
{
    "name": "Nicholas",
    "age": 29,
    "school": {
        "name": "Merrimack College",
        "location": "North Andover, MA"
    }
}
```

В этом примере объект со сведениями об учебном заведении вложен в объект верхнего уровня. Хотя код содержит два свойства с именем `"name"`, это нормально, потому что они принадлежат разным объектам. Однако в отдельном объекте одноименных свойств быть не должно.

В отличие от JavaScript, имена свойств-объектов в JSON всегда нужно заключать в двойные кавычки. При написании JSON-кода вручную отсутствие двойных кавычек или использование одинарных кавычек — частая ошибка.

Массивы

Второй сложный тип данных в JSON — массив. Массивы представляются в JSON с использованием нотации литералов массивов из JavaScript. Например, рассмотрим JavaScript-массив:

```
var values = [25, "hi", true];
```

Этот же массив в JSON будет выглядеть так:

```
[25, "hi", true]
```

Обратите внимание еще раз на отсутствие переменной и точки с запятой. Массивы и объекты можно использовать вместе для представления более сложных коллекций данных, например:

```
[
    {
        "title": "Professional JavaScript",
        "authors": [
            "Nicholas C. Zakas"
        ],
        edition: 3,
        year: 2011
    },
    {
        "title": "Professional JavaScript",
        "authors": [
            "Nicholas C. Zakas"
        ],
        edition: 2,
        year: 2009
    },
    {
        "title": "Professional Ajax",
        "authors": [
            "Nicholas C. Zakas",
            "Jeremy McPeak",
            "Joe Fawcett"
        ],
        edition: 2,
        year: 2008
    },
    {
        "title": "Professional Ajax",
        "authors": [
            "Nicholas C. Zakas",
            "Jeremy McPeak",
            "Joe Fawcett"
        ],
        edition: 1,
        year: 2007
    },
    {
        "title": "Professional JavaScript",
        "authors": [
            "Nicholas C. Zakas"
        ],
        edition: 1,
        year: 2006
    }
]
```

Этот массив содержит несколько объектов, представляющих книги. Каждый объект включает несколько ключей, одним из которых является другой массив, `"authors"`. Объекты и массивы обычно находятся на верхних уровнях JSON-иерархии (хотя такого требования нет) и используются для создания самых разных структур данных.

Синтаксический анализ и сериализация

Рост популярности JSON объясняется не только привычным синтаксисом. Еще важнее то, что он позволяет преобразовать данные в объект, который можно использовать в JavaScript. Это резко контрастирует с XML-кодом, который преобразуется в DOM-документ, что затрудняет извлечение данных. Например, получить название третьей книги из предыдущего фрагмента можно следующим образом:

```
books[2].title
```

Здесь предполагается, что структура данных содержится в переменной `books`. Сравните это выражение с типичным способом просмотра структуры DOM:

```
doc.getElementsByTagName("book")[2].getAttribute("title")
```

Неудивительно, что JSON приобрел невероятную популярность среди JavaScript-разработчиков и стал стандартом де-факто в веб-сервисах.

Объект JSON

Возможности ранних синтаксических JSON-анализаторов практически ограничивались вызовом `eval()` из JavaScript. Поскольку JSON является подмножеством синтаксиса JavaScript, функция `eval()` может анализировать, интерпретировать и возвращать данные как объекты и массивы JavaScript. В ECMAScript 5 средства синтаксического анализа JSON формализованы в виде встроенного глобального объекта `JSON`. Он поддерживается в Internet Explorer 8+, Firefox 3.5+, Safari 4+, Chrome и Opera 10.5+, а JSON-прокладка (shim) для старых браузеров доступна по адресу https://github.com/douglascrockford/JSON-js. Из-за риска столкнуться с исполняемым кодом обрабатывать JSON-код в старых браузерах лишь с помощью функции `eval()` опасно. Использование JSON-прокладки — оптимальный вариант для браузеров без встроенных JSON-средств синтаксического анализа.

У объекта `JSON` есть два метода: `stringify()` и `parse()`. В простых сценариях они просто сериализуют объект JavaScript в строку JSON и преобразуют ее в значение JavaScript соответственно. Вот пример:

Листинг JSONStringifyExample01.htm

```
var book = {
            title: "Professional JavaScript",
```

```
        authors: [
            "Nicholas C. Zakas"
        ],
        edition: 3,
        year: 2011
    };

var jsonText = JSON.stringify(book);
```

Этот код сериализует JavaScript-объект в JSON-строку методом `JSON.stringify()` и сохраняет ее в переменной `jsonText`. По умолчанию метод `JSON.stringify()` возвращает JSON-строку без дополнительных пробелов или отступов, так что в переменной `jsonText` сохраняется следующее значение:

```
{"title":"Professional JavaScript","authors":["Nicholas C. Zakas"],
"edition":3,"year":2011}
```

При сериализации JavaScript-объекта функции и члены прототипа специально не включаются в результат. Кроме того, пропускаются любые свойства со значением `undefined`. Итоговая строка содержит только свойства экземпляра, относящиеся к одному из JSON-типов данных.

Метод `JSON.parse()` создает из JSON-строки соответствующее JavaScript-значение. Например, так можно создать объект, аналогичный объекту `book`:

```
var bookCopy = JSON.parse(jsonText);
```

Переменные `book` и `bookCopy` являются отдельными объектами, которые не связаны друг с другом, хотя и имеют общие свойства.

Если текст, переданный в метод `JSON.parse()`, не является допустимой JSON-строкой, генерируется ошибка.

Параметры сериализации

Кроме сериализуемого JavaScript-объекта метод `JSON.stringify()` принимает еще два аргумента, с помощью которых можно указать альтернативные способы сериализации объекта. Первым аргументом является фильтр, которым может быть массив или функция, а вторым — параметр отступа для итоговой JSON-строки. Используя их по отдельности или вместе, можно задействовать очень полезные механизмы управления JSON-сериализацией.

Фильтрация результатов

Если вторым аргументом метода `JSON.stringify()` является массив, метод сериализует только свойства объекта, указанные в массиве, например:

Листинг JSONStringifyExample01.htm


```
var book = {
               "title": "Professional JavaScript",
                "authors": [
                    "Nicholas C. Zakas"
                ],
                edition: 3,
                year: 2011
           };

var jsonText = JSON.stringify(book, ["title", "edition"]);
```

Здесь в метод `JSON.stringify()` в качестве второго аргумента передается массив со строками `"title"` и `"edition"`. Они соответствуют свойствам сериализуемого объекта, так что итоговая JSON-строка будет содержать только эти свойства:

```
{"title":"Professional JavaScript","edition":3}
```

Если вторым аргументом метода является функция, его поведение немного отличается. Эта функция должна принимать два аргумента: имя ключа свойства и значение свойства. Ключ указывает, что нужно сделать со свойством. Он всегда является строкой, но может быть пустой строкой, если значение не соответствует ключу.

Чтобы изменить способ сериализации объекта, возвратите из функции значение, которое нужно сериализовать при указанном ключе. Помните, что возвращение значения `undefined` приводит к тому, что свойство не включается в результат, например:

Листинг JSONStringifyExample02.htm


```
var book = {
               "title": "Professional JavaScript",
                "authors": [
                    "Nicholas C. Zakas"
                ],
                edition: 3,
                year: 2011
           };

var jsonText = JSON.stringify(book, function(key, value){
    switch(key){
        case "authors":
            return value.join(",")

        case "year":
            return 5000;

        case "edition":
            return undefined;

        default:
            return value;
    }
});
```

Эта функция выполняет фильтрацию на основе ключа. Для ключа `"authors"` значение преобразуется из массива в строку, для ключа `"year"` возвращается значение 5000, а ключ `"edition"` вообще опускается — для него возвращается значение `undefined`. Важно указать вариант, предлагаемый по умолчанию и просто возвращающий переданное в функцию значение, чтобы в результат были добавлены все остальные значения. При первом вызове этой функции аргумент `key` равен пустой строке, а `value` ссылается на объект `book`. Итоговая JSON-строка такова:

```
{"title":"Professional JavaScript","authors":"Nicholas C. Zakas",
"year":5000}
```

Не забывайте, что фильтры применяются ко всем объектам, которые содержатся в сериализуемом объекте, так что если в первом примере из этого раздела передать в метод массив с несколькими объектами, каждый объект в итоге будет содержать только свойства `"title"` и `"edition"`.

В Firefox 3.5–3.6 в методе `JSON.stringify()` есть дефект, который проявляется, если вторым аргументом является функция. Она может работать только как фильтр: возвращение значения `undefined` приводит к тому, что свойство пропускается, а при любом другом значении оно добавляется в результат. Этот дефект исправлен в Firefox 4.

Отступы строк

Третий аргумент метода `JSON.stringify()` определяет отступы и свободное пространство. Если им является число, оно представляет количество пробелов, используемых в качестве отступа на каждом уровне. Например, задать для каждого уровня отступ из четырех пробелов можно следующим образом:

Листинг JSONStringifyExample03.htm


```
var book = {
              "title": "Professional JavaScript",
               "authors": [
                   "Nicholas C. Zakas"
               ],
               edition: 3,
               year: 2011
          };

var jsonText = JSON.stringify(book, null, 4);
```

В результате переменной `jsonText` будет присвоена следующая строка:

```
{
    "title": "Professional JavaScript",
    "authors": [
        "Nicholas C. Zakas"
    ],
    "edition": 3,
    "year": 2011
}
```

Возможно, вы заметили, что метод `JSON.stringify()` вставляет в JSON-код символы перевода строки, чтобы упростить его чтение. Это выполняется для всех допустимых значений аргумента отступа (отступы без перевода строки едва ли полезны). Максимальное значение отступа — 10. Если указать большее число, оно автоматически уменьшается до 10.

Если в качестве отступа указана строка, а не число, она используется в JSON-строке как символ отступа вместо пробела. Это позволяет сделать символом отступа знак табуляции или что-нибудь совершенно произвольное, например два дефиса:

```
var jsonText = JSON.stringify(book, null, "--");
```

В этом случае значение `jsonText` будет таким:

```
{
--"title": "Professional JavaScript",
--"authors": [
----"Nicholas C. Zakas"
--],
--"edition": 3,
--"year": 2011
}
```

Длина строки отступа ограничена десятью символами. Если указать строку длиннее десяти символов, она обрезается.

Метод toJSON()

Иногда возможностей метода `JSON.stringify()` недостаточно для сериализации некоторых объектов. В этих случаях можно добавить к объекту метод `toJSON()`, чтобы объект возвращал правильное представление себя в формате JSON. Между прочим, у встроенного типа `Date` есть метод `toJSON()`, который автоматически преобразует объекты `Date` из JavaScript в строки дат стандарта ISO 8601 (по сути, это то же самое, что вызвать метод `toISOString()` объекта `Date`).

Метод `toJSON()` можно добавить к любому объекту, например:

Листинг JSONStringifyExample05.htm


```
var book = {
               "title": "Professional JavaScript",
                "authors": [
                    "Nicholas C. Zakas"
                ],
                edition: 3,
                year: 2011,
                 toJSON: function(){
                     return this.title;
                 }
           };

var jsonText = JSON.stringify(book);
```

Здесь метод `toJSON()`, определенный для объекта `book`, просто возвращает название книги. Подобно объекту `Date`, это значение сериализуется в простую строку, а не в объект. Из метода `toJSON()` можно возвратить любое сериализованное значение, и код будет работать правильно. Возвращение значения `undefined` приводит к установке значения в `null`, если объект вложен в другой объект, а если объект относится к верхнему уровню, просто возвращается `undefined`.

Метод `toJSON()` можно использовать вместе с функцией фильтрации, поэтому важно понимать, в каком порядке осуществляется сериализация. Когда объект передается в метод `JSON.stringify()`, выполняются следующие действия.

1. Для получения фактического значения вызывается метод `toJSON()`, если он доступен, в противном случае используется способ сериализации, предлагаемый по умолчанию.
2. Если в метод передан второй аргумент, применяется фильтр. В функцию фильтра передается значение, возвращенное на первом этапе.
3. Каждое значение, полученное на втором этапе, сериализуется надлежащим образом.
4. Если указан третий аргумент метода, выполняется форматирование.

Важно понимать этот порядок, когда требуется выбрать для сериализации метод `toJSON()`, функцию фильтра или комбинированный подход.

Параметры синтаксического анализа

Метод `JSON.parse()` тоже принимает дополнительный аргумент — функцию, которая вызывается для каждой пары ключа и значения. Ее называют *функцией восстановления* (reviver function), чтобы отличать от *функции замены* (replacer function), или фильтра, который передается в метод `JSON.stringify()`. Функция восстановления также принимает ключ и значение в качестве аргументов и должна возвращать значение.

Если функция восстановления возвращает значение `undefined`, ключ удаляется из результата; если она возвращает любое другое значение, оно вставляется в результат. Эту функцию часто используют для преобразования строк дат в объекты `Date`, например:

Листинг JSONParseExample02.htm


```
var book = {
              "title": "Professional JavaScript",
               "authors": [
                   "Nicholas C. Zakas"
               ],
               edition: 3,
               year: 2011,
               releaseDate: new Date(2011, 11, 1)
```

```
        };

var jsonText = JSON.stringify(book);

var bookCopy = JSON.parse(jsonText, function(key, value){
    if (key == "releaseDate"){
        return new Date(value);
    } else {
        return value;
    }
});

alert(bookCopy.releaseDate.getFullYear());
```

Этот код добавляет к объекту `book` свойство `releaseDate` типа `Date`. Далее объект сериализуется для получения допустимой JSON-строки, а затем преобразуется обратно в объект `bookCopy`. Функция восстановления ищет ключ `"releaseDate"` и в случае его обнаружения создает из его значения объект `Date`. Итоговое свойство `bookCopy.releaseDate` содержит объект `Date`, так что для него можно вызвать метод `getFullYear()`.

Резюме

JSON — это компактный формат, с помощью которого можно легко представлять сложные структуры данных. Используемое в нем подмножество синтаксиса JavaScript поддерживает объекты, массивы, строки, числа, логические значения и значения `null`. Хотя XML предоставляет те же возможности, JSON-код более лаконичен и лучше поддерживается в JavaScript.

В ECMAScript 5 определен встроенный объект `JSON`, который служит для сериализации объектов в формат JSON и для преобразования данных JSON в JavaScript-объекты с помощью методов `JSON.stringify()` и `JSON.parse()` соответственно. Используя параметры этих методов, можно фильтровать значения или иным образом менять преобразование.

Встроенный объект `JSON` поддерживается в Internet Explorer 8+, Firefox 3.5+, Safari 4+, Opera 10.5 и Chrome.

21 Ajax и Comet

- Использование объекта XMLHttpRequest
- Работа с событиями XMLHttpRequest
- Ограничения на использование Ajax между доменами

В 2005 году Джесс Джеймс Гарретт (Jesse James Garrett) опубликовал в Интернете статью «Ajax: A New Approach to Web Applications» («Ajax: новый подход к программированию веб-приложений») (www.adaptivepath.com/ideas/essays/archives/000385.php). В этой статье он описал технологию, которую назвал *Ajax* (Asynchronous JavaScript+XML — асинхронный JavaScript+XML). Суть предложения заключалась в том, чтобы для улучшения впечатлений пользователей запрашивать дополнительные данные у сервера без выгрузки веб-страницы. Гарретт объяснил, как эта технология может изменить традиционный подход «щелкни и жди».

Ключевым элементом Ajax является объект `XMLHttpRequest` (XHR), представленный корпорацией Microsoft, а затем продублированный другими производителями браузеров. До XHR для обмена данными в стиле Ajax приходилось прибегать к различным хитростям — как правило, для этого использовали скрытые или встроенные фреймы. Объект XHR предоставил оптимизированный интерфейс для отправки запросов серверу и обработки его ответов. Это сделало возможным асинхронное получение дополнительной информации от сервера, то есть для этого больше не нужно обновлять страницу. Вместо этого можно получить данные с помощью объекта XHR, а затем вставить их в страницу, используя DOM. Несмотря на то что в полном названии Ajax упоминается XML, обмен данными с помощью Ajax не зависит от формата; сутью технологии является не использование XML, а получение данных от сервера без обновления страницы.

Технология, которую Гарретт назвал Ajax, на самом деле уже была известна. Этот режим взаимодействия браузера и сервера, который называется *удаленным*

выполнением сценариев (remote scripting), использовался в том или ином виде с 1998 года. Изначально запрашивать сервер из JavaScript можно было с помощью промежуточного компонента, такого как Java-апплет или Flash-ролик. Объект XHR предоставил разработчикам доступ к естественным средствам обмена данными браузера, что позволило работать эффективнее.

Под названием Ajax этот механизм взаимодействия браузера с сервером стал очень популярным в конце 2005 — начале 2006 года. Благодаря возрождению интереса к JavaScript и веб-технологиям в целом для использования этих возможностей появились новые способы и шаблонные процедуры. Теперь с объектом XHR должен уметь работать любой веб-разработчик.

Объект XMLHttpRequest

Объект XHR был представлен в Internet Explorer 5 в виде AcitveX-объекта из MSXML-библиотеки. В браузере доступны три его версии: `MSXML2.XMLHttp`, `MSXML2.XMLHttp.3.0` и `MXSML2.XMLHttp.6.0`. Для использования объекта XHR с MSXML-библиотекой требуется функция, похожая на ту, которую мы задействовали для создания XML-документов в главе 18:

```
// функция для IE до версии 7
function createXHR(){
    if (typeof arguments.callee.activeXString != "string"){
        var versions = ["MSXML2.XMLHttp.6.0", "MSXML2.XMLHttp.3.0",
                        "MSXML2.XMLHttp"],
            i, len;

        for (i=0,len=versions.length; i < len; i++){
            try {
                new ActiveXObject(versions[i]);
                arguments.callee.activeXString = versions[i];
                break;
            } catch (ex){
                // никакой код не требуется
            }
        }
    }

    return new ActiveXObject(arguments.callee.activeXString);
}
```

Эта функция пытается создать последнюю версию объекта XHR из доступных в Internet Explorer.

Internet Explorer 7+, Firefox, Opera, Chrome и Safari поддерживают встроенный объект XHR, который можно создать с помощью конструктора `XMLHttpRequest`:

```
var xhr = new XMLHttpRequest();
```

Если нужна поддержка только Internet Explorer 7 и более поздних версий, вы можете отказаться от предыдущей функции в пользу встроенной реализации объекта XHR. Если же вам нужно поддерживать и ранние версии Internet Explorer, можно расширить функцию `createXHR()`, чтобы она проверяла, доступен ли встроенный объект XHR:

Листинг XHRExample01.htm


```
function createXHR(){
    if (typeof XMLHttpRequest != "undefined"){
        return new XMLHttpRequest();
    } else if (typeof ActiveXObject != "undefined"){
        if (typeof arguments.callee.activeXString != "string"){
            var versions = ["MSXML2.XMLHttp.6.0", "MSXML2.XMLHttp.3.0",
                            "MSXML2.XMLHttp"],
                i, len;

            for (i=0,len=versions.length; i < len; i++){
                try {
                    new ActiveXObject(versions[i]);
                    arguments.callee.activeXString = versions[i];
                    break;
                } catch (ex){
                    // никакой код не требуется
                }
            }
        }

        return new ActiveXObject(arguments.callee.activeXString);
    } else {
        throw new Error("No XHR object available.");
    }
}
```

Новый код в этой функции сначала проверяет, доступен ли встроенный объект XHR, и если да — возвращает его экземпляр. Если обнаружить встроенный объект не удается, проверяется доступность объекта ActiveX. Если оба объекта недоступны, генерируется ошибка. Вы можете создать объект XHR в любом браузере, используя следующий код:

```
var xhr = createXHR();
```

Поскольку реализация объекта XHR в каждом браузере совместима с его оригинальной версией из Internet Explorer, созданный объект XHR во всех браузерах можно использовать одинаково.

Использование объекта XHR

Чтобы приступить к работе с объектом XHR, нужно сначала вызвать метод `open()`, который принимает три аргумента: тип отправляемого запроса (`"get"`, `"post"` и т. п.),

URL-адрес запроса и логическое значение, указывающее, нужно ли отправить запрос асинхронно, например:

```
xhr.open("get", "example.php", false);
```

Эта строка создает синхронный запрос `GET` страницы `example.php`. Имейте в виду, что URL-адрес интерпретируется относительно страницы, с которой вызывается код, хотя можно указать и абсолютный путь. Также учтите, что вызов `open()` не отправляет запрос, а только готовит его к отправке.

Доступны URL-адреса только из того же источника, то есть из того же домена с такими же номером порта и протоколом. Если какой-либо из этих элементов задан в URL-адресе иначе, чем на странице, с которой совершается запрос, возникнет ошибка безопасности.

Чтобы отправить указанный запрос, необходимо вызвать метод `send()`:

Листинг XHRExample01.htm

```
xhr.open("get", "example.txt", false);
xhr.send(null);
```

Метод `send()` принимает в качестве аргумента данные, которые нужно отправить как тело запроса. Если тело отправлять не нужно, вы должны передать в метод значение `null`, потому что в некоторых браузерах этот аргумент обязателен. Как только метод `send()` вызван, запрос направляется серверу.

Поскольку запрос выполняется синхронно, выполнение JavaScript-кода приостанавливается, пока не будет возвращен ответ. При получении ответа содержащиеся в нем данные назначаются свойствам объекта XHR. Нас интересуют следующие свойства:

- `responseText` — текст, возвращенный как тело ответа;
- `responseXML` — DOM-документ с данными ответа, если ответ содержит контент типа `"text/xml"` или `"application/xml"`;
- `status` — HTTP-состояние ответа;
- `statusText` — описание HTTP-состояния.

При получении ответа первым делом нужно проверить его свойство `status`, чтобы убедиться, что он был возвращен успешно. Как правило, успешно возвращенные ответы имеют 200-е коды HTTP-состояния, и если тип контента правилен, в этом случае какой-то контент будет содержаться в свойстве `responseText` и, возможно, в `responseXML`. Код состояния 304 указывает, что ресурс не был изменен и выдается из кэша браузера; это также означает, что ответ доступен. Чтобы убедиться в получении правильного ответа, следует проверить все эти состояния:

Листинг XHRExample01.htm


```
xhr.open("get", "example.txt", false);
xhr.send(null);

if ((xhr.status >= 200 && xhr.status < 300) || xhr.status == 304){
    alert(xhr.responseText);
} else {
    alert("Request was unsuccessful: " + xhr.status);
}
```

Этот код выводит на экран либо контент, возвращенный сервером, либо сообщение об ошибке, в зависимости от возвращенного кода состояния. Рекомендуется всегда проверять свойство `status` для определения оптимальных действий и не использовать для этого свойство `statusText`, потому что в некоторых браузерах оно работает ненадежно. Свойству `responseText` независимо от типа контента всегда назначается тело ответа, а свойство `responseXML` имеет значение `null`, если тип данных отличен от XML.

Несколько браузеров по ошибке возвращают код состояния 204. В Internet Explorer версии XHR, основанные на ActiveX, присваивают свойству status значение 1223 при получении ответа 204, а встроенные объекты XHR изменяют код 204 на 200. Opera при получении кода 204 присваивает свойству status значение 0.

Вместо синхронных запросов, наподобие предыдущего, обычно лучше использовать асинхронные, потому что они не требуют приостанавливать выполнение JavaScript-кода до получения ответа. У объекта XHR есть свойство `readyState`, которое указывает текущий этап цикла «запрос—ответ». Оно может иметь следующие значения:

- 0 — состояние не инициализировано. Метод `open()` еще не вызывался.
- 1 — запрос создан. Метод `open()` был вызван, а метод `send()` — нет.
- 2 — запрос отправлен. Метод `send()` был вызван, но ответ еще не получен.
- 3 — идет получение ответа. Некоторые данные ответа получены.
- 4 — запрос выполнен. Все данные ответа получены и доступны.

Когда изменяется значение свойства `readyState`, генерируется событие `readystatechange`, в обработчике которого можно проверить значение `readyState`. Вообще говоря, нас интересует только значение 4, которое указывает, что все данные готовы. Обработчик события `readystatechange` необходимо задать до вызова метода `open()`, чтобы код работал во всех браузерах. Рассмотрим пример:

Листинг XHRAsyncExample01.htm


```
var xhr = createXHR();
xhr.onreadystatechange = function(){
    if (xhr.readyState == 4){
        if ((xhr.status >= 200 && xhr.status < 300) || xhr.status == 304){
            alert(xhr.responseText);
```

```
        } else {
            alert("Request was unsuccessful: " + xhr.status);
        }
    }
};
xhr.open("get", "example.txt", true);
xhr.send(null);
```

Обратите внимание, что этот код подключает обработчик события к объекту `xhr` в стиле DOM Level 0, потому что не все браузеры поддерживают подход DOM Level 2. В отличие от других обработчиков событий, в обработчик `onreadystatechange` объект `event` не передается. Вместо этого для определения дальнейших действий нужно использовать сам объект XHR.

В этом примере в обработчике события readystatechange используется объект XHR, а не this из-за проблем с областью видимости обработчика. Использование this может вызвать в некоторых браузерах сбой в работе функции, так что безопаснее использовать переменную экземпляра XHR.

С помощью метода `abort()` можно отменить асинхронный запрос до получения ответа:

```
xhr.abort();
```

При вызове этого метода объект XHR прекращает генерировать события; кроме того, при этом блокируется доступ к любым его свойствам, связанным с ответом. Как только запрос отменен, следует присвоить ссылке на объект XHR значение `null`. Из-за возможных проблем с памятью повторно использовать объект XHR не рекомендуется.

Заголовки HTTP

Вместе с каждым HTTP-запросом и HTTP-ответом отправляется заголовочная информация, которая может быть полезна разработчику. Для доступа к заголовкам запроса и ответа используются методы объекта XHR.

По умолчанию при отправке XHR-запроса отправляются следующие заголовки:

- `Accept` — типы контента, которые браузер может обработать.
- `Accept-Charset` — кодировки, поддерживаемые браузером.
- `Accept-Encoding` — способы сжатия, поддерживаемые браузером.
- `Accept-Language` — языки, поддерживаемые браузером.
- `Connection` — тип подключения браузера к серверу.
- `Cookie` — cookie-файлы страницы.

- `Host` — домен страницы, которая инициирует запрос.
- `Referer` — URI страницы, инициирующей запрос. В спецификации HTTP имя этого заголовка содержит ошибку, которую приходится воспроизводить для обеспечения совместимости (правильное написание — «referrer»).
- `User-Agent` — строка пользовательского агента браузера.

Хотя набор отправляемых заголовков запроса зависит от браузера, некоторые заголовки отправляются в большинстве случаев. Дополнительные заголовки запроса можно задать с помощью метода `setRequestHeader()`, который принимает имя заголовка и его значение. Для отправки заголовков запроса нужно вызвать метод `setRequestHeader()` после метода `open()`, но до `send()`, например:

Листинг XHRRequestHeadersExample01.htm


```
var xhr = createXHR();
xhr.onreadystatechange = function(){
    if (xhr.readyState == 4){
        if ((xhr.status >= 200 && xhr.status < 300) || xhr.status == 304){
            alert(xhr.responseText);
        } else {
            alert("Request was unsuccessful: " + xhr.status);
        }
    }
};
xhr.open("get", "example.php", true);
xhr.setRequestHeader("MyHeader", "MyValue");
xhr.send(null);
```

С помощью пользовательских заголовков запроса можно указывать серверу правильный план действий. Рекомендуется всегда указывать собственные имена заголовков вместо тех, что обычно отправляет браузер, потому что заголовки, предлагаемые по умолчанию, могут влиять на ответ сервера. Одни браузеры позволяют перезаписывать заголовки, предлагаемые по умолчанию, а другие — нет.

Получить заголовок ответа из объекта XHR можно методом `getResponseHeader()`, передав в него имя нужного заголовка. Также можно получить все заголовки как одну строку с помощью метода `getAllResponseHeaders()`. Вот пример с обоими методами:

```
var myHeader = xhr.getResponseHeader("MyHeader");
var allHeaders = xhr.getAllResponseHeaders();
```

Заголовки можно использовать для передачи дополнительных структурированных данных от сервера браузеру. Метод `getAllResponseHeaders()` обычно возвращает данные вроде следующих:

```
Date: Sun, 14 Nov 2004 18:04:03 GMT
Server: Apache/1.3.29 (Unix)
Vary: Accept
```

```
X-Powered-By: PHP/4.3.8
Connection: close
Content-Type: text/html; charset=iso-8859-1
```

Получив эти данные, можно выполнить их синтаксический анализ и найти все имена отправленных заголовков вместо того, чтобы проверять наличие каждого заголовка по отдельности.

Запросы GET

Из разных типов запросов наиболее востребованы запросы GET, которые обычно используются, если нужно запросить у сервера ту или иную информацию. При необходимости к URL-адресу в запросе можно добавить аргументы строки запроса для передачи информации серверу. При работе с XHR строка запроса обязательна и должна быть правильно закодирована в URL-адресе, который передается в метод `open()`.

Одной из наиболее частых ошибок при работе с запросами GET является неправильное форматирование строки запроса. Все имена и значения в строке запроса должны быть закодированы с помощью метода `encodeURIComponent()` перед их добавлением к URL-адресу, а все пары имен и значений должны быть разделены амперсандом, например:

```
xhr.open("get", "example.php?name1=value1&name2=value2", true);
```

Для добавления аргументов строки запроса к существующему URL-адресу можно использовать следующую вспомогательную функцию:

```
function addURLParam(url, name, value) {
    url += (url.indexOf("?") == -1 ? "?" : "&");
    url += encodeURIComponent(name) + "=" + encodeURIComponent(value);
    return url;
}
```

Функция `addURLParam()` принимает три аргумента: URL-адрес, к которому нужно добавить параметр, имя параметра и его значение. В первую очередь функция проверяет, содержит ли URL-адрес вопросительный знак (чтобы определить, есть ли другие параметры в URL-адресе). Если нет, функция присоединяет к адресу вопросительный знак, в противном случае добавляется амперсанд. Затем имя и значение кодируются и добавляются к концу URL-адреса. Наконец, функция возвращает обновленный URL-адрес.

Эту функцию можно использовать для составления URL-адреса запроса следующим образом:

```
var url = "example.php";

// добавление аргументов
```

```
url = addURLParam(url, "name", "Nicholas");
url = addURLParam(url, "book", "Professional JavaScript");

// инициирование запроса
xhr.open("get", url, false);
```

Функция `addURLParam()` гарантирует, что строка запроса будет правильно отформатирована для использования с объектом XHR.

Запросы POST

Вторым по популярности типом запросов является запрос POST. Он обычно используется, если нужно отправить серверу данные, которые должны быть сохранены. Предполагается, что тело каждого запроса POST содержит данные, тогда как запросы GET обычно их не содержат. Тело запроса POST может содержать большой объем данных любого формата. Инициировать запрос POST можно, указав `"post"` в качестве первого аргумента метода `open()`, например:

```
xhr.open("post", "example.php", true);
```

Далее нужно передать некоторые данные в метод `send()`. Поскольку объект XHR был разработан для работы с XML, вы можете передать в метод XML-документ DOM, который будет сериализован и отправлен как тело запроса. Кроме того, можно передать в метод любую строку, которую требуется отправить серверу.

По умолчанию запрос POST не воспринимается сервером как отправка веб-формы. Чтобы получить переданные данные, сервер должен прочитать необработанные данные запроса. Тем не менее вы можете имитировать отправку формы с помощью объекта XHR. Сначала для этого нужно присвоить заголовку `Content-Type` значение `application/x-www-form-urlencoded`, которое представляет тип контента, задаваемый при отправке формы. Затем нужно создать строку соответствующего формата. Как отмечено в главе 14, данные запроса POST отправляются в том же формате, что и строка запроса. Если форму, уже имеющуюся на странице, нужно сериализовать и отправить серверу с помощью объекта XHR, для создания соответствующей строки можно использовать функцию `serialize()`, упоминавшуюся в главе 14:

Листинг XHRPostExample01.htm


```
function submitData(){
    var xhr = createXHR();
    xhr.onreadystatechange = function(){
        if (xhr.readyState == 4){
            if ((xhr.status >= 200 && xhr.status < 300) ||
                    xhr.status == 304){
                alert(xhr.responseText);
            } else {
                alert("Request was unsuccessful: " + xhr.status);
            }
        }
```

```
    };

    xhr.open("post", "postexample.php", true);
    xhr.setRequestHeader("Content-Type",
                         "application/x-www-form-urlencoded");
    var form = document.getElementById("user-info");
    xhr.send(serialize(form));
}
```

Эта функция сериализует данные формы с идентификатором `"user-info"` и отправляет их серверу. После этого отправленные данные можно получить в PHP-файле `postexample.php` с помощью переменной `$_POST`. Рассмотрим пример:

Листинг postexample.php

```
<?php
    header("Content-Type: text/plain");
    echo <<<EOF
Имя: {$_POST['user-name']}
Эл. почта: {$_POST['user-email']}
EOF;
?>
```

Если бы мы не добавили заголовок `Content-Type`, данные не были бы доступны в суперглобальной переменной `$_POST` и для доступа к ним пришлось бы использовать переменную `$HTTP_RAW_POST_DATA`.

Выполнение запросов POST требует больше ресурсов в сравнении с запросами GET. В плане быстродействия запросы GET могут выполняться до двух раз быстрее, чем запросы POST, отправляющие тот же объем данных.

XMLHttpRequest LEVEL 2

Популярность объекта XHR в качестве стандарта подтолкнула консорциум W3C к созданию официальных спецификаций для регламентации его поведения. В спецификации XMLHttpRequest Level 1 были определены детали уже существующей реализации объекта XHR, а в XMLHttpRequest Level 2 он был расширен. Не во всех браузерах спецификация Level 2 реализована полностью, но в той или иной степени ее поддерживает каждый браузер.

Тип FormData

В современных веб-приложениях часто требуется сериализовать данные формы, и для этого в спецификации XMLHttpRequest Level 2 представлен тип `FormData`. Он позволяет с легкостью выполнять сериализацию существующих форм и создавать данные в том же формате, что и в форме, для их передачи с помощью объекта XHR. Следующий код создает объект `FormData` и заполняет его некоторыми данными:

```
var data = new FormData();
data.append("name", "Nicholas");
```

Метод `append()` принимает в качестве аргументов ключ и значение — по сути, имя поля формы и содержащееся в нем значение. Вы можете добавить к объекту любое количество таких пар. Кроме того, можно предварительно заполнить пары ключей и значений данными элемента формы, передав элемент в конструктор `FormData`:

```
var data = new FormData(document.forms[0]);
```

Как только у вас есть экземпляр `FormData`, его можно передать непосредственно в метод `send()` объекта XHR, например:

Листинг XHRFormDataExample01.htm


```
var xhr = createXHR();
xhr.onreadystatechange = function(){
    if (xhr.readyState == 4){
        if ((xhr.status >= 200 && xhr.status < 300) || xhr.status == 304){
            alert(xhr.responseText);
        } else {
            alert("Request was unsuccessful: " + xhr.status);
        }
    }
};

xhr.open("post", "postexample.php", true);
var form = document.getElementById("user-info");
xhr.send(new FormData(form));
```

Тип `FormData` удобен тем, что вам не нужно явно задавать заголовки запроса для объекта XHR. Объект XHR распознает тип переданных данных как экземпляр `FormData` и настраивает заголовки соответствующим образом.

Тип `FormData` поддерживается в Firefox 4+, Safari 5+, Chrome и WebKit для Android 3+.

Тайм-ауты

В Internet Explorer 8 объект XHR был расширен свойством `timeout`, которое указывает время ожидания ответа на запрос перед его отменой. Если свойство `timeout` задано, и ответ не получен в течение указанного количества миллисекунд, генерируется событие `timeout` и вызывается обработчик `ontimeout`. Позднее этот функционал был добавлен в спецификацию XMLHttpRequest Level 2. Рассмотрим пример:

Листинг XHRTimeoutExample01.htm


```
var xhr = createXHR();
xhr.onreadystatechange = function(){
    if (xhr.readyState == 4){
        try {
            if ((xhr.status >= 200 && xhr.status < 300) ||
```

```
                xhr.status == 304){
            alert(xhr.responseText);
        } else {
            alert("Request was unsuccessful: " + xhr.status);
        }
    } catch (ex){
        // предполагается, что исключения обрабатываются
        // в обработчике ontimeout
    }
}
};

xhr.open("get", "timeout.php", true);
xhr.timeout = 1000;        // тайм-аут - 1 секунда (только IE 8+)
xhr.ontimeout = function(){
    alert("Request did not return in a second.");
};
xhr.send(null);
```

Этот пример поясняет использование свойства `timeout`. Присвоение ему значения в 1000 миллисекунд означает, что если ответ на запрос не возвратится за 1 секунду или раньше, запрос отменяется, при этом вызывается обработчик события `timeout`. Чтобы можно было сравнить свойство `readyState` со значением 4, обработчик события `readystatechange` все же должен быть вызван, однако при попытке доступа к свойству `status` после тайм-аута возникнет ошибка. Для предотвращения сбоя следует инкапсулировать код, проверяющий свойство `status`, в инструкцию `try-catch`.

На день написания этой главы тайм-ауты поддерживались только в Internet Explorer 8+.

Метод overrideMimeType()

Метод `overrideMimeType()` был введен в Firefox для переопределения MIME-типа запроса XHR, позднее он был добавлен в спецификацию XMLHttpRequest Level 2. Поскольку от MIME-типа ответа зависит способ обработки ответа объектом XHR, возможность переопределить тип, возвращенный сервером, бывает весьма кстати.

Рассмотрим ситуацию, когда сервер отправляет MIME-тип `text/plain`, который на самом деле содержит XML-код. В этом случае свойство `responseXML` имеет значение `null`, хотя ответ содержит XML-код. Вызвав метод `overrideMimeType()`, вы можете гарантировать, что ответ будет обработан как XML-данные, а не как обычный текст:

```
var xhr = createXHR();
xhr.open("get", "text.php", true);
xhr.overrideMimeType("text/xml");
xhr.send(null);
```

Этот код указывает объекту XHR обработать ответ как XML-данные, а не как обычный текст. Чтобы правильно переопределить MIME-тип ответа, нужно вызвать метод `overrideMimeType()` до метода `send()`.

Метод `overrideMimeType()` поддерживается в Firefox, Safari 4+, Opera 10.5+ и Chrome.

События хода обмена данными

Спецификация Progress Events от W3C — это рабочий проект, определяющий события обмена данными между клиентом и сервером. Сначала эти события были явно ориентированы на XHR, но теперь входят и в другие похожие API. Событий обмена данными шесть:

- `loadstart` — генерируется при получении первого байта ответа;
- `progress` — многократно генерируется во время получения ответа;
- `error` — генерируется, если при попытке выполнить запрос происходит ошибка;
- `abort` — генерируется при завершении подключения с помощью метода `abort()`;
- `load` — генерируется, когда ответ полностью получен;
- `loadend` — генерируется при завершении обмена данными и после событий `error`, `abort` и `load`.

Каждый запрос начинается с события `loadstart`, за которым следует одно или несколько событий `progress`. После этого генерируется событие `error`, `abort` или `load`, и наконец, все завершается событием `loadend`.

Первые пять событий поддерживаются в Firefox 3.5+, Safari 4+, Chrome, Safari для iOS и WebKit для Android. Opera 11 и Internet Explorer 8+ поддерживают только событие `load`. Событие `loadend` в настоящее время не поддерживается никакими браузерами.

Большинство этих событий просты, но два из них имеют нюансы, заслуживающие отдельного внимания.

Событие load

Объект XHR был реализован в Firefox, чтобы упростить модель взаимодействия. В связи с этим было представлено событие `load` для замены события `readystatechange`. Событие `load` генерируется, когда ответ полностью получен, и устраняет необходимость проверки свойства `readyState`. Обработчик события `load` получает объект `event`, у которого свойство `target` содержит экземпляр объекта XHR со всеми свойствами и методами. Не во всех браузерах объект `event` этого события реализован правильно, из-за чего необходимо использовать саму переменную объекта XHR, например:

Листинг XHRProgressEventExample01.htm


```
var xhr = createXHR();
xhr.onload = function(){
    if ((xhr.status >= 200 && xhr.status < 300) || xhr.status == 304){
        alert(xhr.responseText);
    } else {
        alert("Request was unsuccessful: " + xhr.status);
    }
};
xhr.open("get", "altevents.php", true);
xhr.send(null);
```

При получении ответа от сервера событие `load` генерируется независимо от состояния. Это означает, что вы должны проверить свойство `status`, чтобы определить, доступны ли нужные данные. Событие `load` поддерживается в Firefox, Opera, Chrome и Safari.

Событие progress

Другой инновацией, реализованной в объекте XHR компанией Mozilla, является событие `progress`, которое периодически генерируется, когда браузер получает новые данные. В обработчик события `progress` передается объект `event`, который содержит объект XHR в свойстве `target` и имеет три дополнительных свойства:

- `lengthComputable` — логическое значение, которое указывает, доступна ли информация о ходе выполнения операции;
- `position` — количество уже полученных байтов;
- `totalSize` — общее количество ожидаемых байтов согласно заголовку ответа `Content-Length`.

Имея на руках эту информацию, можно вывести для пользователя индикатор обмена данными:

Листинг XHRProgressEventExample01.htm

```
var xhr = createXHR();
xhr.onload = function(event){
    if ((xhr.status >= 200 && xhr.status < 300) || xhr.status == 304){
        alert(xhr.responseText);
    } else {
        alert("Request was unsuccessful: " + xhr.status);
    }
};
xhr.onprogress = function(event){
    var divStatus = document.getElementById("status");
    if (event.lengthComputable){
        divStatus.innerHTML = "Received " + event.position + " of " +
            event.totalSize + " bytes";
    }
};
```

```
xhr.open("get", "altevents.php", true);
xhr.send(null);
```

Для правильного выполнения обработчика `onprogress` нужно назначить его до вызова метода `open()`. В предыдущем примере HTML-элемент заполняется информацией о состоянии запроса каждый раз, когда генерируется событие `progress`. Если ответ содержит заголовок `Content-Length`, вы также можете использовать эту информацию для вычисления доли полученных данных относительно полного ответа.

Обмен ресурсами с запросом происхождения

Одним из главных ограничений взаимодействия в стиле Ajax с помощью объекта XHR является политика безопасности, регламентирующая доступ к разным источникам. По умолчанию объектам XHR доступны ресурсы только в том домене, к которому относится содержащая их веб-страница. Этот механизм защиты предотвращает некоторые злонамеренные действия. Однако потребность в правомочном доступе к ресурсам из разных доменов была настолько острой, что разработчики начали предлагать соответствующие решения для браузеров.

Обмен ресурсами с запросом происхождения (Cross-Origin Resource Sharing, CORS) — это рабочий проект W3C, который определяет способ взаимодействия браузера и сервера при доступе к источникам из других доменов. В основе CORS лежит применение пользовательских HTTP-заголовков с той целью, чтобы браузер и сервер могли получить друг о друге достаточно информации и выяснить, должен ли запрос или ответ завершиться успехом или неудачей.

Простой запрос GET или POST без пользовательских заголовков и с телом формата `text/plain` отправляется с дополнительным заголовком `Origin`. Он содержит источник (протокол, имя домена и порт) страницы, инициирующей запрос, чтобы сервер мог легко определить, должен ли он возвратить ответ. Заголовок `Origin` может выглядеть так:

```
Origin: http://www.nczonline.net
```

Если сервер решает, что запрос допустим, он отправляет браузеру заголовок `Access-Control-Allow-Origin`, дублируя информацию об источнике или возвращая `"*"`, если ресурс общедоступен, например:

```
Access-Control-Allow-Origin: http://www.nczonline.net
```

Если этот заголовок отсутствует или источники в запросе и ответе не соответствуют друг другу, браузер блокирует запрос, в противном случае запрос обрабатывается. Ни запросы, ни ответы не включают файлы cookie.

CORS в Internet Explorer

Корпорация Microsoft представила в Internet Explorer 8 тип `XDomainRequest` (XDR). Объекты этого типа работают подобно XHR, но безопасны при взаимодействии с другими доменами. Объект XDR частично реализует спецификацию CORS и отличается от XHR следующими характеристиками:

- cookie-файлы не отправляются с запросами и не принимаются с ответами;
- задать заголовки запросов, отличные от `Content-Type`, невозможно;
- заголовки ответов недоступны;
- поддерживаются только запросы GET и POST.

Эти изменения снижают остроту проблем, связанных с *межсайтовой подделкой запросов* (Cross-Site Request Forgery, CSRF) и *межсайтовым скриптингом* (Cross-Site Scripting, XSS). Запрашиваемый ресурс может динамически решать, следует ли задать заголовок `Access-Control-Allow-Origin`, опираясь на данные, которые он считает релевантными, такие как пользовательский агент, источник ссылки и т. д. В составе запроса отправляется заголовок `Origin` со значением, указывающим домен источника запроса, что позволяет удаленному ресурсу явно распознать XDR-запрос.

Используется объект XDR почти так же, как объект XHR: вы создаете экземпляр `XDomainRequest` и вызываете метод `open()`, а затем — метод `send()`. В отличие от метода `open()` объекта XHR, одноименный метод объекта XDR принимает только два аргумента: тип запроса и URL-адрес.

Все XDR-запросы выполняются асинхронно, создать синхронный запрос невозможно. При возвращении запроса генерируется событие `load`, а ответ назначается свойству `responseText`, например:

Листинг XDomainRequestExample01.htm


```
var xdr = new XDomainRequest();
xdr.onload = function(){
    alert(xdr.responseText);
};
xdr.open("get", "http://www.somewhere-else.com/page/");
xdr.send(null);
```

При получении ответа вам доступен только его необработанный текст; определить код состояния ответа нельзя. Для всех допустимых ответов генерируется событие `load`, а при любых сбоях, в том числе при отсутствии заголовка `Access-Control-Allow-Origin` в ответе, — событие `error`. К сожалению, дополнительные сведения об ошибке не предоставляются, так что приходится довольствоваться знанием того, что запрос завершился неудачей. Чтобы распознать ошибку, следует задать обработчик события `error`, например:

Листинг XDomainRequestExample01.htm

```
var xdr = new XDomainRequest();
xdr.onload = function(){
    alert(xdr.responseText);
};
xdr.onerror = function(){
    alert("An error occurred.");
};
xdr.open("get", "http://www.somewhere-else.com/page/");
xdr.send(null);
```

XDR-запрос может завершиться неудачей по целому ряду причин, поэтому следует всегда обрабатывать событие error, иначе сбой может остаться незамеченным.

С помощью метода abort() можно отменить запрос до возвращения ответа:

```
xdr.abort();    // отмена запроса
```

Подобно объекту XHR, объект XDR поддерживает свойство timeout и его обработчик ontimeout, например:

```
var xdr = new XDomainRequest();
xdr.onload = function(){
    alert(xdr.responseText);
};
xdr.onerror = function(){
    alert("An error occurred.");
};
xdr.timeout = 1000;
xdr.ontimeout = function(){
    alert("Request took too long.");
};
xdr.open("get", "http://www.somewhere-else.com/page/");
xdr.send(null);
```

В этом примере на запрос выделена одна секунда, по истечении которой вызывается обработчик ontimeout.

Чтобы можно было выполнять запросы POST, объект XDR предоставляет свойство contentType, позволяющее указать формат отправляемых данных, например:

```
var xdr = new XDomainRequest();
xdr.onload = function(){
    alert(xdr.responseText);
};
xdr.onerror = function(){
    alert("An error occurred.");
};
xdr.open("post", "http://www.somewhere-else.com/page/");
xdr.contentType = "application/x-www-form-urlencoded";
xdr.send("name1=value1&name2=value2");
```

Это свойство обеспечивает единственный способ доступа к информации заголовка через объект XDR.

CORS в других браузерах

В Firefox 3.5+, Safari 4+, Chrome, Safari для iOS и WebKit для Android поддержка CORS реализована в виде объекта `XMLHttpRequest`. При попытке открыть ресурс из другого источника этот объект автоматически задействуется без дополнительного кода. Чтобы запросить ресурс из другого домена, следует использовать стандартный объект XHR и передать в метод `open()` абсолютный URL-адрес, например:

```
var xhr = createXHR();
xhr.onreadystatechange = function(){
    if (xhr.readyState == 4){
        if ((xhr.status >= 200 && xhr.status < 300) || xhr.status == 304){
            alert(xhr.responseText);
        } else {
            alert("Request was unsuccessful: " + xhr.status);
        }
    }
};
xhr.open("get", "http://www.somewhere-else.com/page/", true);
xhr.send(null);
```

В отличие от объекта XDR в Internet Explorer, кроссдоменный объект XHR обеспечивает доступ к свойствам `status` и `statusText` и поддерживает синхронные запросы. Однако у него есть некоторые дополнительные ограничения, служащие для обеспечения безопасности:

- пользовательские заголовки нельзя задать с помощью метода `setRequestHeader()`;
- cookie-файлы не отправляются и не принимаются;
- метод `getAllResponseHeaders()` всегда возвращает пустую строку.

Поскольку и у обычных, и у кроссдоменных запросов один интерфейс, лучше всегда использовать относительный URL-адрес для доступа к локальному ресурсу, и абсолютный — для доступа к удаленному. Это делает код однозначным и помогает предотвращать такие проблемы, как ограничение доступа к заголовкам и (или) cookie-файлам для локальных ресурсов.

Предварительные запросы

CORS позволяет применять пользовательские заголовки, методы, отличные от GET или POST, и разные типы контента в теле запроса. Для этого служит прозрачный механизм верификации сервера, который называется *предварительными запросами* (preflighted requests). Предварительный запрос сервера получается, когда вы пытаетесь выполнить запрос с одним из расширенных параметров. Такой запрос осуществляется по методу OPTIONS и содержит следующие заголовки:

- `Origin` — то же, что и в простых запросах;
- `Access-Control-Request-Method` — метод выполнения запроса;
- `Access-Control-Request-Headers` (необязательный заголовок) — список пользовательских заголовков, разделенных запятыми.

Вот пример запроса POST с пользовательским заголовком `NCZ`:

```
Origin: http://www.nczonline.net
Access-Control-Request-Method: POST
Access-Control-Request-Headers: NCZ
```

Получив такой запрос, сервер может определить, следует ли разрешить запросы этого типа, а затем сообщает о принятом решении браузеру, отправляя в ответе следующие заголовки:

- `Access-Control-Allow-Origin` — то же, что и в простых запросах;
- `Access-Control-Allow-Methods` — список разрешенных методов, разделенных запятыми;
- `Access-Control-Allow-Headers` — список разрешенных сервером заголовков, разделенных запятыми;
- `Access-Control-Max-Age` — время в секундах, в течение которого предварительный запрос должен находиться в кэше.

Вот пример:

```
Access-Control-Allow-Origin: http://www.nczonline.net
Access-Control-Allow-Methods: POST, GET
Access-Control-Allow-Headers: NCZ
Access-Control-Max-Age: 1728000
```

Когда предварительный запрос обработан, результат кэшируется на время, указанное в ответе; то есть накладные расходы, связанные с дополнительным HTTP-запросом, имеют место только при первом запросе данного типа.

Предварительные запросы поддерживаются в Firefox 3.5+, Safari 4+ и Chrome. Internet Explorer до версии 10 включительно их не поддерживает.

Запросы с учетными данными

По умолчанию запросы ресурсов из других доменов не предоставляют учетные данные (cookie-файлы, данные для проверки подлинности HTTP или клиентские SSL-сертификаты). Вы можете указать, что в запросе нужно отправить учетные данные, присвоив свойству `withCredentials` значение `true`. Если сервер поддерживает запросы с учетными данными, он возвратит ответ со следующим HTTP-заголовком:

```
Access-Control-Allow-Credentials: true
```

Если в ответе на запрос с учетными данными нет этого заголовка, браузер не передает ответ JavaScript-сценарию (при этом свойство `responseText` является пустой строкой, свойство `status` равно 0, кроме того, вызывается обработчик `onerror()`). Имейте в виду, что сервер также может отправить этот HTTP-заголовок в ответе на предварительный запрос, показывая, что источнику разрешено отправлять запросы с учетными данными.

Свойство `withCredentials` поддерживается в Firefox 3.5+, Safari 4+ и Chrome. Internet Explorer до версии 10 включительно его не поддерживает.

Кроссбраузерный CORS

Хотя браузеры различаются по уровню встроенной поддержки CORS, все они поддерживают как минимум простые запросы (не предварительные запросы и не запросы с учетными данными), так что имеет смысл создать кроссбраузерное решение. Чтобы определить, поддерживает ли объект XHR технологию CORS, проще всего проверить наличие свойства `withCredentials`. Затем можно дополнить это проверкой наличия объекта `XDomainRequest`, чтобы охватить все браузеры:

Листинг CrossBrowserCORSRequestExample01.htm


```
function createCORSRequest(method, url){
    var xhr = new XMLHttpRequest();
    if ("withCredentials" in xhr){
        xhr.open(method, url, true);
    } else if (typeof XDomainRequest != "undefined"){
        xhr = new XDomainRequest();
        xhr.open(method, url);
    } else {
        xhr = null;
    }
    return xhr;
}

var request = createCORSRequest("get",
                                "http://www.somewhere-else.com/page/");
if (request){
    request.onload = function(){
        // какие-то действия со свойством request.responseText
    };
    request.send();
}
```

Интерфейсы объектов `XMLHttpRequest` в Firefox, Safari и Chrome и объекта `XDomainRequest` в Internet Explorer достаточно похожи, чтобы этот паттерн нормально работал. Общие свойства и методы интерфейсов таковы:

- `abort()` — отменяет уже выполняемый запрос;
- `onerror` — используется вместо обработчика `onreadystatechange` для распознавания ошибок;

- `onload` — используется вместо обработчика `onreadystatechange` для распознавания успешно выполненных запросов;
- `responseText` — служит для получения содержимого ответа;
- `send()` — служит для отправки запроса.

Каждый из этих элементов доступен у объекта, возвращенного методом `createCORSRequest()`, и одинаково работает во всех браузерах.

Альтернативные методики кроссдоменного взаимодействия

До появления CORS реализовать кроссдоменное взаимодействие в стиле Ajax было сложнее, поэтому для выполнения некоторых кроссдоменных запросов без объекта XHR разработчики использовали возможности DOM. Несмотря на повсеместное распространение CORS, эти методики все еще популярны, потому что они не требуют внесения изменений на стороне сервера.

Проверка связи с помощью изображения

Одна из первых методик кроссдоменного взаимодействия была основана на использовании тега `<img>`. С любой страницы можно загружать изображения из других доменов, не беспокоясь об ограничениях, например это основной способ отслеживания показов рекламных объявлений. Как отмечено в главе 13, вы также можете динамически создавать изображения и использовать их обработчики `onload` и `onerror` для регистрации получения ответов.

Динамическое создание изображений часто используется для *проверки связи с помощью изображения* (image ping). Так называют простое кроссдоменное однонаправленное взаимодействие с сервером. Данные при этом отправляются с помощью аргументов строки запроса, а ответ может быть любым, хотя обычно им является пиксельное изображение или ответ 204. При использовании этого приема браузер не может получить никакие специфические данные, зато он может узнать о получении ответа по событиям `load` и `error`. Вот простой пример:

Листинг ImagePingExample01.htm


```
var img = new Image();
img.onload = img.onerror = function(){
    alert("Done!");
};
img.src = "http://www.example.com/test?name=Nicholas";
```

Этот код создает экземпляр `Image`, а затем задает одну функцию в качестве обработчика событий `load` и `error`. Это гарантирует, что независимо от ответа вы будете

уведомлены о завершении запроса. Запрос начинается при установке свойства `src`, при этом в данном примере с ним отправляется параметр `name`.

Проверка связи с помощью изображения обычно применяется для отслеживания щелчков пользователя на странице или для динамического показа рекламы. Два главных недостатка этого приема заключаются в том, что вы можете отправлять только запросы GET и не можете получить доступ к тексту ответа с сервера. По этим причинам данный прием лучше всего подходит для однонаправленного взаимодействия браузера с сервером.

JSONP

JSONP (JSON with padding — JSON с набивкой) называют специальный вариант формата JSON, который стал популярен в веб-сервисах. JSONP-код выглядит как JSON-код, за исключением того, что данные заключены в синтаксический аналог вызова функции, например:

```
callback({ "name": "Nicholas" });
```

Синтаксис JSONP состоит из двух частей: функции обратного вызова и данных. Функция обратного вызова выполняется для страницы при получении ответа. Обычно имя этой функции указывается как часть запроса. Данные — это просто JSON-данные, передаваемые функции. Типичный JSONP-запрос выглядит так:

```
http://freegeoip.net/json/?callback=handleResponse
```

Это URL-адрес JSONP-сервиса геолокации. Функцию обратного вызова в JSONP-сервисах часто указывают как аргумент строки запроса; в данном случае она называется `handleResponse()`.

JSONP используется с динамическими элементами `<script>` (подробности см. в главе 13), при этом атрибут `src` назначается кроссдоменному URL-адресу. Подобно элементу `<img>`, элемент `<script>` может загружать ресурсы из других доменов без ограничений. Благодаря тому, что JSONP является допустимым JavaScript-кодом, ответ JSONP извлекается на страницу и незамедлительно выполняется по завершении запроса. Вот пример:

Листинг JSONPExample01.htm


```
function handleResponse(response){
    alert("Ваш IP-адрес: " + response.ip + "; город: " +
          response.city + ", " + response.region_name);
}

var script = document.createElement("script");
script.src = "http://freegeoip.net/json/?callback=handleResponse";
document.body.insertBefore(script, document.body.firstChild);
```

Этот код выводит на экран ваш IP-адрес и сведения о расположении, полученные от сервиса геолокации.

Формат JSONP очень популярен среди веб-разработчиков благодаря простоте и удобству его использования. Он привлекательнее проверки связи с помощью изображения тем, что текст ответа доступен напрямую, что делает возможным двунаправленное взаимодействие между браузером и сервером. Однако JSONP имеет и недостатки.

Первый недостаток заключается в том, что вы извлекаете исполняемый код на свою страницу из другого документа. Если этот домен не является доверенным, он может легко заменить ответ вредоносным кодом, и у вас не будет иного выхода, кроме как удалить JSONP-вызов. При использовании веб-сервиса, которым управляете не вы, убедитесь, что его источник заслуживает доверия.

Вторым недостатком является то, что нет простого способа определить, что JSONP-запрос завершился неудачей. Хотя в HTML5 определен обработчик события `error` для элементов `<script>`, он еще не реализован в браузерах. Разработчики часто использовали таймеры, чтобы узнать, получен ли ответ за указанное время, но этот способ недостаточно хорош, потому что подключения различаются по скорости и пропускной способности.

Comet

Термин *Comet* придумал Алекс Расселл (Alex Russell) для описания более сложной методики Ajax, которую иногда называют *отправкой данных по инициативе сервера* (server push). В то время как в Ajax страница запрашивает данные с сервера, в Comet сервер отправляет данные странице. Этот подход позволяет добавлять информацию на страницу почти в реальном времени, что делает его идеальным для вывода спортивных результатов или биржевых сводок.

Есть два популярных подхода к реализации Comet: *длинные опросы* (long polling) и *потоковая передача данных* (streaming). Длинные опросы — это новая разновидность традиционных опросов (также называемых короткими), при которых браузер отправляет запросы серверу через регулярные интервалы времени, чтобы узнать, доступны ли какие-либо данные. На рис. 21.1 показана временная шкала короткого опроса.

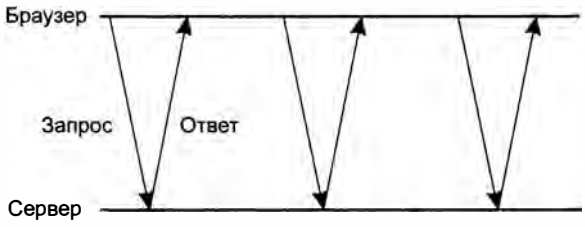

Рис. 21.1

Длинный опрос — это короткий опрос наоборот. Страница инициирует запрос сервера, а сервер поддерживает подключение активным, пока у него есть данные, которые требуется отправить. Когда данные отправлены, браузер закрывает подключение и немедленно инициирует новое подключение к серверу. Это продолжается, пока страница открыта в браузере. Временная шкала длинного опроса показана на рис. 21.2.

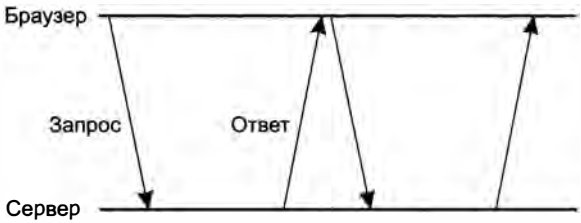

Рис. 21.2

При коротких и длинных опросах браузер должен инициировать подключение к серверу до получения данных. Важное их различие состоит в том, как сервер отправляет данные обратно. При коротком опросе ответ отправляется незамедлительно независимо от доступности данных, тогда как при длинном опросе ответ отправляется спустя период ожидания. Преимущество опросов в том, что их поддерживают все браузеры с помощью объекта XHR и метода `setTimeout()`. Вам нужно управлять только временем отправки запросов.

Вторым популярным подходом к Comet является потоковая передача данных по протоколу HTTP. Потоковая передача отличается от обеих методик опросов тем, что на весь срок существования страницы используется одно HTTP-подключение. Браузер отправляет запрос серверу, а сервер поддерживает подключение открытым, периодически отправляя данные. Например, на PHP-сервере может выполняться следующий сценарий:

```
<?php
    $i = 0;
    while(true){

        // вывод данных и очистка буфера
        echo "Number is $i";
        flush();

        // ожидание в течение нескольких секунд
        sleep(10);

        $i++;
    }
```

Все серверные языки поддерживают печать в буфер вывода и его очистку (когда содержимое буфера отправляется клиенту). Эти процессы лежат в основе потоковой передачи данных по протоколу HTTP.

Объект XHR можно использовать для потоковой передачи данных по протоколу HTTP в Firefox, Safari, Opera и Chrome, прослушивая событие `readystatechange` на предмет состояния 3 (`readyState` равно 3). Состояние готовности 3 периодически возникает во всех этих браузерах при получении данных с сервера. В этот момент свойство `responseText` содержит все полученные данные; это означает, что вам нужно обрезать новейший фрагмент, отследив, что было отправлено ранее. Реализация потоковой передачи данных по протоколу HTTP с помощью XHR выглядит следующим образом:

Листинг HTTPStreamingExample01.htm


```
function createStreamingClient(url, progress, finished){

    var xhr = new XMLHttpRequest(),
        received = 0;

    xhr.open("get", url, true);
    xhr.onreadystatechange = function(){
        var result;

        if (xhr.readyState == 3){

            // получение только новых данных и настройка счетчика
            result = xhr.responseText.substring(received);
            received += result.length;

            // вызов функции обратного вызова progress
            progress(result);

        } else if (xhr.readyState == 4){
            finished(xhr.responseText);
        }
    };
    xhr.send(null);
    return xhr;
}

var client = createStreamingClient("streaming.php", function(data){
                alert("Received: " + data);
            }, function(data){
                alert("Done!");
            });
```

Функция `createStreamingClient()` принимает три аргумента: URL-адрес для подключения, функцию, вызываемую при получении большего объема данных, и функцию, вызываемую при закрытии подключения. После закрытия подключения может потребоваться открыть его заново, так что имеет смысл следить за этим.

Когда генерируется событие `readystatechange` и свойство `readyState` имеет значение 3, свойство `responseText` обрезается, чтобы можно было возвратить только последние данные. Переменная `received` отслеживает количество уже обработанных символов, увеличиваясь каждый раз при обработке состояния готовности 3

(`readyState` равно 3). Затем выполняется функция обратного вызова `progress`, которой передаются новые данные. Когда свойство `readyState` равно 4, выполняется функция обратного вызова `finished`, которой передается весь контекст ответа.

Хотя этот пример сравнительно прост и работает в большинстве браузеров (за исключением Internet Explorer), управление подключениями в паттернах Comet легко реализовать неправильно, а его освоение требует времени. Разработчики браузеров считают, что Comet является перспективной веб-технологией, и создали два интерфейса, чтобы упростить работу с ней.

События, посылаемые сервером

События, посылаемые сервером (Server-Sent Events, SSE), — это API и паттерн для реализации взаимодействий Comet в режиме чтения. SSE создает однонаправленное HTTP-подключение к серверу, по которому сервер может передать любой объем информации. Ответ сервера должен иметь MIME-тип `text/eventstream` и выводит информацию специфического формата, которую API браузера принимает и делает доступной для JavaScript-кода. SSE поддерживает короткие опросы, длинные опросы и потоковую передачу данных по протоколу HTTP и таким образом автоматически определяет, когда нужно повторно подключиться к серверу в случае отключения от него. В результате получается очень простой и полезный API, который позволяет легко работать с Comet.

События, посылаемые сервером, поддерживаются в Firefox 6+, Safari 5+, Opera 11+, Chrome и Safari для iOS 4+.

API

JavaScript API для SSE похож на другие новые API передачи сообщений JavaScript. Чтобы подписаться на новый поток событий, нужно для начала создать объект `EventSource`, передав ему точку входа:

```
var source = new EventSource("myevents.php");
```

Имейте в виду, что указанный URL-адрес должен иметь тот же источник (схема, домен и порт), что и страница, на которой создается объект. У экземпляра `EventSource` есть свойство `readyState`, которое может иметь значение 0, если он подключается к серверу, 1, если подключение открыто, и 2, если подключение закрыто.

Также есть три события:

- `open` — генерируется при установлении подключения;
- `message` — генерируется при получении нового события от сервера;
- `error` — генерируется, если установить подключение невозможно.

В обычных ситуациях чаще всего используется обработчик события `message`:

```
source.onmessage = function(event){
    var data = event.data;
    // какие-то действия с данными
};
```

Информация, отправляемая сервером, возвращается как строковое значение `event.data`.

По умолчанию объект `EventSource` пытается поддерживать открытое подключение к серверу. Если подключение закрывается, предпринимается попытка восстановить его. Это означает, что SSE работает с длинными опросами и потоковой передачей данных по протоколу HTTP. С помощью метода `close()` можно выполнить принудительное отключение объекта и прекратить попытки восстановления подключения:

```
source.close();
```

Поток событий

События сервера отправляются при длительном HTTP-подключении вместе с ответом с MIME-типом `text/eventstream`. Ответ содержит обычный текст и в простейшей форме состоит из префикса `data:`, за которым следует текст, например:

```
data: foo

data: bar

data: foo
data: bar
```

Первая часть этого потока генерирует событие `message`, у которого свойство `event.data` имеет значение `"foo"`; вторая часть генерирует событие `message`, у которого свойство `event.data` равно `"bar"`; третья генерирует событие `message` со свойством `event.data`, равным `"foo\nbar"` (обратите внимание на знак перевода строки). При наличии двух или более последовательных строк, начинающихся на `data:`, они интерпретируются как многострочный фрагмент данных, а значения объединяются с помощью знака перевода строки. Событие `message` не генерируется, пока после строки с префиксом `data:` не встречается пустая строка, так что не забывайте включать дополнительный знак перевода строки при генерировании потока событий на сервере.

Вы также можете связать идентификатор с конкретным событием, добавив строку `id:` до или после строк с префиксом `data`:

```
data: foo
id: 1
```

После задания идентификатора объект `EventSource` отслеживает последнее сгенерированное событие. При разрыве подключения с запросом отправляется специальный

HTTP-заголовок `Last-Event-ID`, чтобы сервер мог определить, какое событие уместно сгенерировать следующим. Это важно для поддержания правильного порядка следования очередных фрагментов данных при нескольких подключениях.

Веб-сокеты

Веб-сокеты (Web Sockets) — один из самых обсуждаемых новых API для браузеров. Веб-сокеты обеспечивают полнодуплексное двунаправленное взаимодействие с сервером по одному длительному соединению. При создании веб-сокета в JavaScript-коде серверу отправляется HTTP-запрос для инициирования подключения. Когда сервер отвечает, подключение использует обновление HTTP для переключения с HTTP на протокол Web Socket. Это означает, что веб-сокеты невозможно реализовать на стандартном HTTP-сервере — для правильной работы требуется специализированный сервер, поддерживающий этот протокол.

Поскольку веб-сокеты используют собственный протокол, их схема URL немного отличается. Вместо схем `http://` и `https://` применяются схемы `ws://` для небезопасных подключений и `wss://` для безопасных. При указании URL-адреса веб-сокета необходимо задавать схему, так как в будущем может быть реализована поддержка других схем.

Преимущество использования специализированного протокола вместо HTTP состоит в том, что между клиентом и сервером можно передавать совсем небольшие объемы данных без служебных HTTP-данных. Применение меньших пакетов данных делает веб-сокеты идеальным решением для мобильных приложений, в которых особенно важны такие факторы, как пропускная способность и время задержки. Недостаток специализированного протокола состоит в том, что для его определения потребовалось больше времени в сравнении с JavaScript API. Принятие протокола Web Sockets многократно откладывалось из-за проблем с его согласованностью и безопасностью. В Firefox 4 и Opera 11 протокол Web Sockets был включен по умолчанию, но непосредственно перед выпуском браузеров его пришлось отключить из соображений безопасности. Теперь протокол Web Sockets поддерживается в Firefox 6+, Safari 5+, Chrome и Safari для iOS 4+.

API

Чтобы создать веб-сокет, создайте объект `WebSocket`, передав в конструктор URL-адрес подключения:

```
var socket = new WebSocket("ws://www.example.com/server.php");
```

Конструктор `WebSocket` принимает абсолютный URL-адрес. Политика одинакового источника не применяется к веб-сокетам, так что вы можете открыть подключение к любому сайту. Сервер сам решает, будет ли он взаимодействовать со страницей из конкретного источника (он может определить источник запроса по информации, полученной в ходе установления подключения).

Когда объект `WebSocket` создан, браузер пытается создать подключение. Подобно XHR, у объекта `WebSocket` есть свойство `readyState`, которое указывает текущее состояние. Однако его значения отличаются от значений у XHR:

- `WebSocket.OPENING` (0) — подключение устанавливается;
- `WebSocket.OPEN` (1) — подключение установлено;
- `WebSocket.CLOSING` (2) — начинается закрытие подключения;
- `WebSocket.CLOSE` (3) — подключение закрыто.

У объекта `WebSocket` нет события `readystatechange`, однако есть другие события, соответствующие различным состояниям. Первоначальное значение `readyState` всегда равно 0.

Подключение через веб-сокет можно закрыть в любой момент методом `close()`:

```
socket.close();
```

При вызове метода `close()` значение `readyState` немедленно изменяется на 2 (закрытие), а по завершении операции — на 3.

Отправка и получение данных

Когда веб-сокет открыт, вы можете отправлять и получать данные через соединение. Чтобы отправить данные серверу, вызовите метод `send()`, передав в него нужную строку, например:

```
var socket = new WebSocket("ws://www.example.com/server.php");
socket.send("Hello world!");
```

Поскольку веб-сокеты позволяют отправлять по соединению только обычный текст, более сложные структуры данных перед отправкой нужно сериализовать. Следующий код сериализует данные в JSON-строку, а затем отправляет ее серверу:

```
var message = {
    time: new Date(),
    text: "Hello world!",
    clientId: "asdfp8734rew"
};

socket.send(JSON.stringify(message));
```

Для доступа к данным сервер должен будет выполнить синтаксический анализ JSON-кода.

Когда сервер отправляет сообщение клиенту, для объекта `WebSocket` генерируется событие `message`. Оно работает так же, как и в других протоколах обмена сообщениями, при этом данные доступны через свойство `event.data`:

```
socket.onmessage = function(event){
    var data = event.data;

    // какие-то действия с данными
};
```

Как и данные, отправляемые серверу методом `send()`, значение свойства `event.data` всегда является строкой. Если вы ожидаете данные в другом формате, необходимо вручную выполнить их синтаксический анализ.

Другие события

У объекта `WebSocket` есть еще три события, которые генерируются во время существования подключения:

- `open` — генерируется при успешном подключении;
- `error` — генерируется при возникновении ошибки, при этом сохранить подключение не удается;
- `close` — генерируется при закрытии подключения.

Объект `WebSocket` не поддерживает слушатели событий DOM Level 2, так что обрабатывать эти события нужно в стиле DOM Level 0:

```
var socket = new WebSocket("ws://www.example.com/server.php");

socket.onopen = function(){
    alert("Connection established.");
};

socket.onerror = function(){
    alert("Connection error.");
};

socket.onclose = function(){
    alert("Connection closed.");
};
```

Из этих трех событий только у события `close` объект `event` содержит дополнительную информацию. У него есть три дополнительных свойства:

- `wasClean` — логическое значение, указывающее, правильно ли было закрыто подключение;
- `code` — числовой код состояния, отправленный сервером;
- `reason` — строковое сообщение, отправленное сервером.

Вы можете показать эту информацию пользователю или применить ее для анализа:

```
socket.onclose = function(event){
    console.log("Was clean? " + event.wasClean +
                " Code=" + event.code + " Reason=" + event.reason);
};
```

SSE или веб-сокеты?

При выборе SSE или веб-сокетов для конкретного сценария использования следует учесть несколько факторов. Прежде всего, можете ли вы установить сервер веб-сокетов? Поскольку протокол Web Socket отличается от HTTP, имеющиеся у вас веб-серверы могут не поддерживать обмен данными с помощью веб-сокетов. В то же время SSE работает поверх обычного протокола HTTP, так что, возможно, вам удастся использовать существующие веб-серверы.

Второй вопрос заключается в том, нужно ли вам двунаправленное взаимодействие. Если вам требуется только читать данные (например, результаты соревнований) с сервера, реализовать SSE может быть проще. Если требуется поддержка двунаправленного обмена данными (например, для чата), веб-сокеты могут оказаться предпочтительнее. Если веб-сокеты вам не подходят, двунаправленное взаимодействие можно также реализовать с помощью XHR и SSE.

Безопасность

На тему безопасности Ajax и Comet опубликовано много статей и даже целые книги. Обсуждать безопасность сложных Ajax-приложений можно очень долго, но есть некоторые базовые принципы безопасности Ajax, заслуживающие особого внимания.

Прежде всего, любой URL-адрес, доступный с помощью XHR, также доступен браузеру или серверу. Возьмем для примера следующий URL-адрес:

```
/getuserinfo.php?id=23
```

Предположим, что при запросе этого URL-адреса возвращаются некоторые данные о пользователе с идентификатором 23. Ничто не мешает кому-нибудь изменить идентификатор в URL-адресе на 24, 56 или любое другое значение. Файлу `getuserinfo.php` должно быть известно, действительно ли у инициатора запроса есть доступ к запрошенным данным, в противном случае сервер будет возвращать их кому угодно.

Такой несанкционированный доступ к ресурсу называется межсайтовой подделкой сценариев (CSRF), при этом неавторизованная система выдает себя серверу, который обрабатывает запрос, за уполномоченную. И крупные, и небольшие Ajax-приложения подвержены CSRF-атакам, проводимым с самыми разными целями: от проверки на уязвимость для совершенствования защиты до вредоносных атак, имеющих целью похищение или уничтожение данных.

Для защиты доступа к URL-адресам с помощью XHR обычно проверяют наличие прав на доступ у отправителя запроса. Это можно сделать следующим образом:

- потребовать использовать SSL для доступа к ресурсам, которые могут быть запрошены с помощью XHR;
- потребовать отправлять вычисляемый маркер с каждым запросом.

Имейте в виду, что следующие меры не защищают от CSRF-атак:

- требование использовать запрос POST вместо GET — это легко изменить;
- использование источника ссылки для определения происхождения запроса — его легко подделать;
- проверка прав с помощью cookie-файла — его также легко подделать.

Объект XHR обеспечивает защиту, которая на первый взгляд кажется надежной, но в реальности довольно неэффективна. У метода `open()` на самом деле есть два дополнительных аргумента: имя пользователя и пароль, отправляемые вместе с запросом. Их можно использовать, чтобы запрашивать страницы у сервера с помощью протокола SSL, например:

```
// НЕ ДЕЛАЙТЕ ТАК!!!
xhr.open("get", "example.php", true, "username", "password");
```

Передавать имя пользователя и пароль с помощью метода open() не следует. Хранить имена пользователей и пароли в JavaScript-коде небезопасно, потому что любой пользователь с JavaScript-отладчиком может просмотреть значения переменных и узнать имя пользователя и пароль, которые хранятся как обычный текст.

Резюме

Ajax — это технология получения данных с сервера без обновления текущей страницы. Перечислим ее ключевые характеристики:

- Главный объект, которому технология Ajax обязана своей популярностью, называется `XMLHttpRequest` (XHR).
- Этот объект был разработан корпорацией Microsoft и представлен в Internet Explorer 5 для получения XML-данных с сервера с помощью JavaScript.
- С тех пор объект XHR был продублирован в Firefox, Safari, Chrome и Opera, а консорциум W3C издал спецификацию его поведения, сделав XHR веб-стандартом.
- Несмотря на некоторые различия реализаций, основы работы с объектом XHR мало чем отличаются в разных браузерах, поэтому его можно безопасно использовать в веб-приложениях.

Одним из основных ограничений XHR является политика одинакового источника, которая требует использовать для взаимодействия один домен, один порт и один протокол. Любая попытка доступа к ресурсам в обход этого ограничения приводит к ошибке безопасности, если не используется одобренное кроссдоменное решение. Это решение называется обменом ресурсами с запросом происхождения (CORS); в Internet Explorer 8+ оно реализовано посредством объекта `XDomainRequest`, а в других браузерах — изначально посредством объекта XHR. Проверка связи с помощью

изображения и JSONP — две другие технологии обмена данными с разными доменами, но они менее надежны, чем CORS.

Comet — это расширение Ajax, позволяющее серверу инициировать выдачу данных клиенту почти в реальном времени. Comet предлагает два основных подхода: длинные опросы и потоковую передачу данных по протоколу HTTP. Первый подход поддерживается всеми браузерами, второй — только некоторыми. События, посылаемые сервером (SSE), — это браузерный API для взаимодействия в стиле Comet, поддерживающий и длинные опросы, и HTTP-потоки.

Веб-сокеты обеспечивают полнодуплексное двунаправленное взаимодействие с сервером. В отличие от других решений, веб-сокеты используют не HTTP, а специализированный протокол, оптимизированный для быстрой доставки небольших блоков данных. Это требует применения другого веб-сервера, но обеспечивает преимущество в скорости.

Шумиха вокруг Ajax и Comet привлекла многих разработчиков к JavaScript и вызвала очередной всплеск интереса к веб-разработке. Концепции, связанные с Ajax, все еще сравнительно новы и, несомненно, продолжат развиваться.

Тема Ajax важна и обширна, но ее полное обсуждение выходит за рамки темы данной книги. Дополнительные сведения об Ajax см. в книге Professional Ajax, 2nd Edition (Wiley, 2007; ISBN: 978-0-470-10949-6).

22 Более сложные приемы

- Расширенное применение функций
- Защита объектов от несанкционированного доступа
- Работа с таймерами

JavaScript — невероятно гибкий язык, который можно использовать по-разному. Обычно JavaScript-код пишут в процедурном или объектно-ориентированном стиле, но благодаря динамической природе языка на нем можно реализовывать гораздо более элегантные и интересные паттерны. Описанные в этой главе приемы позволяют добиваться впечатляющих результатов за счет использования возможностей языка ECMAScript, расширений модели BOM и функциональности модели DOM.

Расширенное применение функций

Функции входят в число самых интересных элементов JavaScript. Они могут быть довольно простыми и процедурными по природе или очень сложными и динамичными. Возможности функций можно расширить с помощью замыканий, к тому же с указателями на функции очень легко работать, потому что все JavaScript-функции являются объектами. Все это делает их интересными и мощными средствами программирования. В последующих разделах описаны некоторые расширенные способы использования JavaScript-функций.

Безопасное распознавание типов

Встроенные в JavaScript механизмы распознавания типов не предотвращают ошибки, совершаемые по неосторожности, и на самом деле иногда дают неправильные

результаты. Например, у оператора `typeof` есть особенности, которые делают его ненадежным при распознавании данных некоторых типов. Так, Safari до версии 4 включительно возвращает `"function"`, если оператор `typeof` применяется к регулярному выражению, поэтому сложно однозначно определить, является ли значение функцией.

Оператор `instanceof` также может стать источником проблем в том смысле, что его трудно использовать, если глобальных областей видимости несколько, например при наличии нескольких фреймов. Классический пример этой проблемы — попытка распознать объект как массив с помощью следующего кода (см. главу 5):

```
var isArray = value instanceof Array;
```

Этот код возвращает `true` только в том случае, если переменная `value` является массивом и была создана в глобальной области видимости как конструктор `Array` (если помните, `Array` является свойством объекта `window`). Если переменная `value` является массивом из другого фрейма, этот код возвращает `false`.

Другая проблема с распознаванием типов имеет место при попытке определить, является ли объект встроенным или реализован разработчиком. Проблема привлекла внимание, когда разработчики браузеров приступили к реализации встроенного объекта `JSON`. Поскольку многие уже использовали разработанную Дугласом Крокфордом (Douglas Crockford) JSON-библиотеку, содержащую глобальный объект `JSON`, разработчикам было трудно определить, какой из объектов имеется на странице.

Все эти проблемы имеют одно решение. Вызвав встроенный метод `toString()` типа `Object` с любым значением, можно получить строку формата `"[object` *ИмяВстроенногоКонструктора*`]"`. У каждого объекта есть внутреннее свойство `[[Class]]`, которое указывает имя конструктора, возвращаемое в составе этой строки, например:

```
alert(Object.prototype.toString.call(value));    // "[object Array]"
```

Поскольку имя встроенного конструктора в случае массивов тоже не зависит от глобального контекста, в котором был создан массив, вызов `toString()` возвращает согласованное значение. Это позволяет создать функцию вроде следующей:

```
function isArray(value){
    return Object.prototype.toString.call(value) == "[object Array]";
}
```

Подобный подход позволяет также определить, является ли значение встроенной функцией или регулярным выражением:

```
function isFunction(value){
    return Object.prototype.toString.call(value) == "[object Function]";
}
function isRegExp(value){
    return Object.prototype.toString.call(value) == "[object RegExp]";
}
```

Имейте в виду, что в Internet Explorer функция `isFunction()` возвращает `false` для любых функций, реализованных как COM-объекты, а не как встроенные JavaScript-функции (подробности см. в главе 10).

Этот прием также часто используется для распознавания встроенного объекта `JSON`. Метод `toString()` типа `Object` не распознает имена конструкторов, не являющихся встроенными, так что любые объекты, созданные с помощью конструкторов, определенных разработчиком, возвращают значение `"[object Object]"`. Некоторые JavaScript-библиотеки используют следующий код:

```
var isNativeJSON = window.JSON && Object.prototype.toString.call(JSON) ==
    "[object JSON]";
```

Возможность различать встроенные и внешние JavaScript-объекты очень важна для веб-разработчиков, потому что позволяет получить представление о возможностях объекта. Этот прием подойдет для точного определения возможностей любого объекта.

Помните, что значение, возвращаемое методом Object.prototype.toString(), можно переопределить. Прием, описанный в этом разделе, предполагает, что метод Object.prototype.toString() является встроенным, а не переопределенным разработчиком.

Безопасные для областей видимости конструкторы

В главе 6 мы обсудили определение пользовательских объектов с помощью конструкторов. Если помните, конструктор — это просто функция, которая вызывается с помощью оператора `new`. При таком способе использования объект `this` внутри конструктора указывает на новый экземпляр типа, например:

Листинг ScopeSafeConstructorsExample01.htm


```
function Person(name, age, job){
    this.name = name;
    this.age = age;
    this.job = job;
}

var person = new Person("Nicholas", 29, "Software Engineer");
```

В этом примере конструктор `Person` задает с помощью объекта `this` свойства `name`, `age` и `job`. При использовании конструктора с оператором `new` создается новый объект `Person`, которому назначаются свойства. Проблема возникает, если конструктор вызывается без оператора `new`. Поскольку указатель `this` связывается со значением во время выполнения, непосредственный вызов `Person()` сопоставляет `this` с глобальным объектом (`window`), что приводит к случайному расширению не того объекта, например:

Листинг ScopeSafeConstructorsExample01.htm

```
var person = Person("Nicholas", 29, "Software Engineer");
alert(window.name);         // "Nicholas"
alert(window.age);          // 29
alert(window.job);          // "Software Engineer"
```

Здесь объект `window` расширяется тремя свойствами, предназначенными для экземпляра `Person`. Это происходит потому, что конструктор был вызван как обычная функция, без оператора `new`. Проблема возникает из-за позднего связывания объекта `this`, который в данном случае был сопоставлен с объектом `window`. Поскольку свойство `name` объекта `window` используется для идентификации объектов ссылок и фреймов, такая случайная перезапись свойства может привести и к другим ошибкам на странице. Решить проблему можно, создав *безопасный для областей видимости конструктор* (scope-safe constructor).

Такие конструкторы перед выполнением каких-либо изменений проверяют, является ли объект `this` экземпляром правильного типа. Если нет, конструктор создает и возвращает новый экземпляр, например:

Листинг ScopeSafeConstructorsExample02.htm


```
function Person(name, age, job){
    if (this instanceof Person){
        this.name = name;
        this.age = age;
        this.job = job;
    } else {
        return new Person(name, age, job);
    }
}

var person1 = Person("Nicholas", 29, "Software Engineer");
alert(window.name);         // ""
alert(person1.name);        // "Nicholas"

var person2 = new Person("Shelby", 34, "Ergonomist");
alert(person2.name);        // "Shelby"
```

Инструкция `if` в этом конструкторе `Person` проверяет, является ли объект `this` экземпляром `Person`. Если да, это означает, что был использован оператор `new` либо конструктор был вызван в контексте существующего экземпляра `Person`. В обоих случаях инициализация объекта продолжается обычным образом. Если `this` не является экземпляром `Person`, конструктор вызывается еще раз с помощью оператора `new`, а созданный объект возвращается из функции. В итоге вызов конструктора `Person` с оператором `new` или без него возвращает новый экземпляр `Person`, что предотвращает случайное задание свойств глобального объекта.

При использовании безопасных для областей видимости конструкторов нужно иметь в виду, что при этом вы фиксируете контекст, в котором может быть вызван конструктор. Если используется паттерн наследования, основанный на краже

конструктора, но цепочка прототипов не применяется, наследование может быть нарушено, например:

Листинг ScopeSafeConstructorsExample03.htm

```
function Polygon(sides){
    if (this instanceof Polygon) {
        this.sides = sides;
        this.getArea = function(){
            return 0;
        };
    } else {
        return new Polygon(sides);
    }
}

function Rectangle(width, height){
    Polygon.call(this, 2);
    this.width = width;
    this.height = height;
    this.getArea = function(){
        return this.width * this.height;
    };
}

var rect = new Rectangle(5, 10);
alert(rect.sides);        // undefined
```

В этом коде конструктор `Polygon` безопасен для области видимости, а конструктор `Rectangle` — нет. При создании нового экземпляра `Rectangle` он должен унаследовать свойство `sides` от типа `Polygon` благодаря вызову `Polygon.call()`. Однако из-за того, что конструктор `Polygon` безопасен для области видимости, объект `this` не является экземпляром `Polygon`, поэтому создается и возвращается новый объект `Polygon`. Объект `this` в конструкторе `Rectangle` не расширяется, а значение, возвращенное методом `Polygon.call()`, не используется, так что у созданного объекта `Rectangle` нет свойства `sides`.

Эта проблема не возникает, если с паттерном Кража конструктора используется цепочка прототипов или паразитное комбинированное наследование. Рассмотрим пример:

Листинг ScopeSafeConstructorsExample04.htm


```
function Polygon(sides){
    if (this instanceof Polygon) {
        this.sides = sides;
        this.getArea = function(){
            return 0;
        };
    } else {
        return new Polygon(sides);
    }
}
```

```
function Rectangle(width, height){
    Polygon.call(this, 2);
    this.width = width;
    this.height = height;
    this.getArea = function(){
        return this.width * this.height;
    };
}

Rectangle.prototype = new Polygon();

var rect = new Rectangle(5, 10);
alert(rect.sides);        // 2
```

В этом коде экземпляр `Rectangle` является также экземпляром типа `Polygon`, так что метод `Polygon.call()` работает надлежащим образом, добавляя свойство `sides` к экземпляру `Rectangle`.

Безопасные для областей видимости конструкторы полезны, если несколько разработчиков пишут JavaScript-код, который должен выполняться на одной странице. В этом контексте случайные изменения глобального объекта могут привести к ошибкам, которые часто трудно отследить. Использование безопасных для областей видимости конструкторов считается оптимальной методикой, если только вы не реализуете наследование, используя исключительно кражу конструктора.

Отложенная загрузка функций

Из-за различий браузеров большинство JavaScript-сценариев содержат много инструкций `if`, предназначенных для выбора нужной ветви кода. Возьмем для примера функцию `createXHR()` из предыдущей главы:

```
function createXHR(){
    if (typeof XMLHttpRequest != "undefined"){
        return new XMLHttpRequest();
    } else if (typeof ActiveXObject != "undefined"){
        if (typeof arguments.callee.activeXString != "string"){
            var versions = ["MSXML2.XMLHttp.6.0", "MSXML2.XMLHttp.3.0",
                            "MSXML2.XMLHttp"],
                i, len;

            for (i=0,len=versions.length; i < len; i++){
                try {
                    new ActiveXObject(versions[i]);
                    arguments.callee.activeXString = versions[i];
                    break;
                } catch (ex){
                    // никакой код не требуется
                }
            }
        }
```

```
        return new ActiveXObject(arguments.callee.activeXString);
    } else {
        throw new Error("No XHR object available.");
    }
}
```

При каждом вызове функции `createXHR()` она проверяет, какие возможности поддерживаются браузером. Сначала функция проверяет доступность встроенного объекта XHR, а после этого — доступность объекта XHR, основанного на ActiveX. Если ни один из них обнаружить не удается, генерируется ошибка. Эти действия выполняются при каждом вызове функции, хотя результат ветвления от вызова к вызову не меняется: если браузер поддерживает встроенный объект XHR, он поддерживает его всегда, так что проверка становится ненужной. Код, содержащий дополнительную инструкцию `if`, работает медленнее, чем код без нее, что открывает возможности для оптимизации. Для этого используется прием, который называется *отложенной загрузкой* (lazy loading).

При отложенной загрузке ветвление выполняется только один раз. Есть два способа реализации отложенной загрузки. Первый предполагает изменение функции при ее первом вызове. В этом случае при первом вызове функции она перезаписывается другой функцией, выполнение которой делает ненужным ветвление при последующих вызовах функции. Например, для реализации отложенной загрузки функцию `createXHR()` можно переписать следующим образом:

Листинг LazyLoadingExample01.htm


```
function createXHR(){
    if (typeof XMLHttpRequest != "undefined"){
        createXHR = function(){
            return new XMLHttpRequest();
        };
    } else if (typeof ActiveXObject != "undefined"){
        createXHR = function(){
            if (typeof arguments.callee.activeXString != "string"){
                var versions = ["MSXML2.XMLHttp.6.0", "MSXML2.XMLHttp.3.0",
                                "MSXML2.XMLHttp"],
                    i, len;

                for (i=0,len=versions.length; i < len; i++){
                    try {
                      new ActiveXObject(versions[i]);
                      arguments.callee.activeXString = versions[i];
                        break;
                    } catch (ex){
                        // никакой код не требуется
                    }
                }
            }

            getRelatedTarget: function(event){};
        };
    } else {
```

```
        createXHR = function(){
            throw new Error("No XHR object available.");
        };
    }

    return createXHR();
}
```

В версии `createXHR()` с отложенной загрузкой каждая ветвь инструкции `if` назначает переменной `createXHR` другую функцию, по сути, перезаписывая первоначальную функцию. Функция, назначенная переменной `createXHR`, вызывается в конце приведенного фрагмента. При следующем вызове `createXHR()` будет вызвана функция, назначенная переменной, в результате инструкции `if` не будут обрабатываться заново.

Второй способ отложенной загрузки предполагает указание правильной функции при объявлении функции-контейнера. В этом случае вместо небольшого снижения быстродействия при первом вызове функции имеет место небольшая задержка при первой загрузке функции. Рассмотрим предыдущий пример, в котором реализован этот шаблон:

Листинг LazyLoadingExample02.htm


```
var createXHR = (function(){
    if (typeof XMLHttpRequest != "undefined"){
        return function(){
            return new XMLHttpRequest();
        };
    } else if (typeof ActiveXObject != "undefined"){
        return function(){
            if (typeof arguments.callee.activeXString != "string"){
                var versions = ["MSXML2.XMLHttp.6.0", "MSXML2.XMLHttp.3.0",
                                "MSXML2.XMLHttp"],
                    i, len;

                for (i=0,len=versions.length; i < len; i++){
                    try {
                        new ActiveXObject(versions[i]);
                        arguments.callee.activeXString = versions[i];
                        break;
                    } catch (ex){
                        // никакой код не требуется
                    }
                }
            }

            return new ActiveXObject(arguments.callee.activeXString);
        };
    } else {
        return function(){
            throw new Error("No XHR object available.");
        };
    }
})();
```

Суть этого примера в создании автоматически вызываемой анонимной функции, которая определяет, какую из реализаций функции следует использовать. Обратите внимание, что логика кода та же, что и в предыдущем примере. Мы только изменили первую строку (в которой теперь имеется ключевое слово `var` для определения функции), добавили автоматически вызываемую анонимную функцию и изменили ветви кода так, чтобы каждая из них возвращала правильное определение функции, которое немедленно назначается функции `createXHR()`.

Отложенная загрузка функций привлекательна тем, что при ветвлении кода быстродействие снижается только один раз. Какой из двух шаблонов лучше, зависит от ваших уникальных требований, но оба они позволяют сократить объем выполняемого кода.

Связывание функций

Другой прием, который становится все более популярным, называется *связыванием функции* (function binding). Он подразумевает создание функции, которая вызывает другую функцию со специфическим значением `this` и специфическими аргументами. Этот прием часто используется в сочетании с функциями обратного вызова и обработчиками событий для сохранения контекста выполнения кода при передаче функций в качестве переменных. Рассмотрим пример:

```
var handler = {
    message: "Event handled",

    handleClick: function(event){
        alert(this.message);
    }
};

var btn = document.getElementById("my-btn");
EventUtil.addHandler(btn, "click", handler.handleClick);
```

Этот код создает объект `handler` с методом `handler.handleClick()`, который назначается DOM-кнопке в качестве обработчика события. При щелчке на этой кнопке обработчик выводит оповещение. Может показаться, что в оповещении должна быть выведена строка `"Event handled"` (событие обработано), но на самом деле выводится значение `"undefined"`. Проблема в том, что контекст метода `handler.handleClick()` не сохраняется, из-за чего в большинстве браузеров объект `this` указывает на кнопку DOM, а не на объект `handler` (в Internet Explorer до версии 8 включительно объект `this` указывает на объект `window`). Проблему можно устранить с помощью замыкания, например:

```
var handler = {
    message: "Event handled",

    handleClick: function(event){
        alert(this.message);
```

```
    }
};

var btn = document.getElementById("my-btn");
EventUtil.addHandler(btn, "click", function(event){
    handler.handleClick(event);
});
```

Замыкание используется в этом коде для вызова метода `handler.handleClick()` внутри обработчика события `click`. Конечно, это очень специфичное решение для данного конкретного фрагмента кода. Использование большого количества замыканий может затруднить понимание и отладку кода. По этой причине во многих JavaScript-библиотеках реализована функция, которая связывает функцию с конкретным контекстом. Обычно она называется `bind()`.

Простая функция `bind()` принимает функцию и контекст и возвращает функцию, которая вызывает указанную функцию в заданном контексте с теми же аргументами. Ее синтаксис таков:

Листинг FunctionBindingExample01.htm


```
function bind(fn, context){
    return function(){
        return fn.apply(context, arguments);
    };
}
```

Эта обманчиво простая функция на самом деле очень полезна. Мы создаем в ней замыкание, которое вызывает полученную функцию с помощью метода `apply()`, передавая в него объект `context` и аргументы. Обратите внимание на то, что здесь объект `arguments` предназначен для внутренней функции, а не для функции `bind()`. При вызове возвращенной функции она вызовет переданную функцию в указанном контексте, передав ей все аргументы. Использовать функцию `bind()` можно следующим образом:

Листинг FunctionBindingExample01.htm

```
var handler = {
    message: "Event handled",

    handleClick: function(event){
        alert(this.message);
    }
};

var btn = document.getElementById("my-btn");
EventUtil.addHandler(btn, "click", bind(handler.handleClick, handler));
```

В этом примере функция `bind()` служит для создания функции, которую можно передать в метод `EventUtil.addHandler()` с сохранением контекста. Объект `event` также передается в функцию:

Листинг FunctionBindingExample01.htm


```
var handler = {
    message: "Event handled",

    handleClick: function(event){
        alert(this.message + ":" + event.type);
    }
};

var btn = document.getElementById("my-btn");
EventUtil.addHandler(btn, "click", bind(handler.handleClick, handler));
```

Объект `event` передается методу `handler.handleClick()` обычным образом, поскольку все аргументы передаются непосредственно методу через связанную функцию.

Чтобы сделать этот процесс еще проще, в ECMAScript 5 реализован системный метод `bind()`, доступный для всех функций. Вместо того чтобы определять собственную функцию `bind()`, вы можете вызвать этот метод непосредственно для самой функции, например:

Листинг FunctionBindingExample02.htm

```
var handler = {
    message: "Event handled",

    handleClick: function(event){
        alert(this.message + ":" + event.type);
    }
};

var btn = document.getElementById("my-btn");
EventUtil.addHandler(btn, "click", handler.handleClick.bind(handler));
```

Встроенный метод `bind()` работает подобно предыдущему в том смысле, что вы передаете в него объект, который должен быть значением `this`. Встроенный метод `bind()` доступен в Internet Explorer 9+, Firefox 4+ и Chrome.

Связанные функции полезны, если нужно передать указатель на функцию в качестве значения и выполнить эту функцию в конкретном контексте. Они чаще всего используются для обработки событий, а также вместе с методами `setTimeout()` и `setInterval()`. Однако связанные функции потребляют больше ресурсов, чем обычные, — они требуют больше памяти и работают немного медленнее из-за дополнительных вызовов функций, — поэтому лучше применять их только при необходимости.

Каррирование функций

Со связыванием функций тесно связано *каррирование функций* (function currying). При каррировании создается функция с одним или несколькими уже заданными аргументами, что также называют *частичным применением функции* (partial function

application). Основы этого приема те же, что и при связывании функции, а именно — возвращение новой функции с помощью замыкания. Отличие каррирования в том, что для новой функции дополнительно задаются некоторые аргументы, передаваемые в функцию при ее вызове. Рассмотрим пример:

```
function add(num1, num2){
    return num1 + num2;
}

function curriedAdd(num2){
    return add(5, num2);
}

alert(add(2, 3));        // 5
alert(curriedAdd(3));    // 8
```

Этот код определяет две функции: `add()` и `curriedAdd()`. Вторая по сути является версией `add()`, которая во всех случаях присваивает первому аргументу значение 5. Хотя технически функция `curriedAdd()` не является каррированной, она неплохо поясняет концепцию.

Каррированную функцию обычно создают динамически путем вызова другой функции и ее передачи с аргументами в каррируемую функцию. Вот универсальный способ создания каррированных функций:

Листинг FunctionCurryingExample01.htm

```
function curry(fn){
    var args = Array.prototype.slice.call(arguments, 1);
    return function(){
        var innerArgs = Array.prototype.slice.call(arguments),
            finalArgs = args.concat(innerArgs);
        return fn.apply(null, finalArgs);
    };
}
```

Главная задача этой функции `curry()` — упорядочение аргументов возвращаемой функции. Первым аргументом `curry()` является функция, которую нужно каррировать, а все остальные аргументы представляют передаваемые в нее значения. Чтобы получить аргументы после первого, мы вызываем метод `slice()` объекта `arguments`, передавая в него аргумент 1, который указывает, что первым элементом возвращаемого массива должен быть второй аргумент. В результате массив `args` содержит аргументы внешней функции. Для внутренней функции создается массив `innerArgs`, в котором сохраняются все переданные аргументы (здесь также используется метод `slice()`). После сохранения аргументов внешней и внутренней функций в массивах можно вызвать метод `concat()` для их объединения в переменной `finalArgs` и передать результат в функцию методом `apply()`. Имейте в виду, что приведенная здесь функция не учитывает контекст, так что в качестве первого аргумента в метод `apply()` передается значение `null`. Функцию `curry()` можно использовать следующим образом:

Листинг FunctionCurryingExample01.htm


```
function add(num1, num2){
    return num1 + num2;
}

var curriedAdd = curry(add, 5);
alert(curriedAdd(3));    // 8
```

В этом примере создается каррированная версия функции `add()`, у которой первый аргумент связан со значением 5. При вызове функции `curriedAdd()` с аргументом 3 он становится вторым аргументом функции `add()`, тогда как ее первый аргумент все еще равен 5, что дает в результате 8. Вы также можете указать все аргументы функции, например:

Листинг FunctionCurryingExample01.htm

```
function add(num1, num2){
    return num1 + num2;
}

var curriedAdd = curry(add, 5, 12);
alert(curriedAdd());    // 17
```

Здесь в каррированную функцию `add()` передаются оба аргумента, так что указывать их позже не требуется.

Каррирование функции часто используется в рамках ее связывания, что позволяет создать более сложную функцию `bind()`:

Листинг FunctionCurryingExample02.htm

```
function bind(fn, context){
    var args = Array.prototype.slice.call(arguments, 2);
    return function(){
        var innerArgs = Array.prototype.slice.call(arguments),
            finalArgs = args.concat(innerArgs);
        return fn.apply(context, finalArgs);
    };
}
```

Главные отличия данного кода от функции `curry()` — это количество аргументов, передаваемых в функцию, и влияние этого на результат выполнения кода. В то время как функция `curry()` просто принимает функцию, для которой нужно создать обертку, функция `bind()` принимает функцию и объект `context`. Это означает, что аргументы связанной функции начинаются с третьего, а не со второго, что требует изменения первого вызова метода `slice()`. Другое изменение состоит в том, что в третьей строке с конца в функцию `apply()` передается объект `context`. При вызове функции `bind()` она возвращает функцию, которая связана с указанным контекстом и может иметь некоторое количество уже заданных аргументов. Это может быть полезно, если в дополнение к объекту `event` требуется передать в обработчик событий некоторые аргументы, например:

Листинг FunctionCurryingExample02.htm


```
var handler = {
    message: "Event handled",

    handleClick: function(name, event){
        alert(this.message + ":" + name + ":" + event.type);
    }
};

var btn = document.getElementById("my-btn");
EventUtil.addHandler(btn, "click", bind(handler.handleClick, handler,
                     "my-btn"));
```

В этом обновленном примере метод `handler.handleClick()` принимает два аргумента: имя элемента, с которым вы работаете, и объект `event`. Имя передается в качестве третьего аргумента в метод `bind()`, а затем — в метод `handler.handleClick()`, который еще принимает объект `event`.

В методе `bind()` из ECMAScript 5 также реализовано каррирование. Просто передайте в него дополнительные аргументы после значения `this`:

Листинг FunctionCurryingExample03.htm

```
var handler = {
    message: "Event handled",

    handleClick: function(name, event){
        alert(this.message + ":" + name + ":" + event.type);
    }
};

var btn = document.getElementById("my-btn");
EventUtil.addHandler(btn, "click", handler.handleClick.bind(handler,
                     "my-btn"));
```

Каррированные и связанные функции обеспечивают мощные возможности для динамического создания функций в JavaScript. Выбор функции `bind()` или `curry()` зависит от того, требуется ли использовать объект `context`. С помощью этих функций можно реализовывать сложные алгоритмы и функциональность, но увлекаться ими не стоит из-за дополнительных требований к ресурсам.

Защищенные от изменений объекты

Один из недостатков JavaScript, который долго вызывал нарекания, связан с совместным доступом к объектам: каждый объект может быть изменен любым кодом, который выполняется в том же контексте. Из-за этого разработчики иногда случайно перезаписывали код друг друга или, хуже того, вносили во встроенные объекты несовместимые изменения. Для решения этой проблемы разработчики

ECMAScript 5 реализовали возможность создания *защищенных от изменений объектов* (tamper-proof objects).

В главе 6 мы рассмотрели природу свойств объектов и обсудили, как можно вручную задавать атрибуты `[[Configurable]]`, `[[Writable]]`, `[[Enumerable]]`, `[[Value]]`, `[[Get]]` и `[[Set]]` для изменения поведения свойств. Подобно этому в ECMAScript 5 были добавлены некоторые методы, позволяющие указать, как должен работать весь объект.

Здесь важно помнить, что после защиты объекта от изменений отменить эту операцию будет невозможно.

Нерасширяемые объекты

По умолчанию все JavaScript-объекты являются *расширяемыми* (extensible), то есть вы можете в любое время добавлять к объекту свойства и методы. Например, можно определить объект и позднее добавить в него другое свойство:

```
var person = { name: "Nicholas" };
person.age = 29;
```

Хотя объект `person` полностью определен в первой строке, во второй строке мы добавляем к нему еще одно свойство. С помощью метода `Object.preventExtensions()` можно запретить добавление к объекту новых свойств и методов, например:

Листинг NonExtensibleObjectsExample01.htm

```
var person = { name: "Nicholas" };
Object.preventExtensions(person);

person.age = 29;
alert(person.age);    // undefined
```

После вызова метода `Object.preventExtensions()` добавлять новые свойства или методы к объекту `person` будет нельзя. В нестрогом режиме попытка добавить новый член к объекту просто игнорируется, поэтому свойство `person.age` имеет значение `undefined`. В строгом режиме попытка добавить член к объекту, расширять который запрещено, вызывает ошибку.

Даже если к объекту нельзя добавлять новые члены, все имеющиеся у него члены остаются без изменений, и вы можете изменять и удалять их. С помощью метода `Object.isExtensible()` можно узнать, разрешено ли расширение объекта:

Листинг NonExtensibleObjectsExample02.htm


```
var person = { name: "Nicholas" };
alert(Object.isExtensible(person));        // true

Object.preventExtensions(person);
alert(Object.isExtensible(person));        // false
```

Запечатанные объекты

Следующий уровень защиты объектов в ECMASCript 5 — *запечатанные объекты* (sealed objects). Запечатанные объекты нельзя расширять, а у членов, которые у них уже есть, атрибут `[[Configurable]]` имеет значение `false`. Это означает, что свойства и методы нельзя удалять, а свойства с данными с помощью метода `Object.defineProperty()` нельзя заменять свойствами с функциями доступа, и наоборот. Значения свойств изменять можно.

Запечатать объект можно с помощью метода `Object.seal()`:

Листинг SealedObjectsExample01.htm


```
var person = { name: "Nicholas" };
Object.seal(person);

person.age = 29;
alert(person.age);          // undefined

delete person.name;
alert(person.name);         // "Nicholas"
```

В этом примере попытка добавить свойство `age` игнорируется. Попытка удалить свойство `name` также игнорируется, так что значение остается без изменений. Так код работает в нестрогом режиме. В строгом режиме попытка добавить или удалить член запечатанного объекта приводит к ошибке.

Узнать, запечатан ли объект, можно с помощью метода `Object.isSealed()`. Поскольку запечатанный объект расширять нельзя, метод `Object.isExtensible()` для него возвращает `false`:

Листинг SealedObjectsExample02.htm

```
var person = { name: "Nicholas" };
alert(Object.isExtensible(person));       // true
alert(Object.isSealed(person));           // false

Object.seal(person);
alert(Object.isExtensible(person));       // false
alert(Object.isSealed(person));           // true
```

Замороженные объекты

Самый надежный уровень защиты от изменений — *замороженные объекты* (frozen objects). Их нельзя расширять, они запечатаны, а у их свойств с данными атрибут `[[Writable]]` имеет значение `false`. Свойства с функциями доступа доступны для записи, но только если определена функция `[[Set]]`. В ECMAScript 5 для замораживания объектов используется метод `Object.freeze()`:

Листинг FrozenObjectsExample01.htm

```
var person = { name: "Nicholas" };
Object.freeze(person);

person.age = 29;
alert(person.age);          // undefined

delete person.name;
alert(person.name);         // "Nicholas"

person.name = "Грег";
alert(person.name);         // "Nicholas"
```

Как и в случае нерасширяемых и запечатанных объектов, попытка выполнить запрещенные операции для замороженного объекта в нестрогом режиме игнорируется, а в строгом приводит к ошибке.

Для распознавания замороженных объектов хорошо подходит метод `Object.isFrozen()`. Поскольку замороженные объекты являются запечатанными и расширяемыми, метод `Object.isExtensible()` для них возвращает `false`, а метод `Object.isSealed()` — `true`:

Листинг FrozenObjectsExample02.htm


```
var person = { name: "Nicholas" };
alert(Object.isExtensible(person));       // true
alert(Object.isSealed(person));           // false
alert(Object.isFrozen(person));           // false

Object.freeze(person);
alert(Object.isExtensible(person));       // false
alert(Object.isSealed(person));           // true
alert(Object.isFrozen(person));           // true
```

Замороженные объекты особенно полезны при разработке библиотек. Одна из частых проблем с JavaScript-библиотеками состоит в том, что разработчики случайно (или намеренно) изменяют главный объект библиотеки. Замораживание (или запечатывание) главного объекта библиотеки помогает предотвратить некоторые из таких ошибок.

Расширенные возможности работы с таймерами

Таймеры, создаваемые с помощью методов `setTimeout()` и `setInterval()`, можно использовать для реализации интересного и полезного функционала. Распространено заблуждение, что JavaScript-таймеры являются программными потоками, но на самом деле JavaScript работает в однопоточной среде. Таким образом, таймеры

просто планируют запуск кода на какой-то момент в будущем. Точность времени запуска кода не гарантируется, потому что в течение жизненного цикла страницы контроль над JavaScript-процессом может принадлежать другим фрагментам кода. Код, запускаемый при загрузке страницы, обработчики событий и Ajax-функции обратного вызова выполняются в одном и том же потоке, а определять приоритеты разных фрагментов кода в конкретный момент времени — ответственность браузера.

Иногда полезно представить выполнение JavaScript-кода соответственно временной шкале. При загрузке страницы сначала выполняется любой код, добавленный в нее с помощью элемента `<script>`. Зачастую это просто объявления функций и переменных, используемые позже в течение существования страницы, но иногда этот код может также выполнять первоначальную обработку данных. После этого JavaScript-процесс ожидает указания выполнить другой код. Например, если JavaScript-процесс не занят, при щелчке на кнопке немедленно вызывается обработчик события `click`. Хронология обработки такой страницы может выглядеть так, как показано на рис. 22.1.

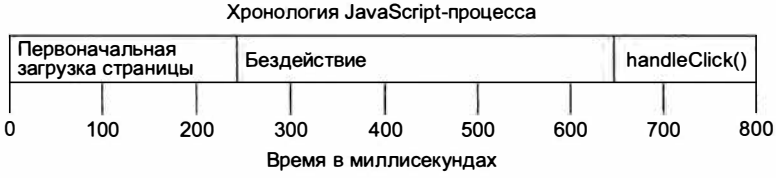

Рис. 22.1

Кроме главного JavaScript-процесса есть также очередь фрагментов кода, который должен быть выполнен, когда процесс в следующий раз перейдет в состояние бездействия. В течение жизненного цикла страницы код добавляется в очередь в том порядке, в котором он должен выполняться. Например, при щелчке на кнопке в очередь добавляется код обработчика этого события, который выполняется, когда появляется такая возможность. При получении Ajax-ответа в очередь добавляется код функции обратного вызова. Никакой JavaScript-код не выполняется незамедлительно; он запускается, как только процесс переходит в состояние бездействия.

Таймеры работают с этой очередью, вставляя в нее код по прошествии определенного интервала времени. Имейте в виду, что добавление кода в очередь не означает, что он будет выполнен немедленно, но код будет запущен, как только появится такая возможность. Установка таймера на 150 миллисекунд не гарантирует, что код будет выполнен через 150 миллисекунд, вместо этого код добавится в очередь через 150 миллисекунд. Если в этот момент очередь будет пустой, код таймера начнет выполняться незамедлительно, из-за чего может сложиться впечатление, что он выполняется точно в указанное время. В других ситуациях для выполнения кода может потребоваться гораздо больше времени.

Рассмотрим следующий фрагмент:

```
var btn = document.getElementById("my-btn");
btn.onclick = function(){
    setTimeout(function(){
        document.getElementById("message").style.visibility = "visible";
    }, 250);

    // другой код
};
```

Обработчик щелчка на кнопке в этом фрагменте задает таймер, который должен быть вызван через 250 миллисекунд. При щелчке на кнопке обработчик `onclick` сначала добавляется в очередь. Когда он вызывается, запускается таймер, а через 250 миллисекунд указанный код добавляется в очередь для выполнения. По сути, вызов `setTimeout()` указывает, что какой-то код должен быть выполнен позже.

Самый важный нюанс в работе таймеров состоит в том, что заданный интервал показывает, когда код таймера будет добавлен в очередь, а не выполнен на самом деле. Если для выполнения обработчика `onclick` в предыдущем примере потребуется 300 миллисекунд, то код таймера будет запущен не раньше, чем через 300 миллисекунд после установки таймера. Независимо от того, как код был добавлен в очередь, он может быть выполнен, только когда освободится JavaScript-процесс (рис. 22.2).

Рис. 22.2

Как видно на рис. 22.2, хотя код таймера был добавлен через 255 миллисекунд, он не может быть выполнен сразу, потому что в это время все еще выполняется обработчик события `click`. Выполнение кода таймера становится возможным спустя 300 миллисекунд, то есть после завершения обработчика события `click`.

Firefox позволяет узнать, насколько запоздал таймер. Для этого используется разница между временем запуска таймера и указанным интервалом, например:

```
// работает только в Firefox
setTimeout(function(diff){
    if (diff > 0) {
        // таймер запоздал
    } else if (diff < 0){
        // таймер сработал раньше положенного
    } else {
        // таймер был вызван вовремя
    }
}, 250);
```

Завершив выполнение одного блока кода, JavaScript-процесс приостанавливается, чтобы могли быть выполнены другие процессы на странице. Поскольку JavaScript-процесс блокирует другие процессы страницы, эти небольшие перерывы необходимы для предотвращения блокировки пользовательского интерфейса (что все же возможно, если код выполняется слишком долго). Установка таймера гарантирует, что перед выполнением кода таймера процесс будет прерван хотя бы один раз.

Повторяющиеся таймеры

Таймеры, созданные с помощью метода `setInterval()`, гарантируют регулярную вставку кода таймера в очередь. Проблема с этим подходом состоит в том, что код таймера может не завершиться, прежде чем будет добавлен в очередь снова. В результате код таймера может быть выполнен несколько раз подряд без перерыва. К счастью, интерпретаторы JavaScript достаточно интеллектуальны, чтобы избегать этой проблемы. При использовании метода `setInterval()` код таймера добавляется в очередь, только если в ней нет других экземпляров кода таймера. Это гарантирует, что время между моментами добавления кода таймера в очередь будет не меньше указанного интервала.

Недостатков у такого управления повторяющимися таймерами два: во-первых, интервалы могут пропускаться, во-вторых, интервалы между запусками кода таймера могут быть меньше ожидаемых. Предположим, что обработчик события `click` задает с помощью метода `setInterval()` повторяющийся таймер с интервалом 200 миллисекунд. Если для выполнения обработчика события требуется чуть больше 300 миллисекунд, а код таймера выполняется примерно столько же, интервал будет пропущен, а код таймера будет выполняться раз за разом (рис. 22.3).

Первый таймер в этом примере добавляется в очередь через 205 миллисекунд, но не может быть выполнен до отметки 300 миллисекунд. При выполнении кода таймера в очередь добавляется еще одна его копия, которая должна быть запущена на 405-й миллисекунде. В начале следующего интервала, на 605-й миллисекунде, код первого таймера все еще выполняется, а в очереди находится другой экземпляр кода таймера, поэтому новый код таймера в очередь не добавляется. Код таймера, добавленный на 405-й миллисекунде, выполняется сразу же после кода таймера, который был добавлен на 5-й миллисекунде.

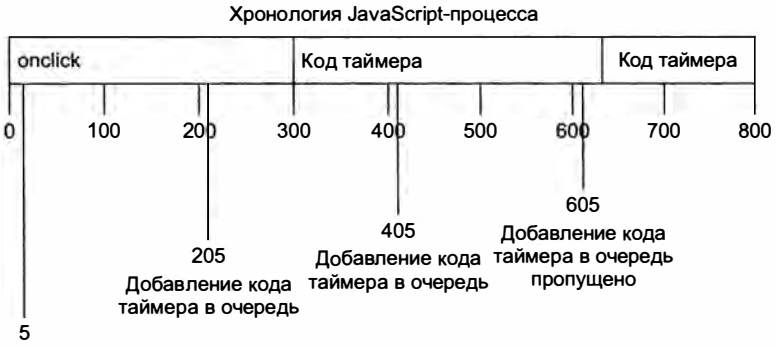

Рис. 22.3

Для решения этих проблем с повторяющимися таймерами, добавленными с помощью метода `setInterval()`, можно использовать сцепленные вызовы `setTimeout()` следующим образом:

```
setTimeout(function(){

    // обработка

    setTimeout(arguments.callee, interval);

}, interval);
```

В этом шаблоне вызовы `setTimeout()` связываются в цепочку, создавая новый таймер при каждом вызове функции. Во втором вызове `setTimeout()` для получения ссылки на выполняемую функцию и установки другого таймера для нее используется свойство `arguments.callee`. Преимущество такого подхода в том, что код нового таймера не вставляется в очередь, пока не будет выполнен код предыдущего таймера, — это гарантирует, что никакие интервалы не окажутся пропущенными. Кроме того, это гарантирует, что перед следующим запуском кода таймера пройдет как минимум указанный интервал, что предотвращает последовательный запуск кода таймеров без перерыва. Этот шаблон чаще всего используется для реализации повторяющихся таймеров, например:

Листинг RepeatingTimersExample.htm


```
setTimeout(function(){

    var div = document.getElementById("myDiv"),
        left = parseInt(div.style.left) + 5;
    div.style.left = left + "px";

    if (left < 200){
        setTimeout(arguments.callee, 50);
    }

}, 50);
```

Этот код перемещает элемент вправо при каждом запуске кода таймера. Перемещение прекращается, когда левая координата достигает значения 200 пикселей. Этот шаблон часто используется для анимации с помощью JavaScript.

У каждого окна, вкладки или фрейма браузера есть собственная очередь выполнения кода. Это означает, что при синхронном выполнении кода JavaScript-вызовы между фреймами или окнами могут спровоцировать конкуренцию. Если необходимо реализовать взаимодействие такого типа, для запуска кода следует создать таймер в целевом окне или фрейме.

Управление процессами

Интерпретатору JavaScript выделяется конечный объем ресурсов в браузере. В отличие от приложений для настольных компьютеров, которым память и процессорное время часто предоставляются в изобилии, JavaScript серьезно ограничивается в этом отношении, чтобы злоумышленники не могли причинить ущерб компьютеру пользователя. Одно из ограничений регламентирует *время выполнения сценария* (long-running script) и предотвращает запуск кода, если предполагаемое время его выполнения или количество инструкций в нем превышает заданное значение. При достижении этого ограничения в браузере пользователя выводится диалоговое окно с ошибкой, указывающее, что сценарий выполняется слишком долго, и предлагающее пользователю продолжить или завершить выполнение кода. Поскольку никому из JavaScript-разработчиков не захочется разочаровывать пользователей таким сообщением, с помощью таймеров это ограничение можно обойти.

Обычно у проблемы длительного выполнения сценария два источника: глубоко вложенные вызовы функций и ресурсоемкие циклы. Задачу длительных ресурсоемких циклов решить проще. Такие циклы обычно соответствуют следующему шаблону:

```
for (var i=0, len=data.length; i < len; i++){
    process(data[i]);
}
```

Проблема здесь в том, что количество обрабатываемых элементов до выполнения кода неизвестно. Если на выполнение функции `process()` требуется 100 миллисекунд, при наличии двух элементов в массиве, возможно, беспокоиться не о чем, но при наличии десяти элементов этот цикл будет выполняться не менее секунды. Время выполнения цикла напрямую зависит от количества элементов в массиве, а поскольку JavaScript-код работает в блокирующем стиле, то чем дольше выполняется сценарий, тем дольше пользователи не смогут взаимодействовать со страницей.

Прежде чем реализовывать цикл, нужно ответить на два важных вопроса.

1. **Нужно ли обрабатывать данные синхронно?** Если обработка данных блокирует завершение какой-то другой операции, возможно, лучше ничего не менять. Если же вы можете с уверенностью ответить на этот вопрос «нет», возможно, имеет смысл отложить обработку некоторых данных на потом.

2. **Нужно ли обрабатывать данные последовательно?** Массивы часто используются для удобной группировки и перебора элементов независимо от порядка их расположения. Если порядок элементов не имеет значения, вероятно, их обработку можно отложить.

Если для выполнения цикла требуется много времени и на один из приведенных вопросов вы можете ответить отрицательно, можно разделить цикл с помощью таймеров. Этот прием называется *разделением массива* (array chunking) и предполагает, что массив обрабатывается небольшими блоками, чаще всего по одному за раз. Суть приема в том, чтобы создать очередь обрабатываемых элементов, воспользоваться таймером для извлечения очередного элемента, обработать его и установить следующий таймер. Базовый шаблон таков:

```
setTimeout(function(){

    // получение следующего элемента и его обработка
    var item = array.shift();
    process(item);

    // установка нового тайм-аута при наличии других элементов
    if(array.length > 0){
        setTimeout(arguments.callee, 100);
    }
}, 100);
```

В этом коде переменная `array` представляет список элементов, которые нужно обработать. С помощью метода `shift()` мы получаем следующий элемент в очереди, а затем передаем его в функцию `process()`. Если в очереди есть другие элементы, устанавливается следующий таймер, для чего используется свойство `arguments.callee`, указывающее на ту же анонимную функцию. Вы можете с легкостью разделить массив с помощью такой функции:

Листинг ArrayChunkingExample.htm


```
function chunk(array, process, context){
    setTimeout(function(){
        var item = array.shift();
        process.call(context, item);

        if (array.length > 0){
            setTimeout(arguments.callee, 100);
        }
    }, 100);
}
```

Функция `chunk()` принимает три аргумента: массив элементов, которые нужно обработать, функцию для обработки элементов и необязательный контекст вызова функции. Внутри функции дублируется приведенный выше базовый шаблон, при этом функция `process()` вызывается с помощью функции `call()`, чтобы при необходимости можно было задать правильный контекст. Интервал таймеров задается

равным 100 миллисекундам, что дает JavaScript-процессу время на переход в состояние бездействия между событиями обработки элементов. Этот интервал можно изменить в соответствии с конкретными требованиями, хотя в большинстве случаев 100 миллисекунд вполне подходят. Эту функцию можно использовать следующим образом:

Листинг ArrayChunkingExample.htm


```
var data = [12,123,1234,453,436,23,23,5,4123,45,346,5634,2234,345,342];

function printValue(item){
    var div = document.getElementById("myDiv");
    div.innerHTML += item + "<br>";
}

chunk(data, printValue);
```

Этот пример выводит каждое значение массива `data` в элементе `<div>`, используя функцию `printValue()`. Поскольку она относится к глобальной области видимости, передавать объект `context` в функцию `chunk()` не требуется.

Помните, что массив, передаваемый в функцию `chunk()`, используется как очередь, так что его элементы по мере обработки данных изменяются. Если нужно, чтобы исходный массив остался неизменным, следует передать в функцию `chunk()` его копию, например:

```
chunk(data.concat(), printValue);
```

Когда для массива вызывается метод `concat()` без аргументов, он возвращает массив с теми же элементами, что и в оригинале. Это гарантирует, что исходный массив не будет изменен функцией.

Разделение массива позволяет обрабатывать элементы отдельными блоками в очереди. После обработки каждого элемента могут быть запущены другие процессы браузера, что предотвращает ошибки, связанные с длительным выполнением сценариев.

Если на выполнение функции требуется более 50 миллисекунд, лучше попробовать разделить ее на меньшие, которые можно использовать с таймерами.

Регулирование функций

Некоторые вычисления и процессы в браузере потребляют больше ресурсов, чем другие. Например, при работе с DOM требуется больше памяти и процессорного времени, чем без DOM. Попытка выполнить слишком много DOM-операций может привести к зависанию или аварийному завершению работы браузера. Это часто происходит в Internet Explorer при обработке события `resize`, которое многократно генерируется при изменении размеров окна браузера. DOM-манипуляции

в обработчике события `resize` могут вызвать сбой в работе браузера из-за высокой частоты изменений. Для обходного решения этой проблемы можно *отрегулировать* (*throttle*) вызов функции с помощью таймеров.

В основе регулирования функции лежит идея, что код не должен повторно выполняться без перерыва. При первом вызове функции создается таймер, запускающий код по прошествии указанного интервала. При вызове функции во второй раз предыдущий таймер отменяется и устанавливается новый. Если предыдущий таймер уже сработал, это не имеет значения. Однако если предыдущий таймер еще не сработал, он, по сути, заменяется новым таймером. Это делается с той целью, чтобы функция была выполнена только при условии отсутствия запросов на ее выполнение в течение некоторого времени. Базовый шаблон таков:

```
var processor = {
    timeoutId: null,

    // метод, выполняющий обработку
    performProcessing: function(){
        // код обработки
    },

    // метод, инициирующий обработку
    process: function(){
        clearTimeout(this.timeoutId);

        var that = this;
        this.timeoutId = setTimeout(function(){
            that.performProcessing();
        }, 100);
    }
};

// попытка начать обработку
processor.process();
```

Этот код создает объект `processor` с двумя методами: `process()` и `performProcessing()`. Первый из них инициирует обработку, в то время как второй реально это делает. При вызове метода `process()` он прежде всего очищает сохраненное значение `timeoutId`, чтобы отменить предыдущие вызовы, а затем создает новый таймер для вызова метода `performProcessing()`. Поскольку контекстом в функции `setTimeout()` всегда является объект `window`, необходимо сохранить ссылку на `this`, чтобы ее можно было использовать позже.

Задаваемый интервал составляет 100 миллисекунд, то есть метод `performProcessing()` будет вызван не раньше, чем через 100 миллисекунд после последнего вызова `process()`. Так, если вызвать метод `process()` 20 раз в течение 100 миллисекунд, метод `performProcessing()` будет вызван только один раз.

Этот шаблон можно упростить с помощью функции `throttle()`, которая автоматически устанавливает и сбрасывает таймер:

Листинг ThrottlingExample.htm


```
function throttle(method, context) {
    clearTimeout(method.tId);
    method.tId = setTimeout(function(){
        method.call(context);
    }, 100);
}
```

Функция `throttle()` принимает два аргумента: функцию, которую нужно выполнить, и область видимости, в которой ее нужно выполнить. Первым делом функция сбрасывает любой установленный таймер. Идентификатор таймера хранится в свойстве `tId` функции, которое может отсутствовать при первой передаче метода в функцию `throttle()`. Затем создается новый таймер, а его идентификатор сохраняется в свойстве `tId` метода. Если имеет место первый вызов функции `throttle()` с этим методом, тогда этот код создает свойство. В коде таймера используется вызов `call()`, чтобы метод был выполнен в надлежащем контексте. Если второй аргумент не указан, метод выполняется в глобальной области видимости.

Как уже было сказано, регулирование чаще всего используется при обработке события `resize`. Если с помощью этого события вы меняете макет страницы, лучше регулировать обработку данных, чтобы браузеру не пришлось выполнять слишком много вычислений за короткий период времени. Предположим, например, что нам нужно изменить высоту элемента `<div>`, чтобы она всегда была равна ширине. Это можно сделать следующим образом:

```
window.onresize = function(){
    var div = document.getElementById("myDiv");
    div.style.height = div.offsetWidth + "px";
};
```

Этот очень простой пример демонстрирует несколько факторов, способных замедлить работу браузера. Во-первых, мы здесь вычисляем свойство `offsetWidth`, что может быть довольно сложной операцией, если к элементу и остальной части страницы применено много CSS-стилей. Во-вторых, задание высоты элемента требует повторной обработки страницы, чтобы эти изменения вступили в силу. Опять же, это может потребовать много ресурсов, если страница содержит большое количество элементов и CSS-стилей. Функция `throttle()` может помочь решить эту проблему, например:

Листинг ThrottlingExample.htm


```
function resizeDiv(){
    var div = document.getElementById("myDiv");
    div.style.height = div.offsetWidth + "px";
}

window.onresize = function(){
    throttle(resizeDiv);
};
```

Здесь функциональность изменения размеров перемещена в отдельную функцию `resizeDiv()`. Обработчик события `resize` вызывает функцию `throttle()`, передавая ей функцию `resizeDiv()`, вместо того чтобы непосредственно вызывать `resizeDiv()`. Во многих случаях пользователям это различие незаметно, но экономия вычислительных ресурсов может быть довольно большой.

Регулирование функции следует использовать, если какой-то код нужно выполнять только периодически, но при этом контролировать частоту его запуска вы не можете. В функции `throttle()` используется интервал, равный 100 миллисекундам, но его можно изменить.

Пользовательские события

Ранее уже отмечалось, что события — главное средство взаимодействия JavaScript-кода с браузером. События основаны на паттерне проектирования *Наблюдатель*, который используется для создания слабо связанного кода. Идея в том, что объект может публиковать события, информирующие о наступлении интересных моментов в его жизненном цикле. Другие объекты могут *наблюдать* за этим объектом, дожидаясь интересных моментов и реагируя на них запуском того или иного кода.

Паттерн Наблюдатель включает объекты двух типов: *субъект* (subject) и *наблюдатель* (observer). Субъект отвечает за публикацию событий, а наблюдатель просто наблюдает за субъектом, подписавшись на события. Ключевая концепция этого паттерна состоит в том, что субъект ничего не знает о наблюдателе, то есть он может существовать и работать должным образом даже без наблюдателя. В то же время наблюдатель знает о субъекте и регистрирует для его событий функции обратного вызова (обработчики событий). Когда вы работаете с DOM, элемент DOM является субъектом, а обработчик событий — наблюдателем.

События — очень популярный способ взаимодействия с DOM, но их также можно применять в другом коде, реализовав пользовательские события. Идея, лежащая в основе пользовательских событий, состоит в том, чтобы создать объект, который будет управлять событиями, позволяя другому коду слушать эти события. Базовый тип, реализующий эту функциональность, можно определить следующим образом:

Листинг EventTarget.js


```
function EventTarget(){
    this.handlers = {};
}

EventTarget.prototype = {
    constructor: EventTarget,

    addHandler: function(type, handler){
        if (typeof this.handlers[type] == "undefined"){
            this.handlers[type] = [];
        }
```

```
        this.handlers[type].push(handler);
    },

    fire: function(event){
        if (!event.target){
            event.target = this;
        }
        if (this.handlers[event.type] instanceof Array){
            var handlers = this.handlers[event.type];
            for (var i=0, len=handlers.length; i < len; i++){
                handlers[i](event);
            }
        }
    },

    removeHandler: function(type, handler){
        if (this.handlers[type] instanceof Array){
            var handlers = this.handlers[type];
            for (var i=0, len=handlers.length; i < len; i++){
                if (handlers[i] === handler){
                    break;
                }
            }

            handlers.splice(i, 1);
        }
    }
};
```

Тип `EventTarget` имеет единственное свойство `handlers`, которое используется для хранения обработчиков событий. У него также есть три метода: `addHandler()` регистрирует обработчик событий конкретного типа, `fire()` генерирует событие и `removeHandler()` отменяет регистрацию обработчика событий конкретного типа.

Метод `addHandler()` принимает два аргумента: тип события и функцию его обработки. При вызове метода мы проверяем, существует ли уже массив с типом события у свойства `handlers`. Если нет, он создается, а затем обработчик добавляется в конец массива методом `push()`.

Когда необходимо сгенерировать событие, вызывается метод `fire()`. В качестве единственного аргумента он принимает объект, содержащий как минимум свойство `type`. Метод `fire()` начинается с задания свойства `target` объекта `event`, если оно еще не задано. После этого он просто ищет массив обработчиков событий нужного типа и вызывает каждый из них, передавая объект `event`. Поскольку это пользовательские события, вы сами можете решать, какую дополнительную информацию нужно добавить в объект `event`.

Метод `removeHandler()` дополняет метод `addHandler()` и принимает те же аргументы: тип события и обработчик события. Этот метод ищет в массиве обработчиков событий обработчик, который нужно удалить. При его обнаружении выполняется

инструкция `break` для выхода из цикла `for`, затем метод `splice()` удаляет обработчик из массива.

Работать с пользовательскими событиями с помощью типа `EventTarget` можно следующим образом:

Листинг EventTargetExample01.htm


```
function handleMessage(event){
    alert("Message received: " + event.message); // получено сообщение
}

// создание объекта
var target = new EventTarget();

// добавление обработчика событий
target.addHandler("message", handleMessage);

// генерирование события
target.fire({ type: "message", message: "Hello world!"});

// удаление обработчика
target.removeHandler("message", handleMessage);

// вторая попытка - обработчика быть не должно
target.fire({ type: "message", message: "Hello world!"});
```

Здесь мы определяем функцию `handleMessage()` для обработки события `message`. Она принимает объект `event` и выводит свойство `message`. Далее мы вызываем метод `addHandler()` объекта `target`, передавая в него строку `"message"` и функцию `handleMessage()`. В следующей строке вызывается метод `fire()` с литералом объекта, содержащим свойства `type` и `message`. В результате вызываются обработчики события `message`, при этом выводится оповещение (из функции `handleMessage()`). Затем обработчик события удаляется, так что при следующем событии оповещение не отображается.

Благодаря тому, что эта функциональность инкапсулирована в пользовательском типе, другие объекты могут унаследовать это поведение от типа `EventTarget`, например:

Листинг EventTargetExample02.htm


```
function Person(name, age){
    EventTarget.call(this);
    this.name = name;
    this.age = age;
}

inheritPrototype(Person,EventTarget);

Person.prototype.say = function(message){
    this.fire({type: "message", message: message});
};
```

Тип `Person` наследуется от `EventTarget` с помощью паразитного комбинированного наследования (см. главу 6). Когда вызывается метод `say()`, генерируется событие с подробностями сообщения. Метод `fire()` часто вызывается в других методах типа, но довольно редко — в общедоступном коде. Этот код можно использовать следующим образом:

Листинг EventTargetExample02.htm

```
function handleMessage(event){
    alert(event.target.name + " says: " + event.message);
}

// создание пользователя
var person = new Person("Nicholas", 29);

// добавление обработчика событий
person.addHandler("message", handleMessage);

// вызов метода объекта, генерирующий событие message
person.say("Hi there.");
```

Функция `handleMessage()` в этом примере отображает оповещение с именем пользователя (полученным с помощью свойства `event.target.name`) и текстом сообщения. Когда вызывается метод `say()` с сообщением, генерируется событие `message`. Оно, в свою очередь, вызывает функцию `handleMessage()`, которая выводит оповещение.

Пользовательские события полезны, если несколько частей кода взаимодействуют между собой в конкретные моменты времени. Если каждый объект ссылается на все остальные объекты, код становится сильно связанным, что затрудняет его сопровождение, потому что изменение одного объекта влияет на другие. Применение пользовательских событий помогает ослабить связь между родственными объектами, обеспечивая изоляцию функциональности. Во многих случаях код, генерирующий события, полностью отделен от кода, слушающего эти события.

Перетаскивание

Перетаскивание — один из самых популярных приемов работы с пользовательским интерфейсом. Идея проста: нажать кнопку мыши на элементе и, удерживая ее нажатой, переместить мышь в другую область, а затем отпустить кнопку, чтобы переместить элемент. Популярность перетаскивания распространилась и на веб-приложения, где оно стало альтернативой традиционным интерфейсам конфигурации.

Суть перетаскивания в том, чтобы создать элемент с абсолютным позиционированием, который можно перемещать мышью. Эта методика уходит корнями к классическому приему веб-программирования, который называется *следом курсора* (cursor trail). След курсора — это изображение или несколько изображений, создающих

своеобразную тень от указателя мыши на странице. Код одноэлементного следа курсора включает установку обработчика события `mousemove` для документа и перемещение элемента в позицию курсора в обработчике, например:

Листинг DragAndDropExample01.htm


```
EventUtil.addHandler(document, "mousemove", function(event){
    var myDiv = document.getElementById("myDiv");
    myDiv.style.left = event.clientX + "px";
    myDiv.style.top = event.clientY + "px";
});
```

В этом примере в качестве левой и верхней координат элемента задаются значения свойств `clientX` и `clientY` объекта `event`, в результате чего элемент помещается в позицию указателя мыши в области просмотра, при этом создается впечатление, что элемент следует за указателем по странице. Чтобы реализовать перетаскивание, нужно лишь своевременно включить его (при нажатии кнопки мыши) и отключить (при отпускании кнопки мыши). Вот код очень простой операции перетаскивания:

Листинг DragAndDropExample02.htm

```
var DragDrop = function(){

    var dragging = null;

    function handleEvent(event){

        // получение объектов event и target
        event = EventUtil.getEvent(event);
        var target = EventUtil.getTarget(event);

        // определение типа события
        switch(event.type){
            case "mousedown":
                if (target.className.indexOf("draggable") > -1){
                    dragging = target;
                }
                break;

            case "mousemove":
                if (dragging !== null){

                    // назначение расположения
                    dragging.style.left = event.clientX + "px";
                    dragging.style.top = event.clientY + "px";
                }
                break;

            case "mouseup":
                dragging = null;
                break;
        }
    };
```

```
    // общедоступный интерфейс
    return {
        enable: function(){
            EventUtil.addHandler(document, "mousedown", handleEvent);
            EventUtil.addHandler(document, "mousemove", handleEvent);
            EventUtil.addHandler(document, "mouseup", handleEvent);
        },

        disable: function(){
            EventUtil.removeHandler(document, "mousedown", handleEvent);
            EventUtil.removeHandler(document, "mousemove", handleEvent);
            EventUtil.removeHandler(document, "mouseup", handleEvent);
        }
    }
}();
```

Объект `DragDrop` инкапсулирует весь базовый функционал перетаскивания. Это объект-одиночка, который использует паттерн Модуль для сокрытия некоторых деталей реализации. Переменная `dragging` сначала имеет значение `null`, а затем ей назначается перетаскиваемый элемент. Иначе говоря, если она не равна `null`, это означает, что пользователь перетаскивает какой-то элемент. Функция `handleEvent()` обрабатывает все три события мыши, связанные с перетаскиванием. Сначала она получает ссылки на объект `event` и целевой элемент события. После этого инструкция `switch` определяет, какое событие произошло. Когда происходит событие `mousedown`, сначала проверяется значение `class` целевого события, и если там присутствует класс `"draggable"`, `target` становится `dragging`. Такой метод позволяет указывать перетаскиваемые элементы с помощью разметки, а не JavaScript-кода.

Событие `mousemove` обрабатывается в функции `handleEvent()` практически так же, только в этом случае мы проверяем, имеет ли переменная `dragging` значение `null`. Если нет, она содержит перетаскиваемый элемент, и мы изменяем его позицию надлежащим образом. При обработке события `mouseup` мы просто присваиваем переменной `dragging` значение `null`, что, по сути, противоположно событию `mousemove`.

Объект `DragDrop` также содержит открытые методы `enable()` и `disable()`, которые просто подключают и отключают все обработчики событий соответственно. Эти методы обеспечивают дополнительный контроль над перетаскиванием.

Чтобы использовать объект `DragDrop`, просто включите его в код страницы и вызовите метод `enable()`. Перетаскивание автоматически будет включено для всех элементов, относящихся к классу `"draggable"`, например:

```
<div class="draggable" style="position:absolute; background:red"></div>
```

Чтобы элемент можно было перетаскивать, его позиционирование должно быть абсолютным.

Исправленное перетаскивание

Запустив приведенный пример, вы заметите, что левый верхний угол элемента всегда тянется за курсором, из-за чего возможны рывки, когда пользователь начинает перемещать мышь. В идеале все должно выглядеть так, как если бы курсор «подцеплял» элемент, то есть при перетаскивании элемента курсор должен находиться в точке нажатия кнопки мыши (рис. 22.4).

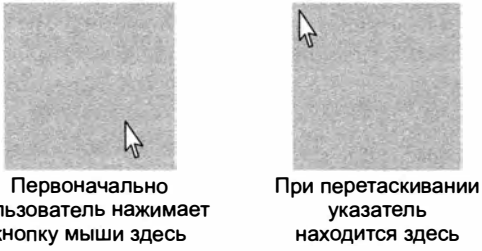

Рис. 22.4

Для достижения такого эффекта нужно вычислить разницу между левым верхним углом элемента и расположением курсора. Это значение нужно определить при генерировании события `mousedown` и сохранить до возникновения события `mouseup`. Смещение курсора от левого верхнего угла можно определить, сравнив свойства `clientX` и `clientY` объекта `event` со свойствами `offsetLeft` и `offsetTop` элемента (рис. 22.5).

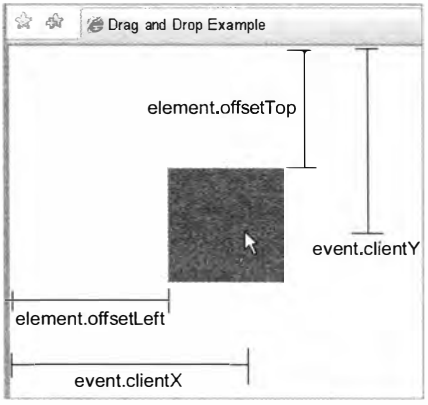

Рис. 22.5

Чтобы сохранить смещения курсора по горизонтали и вертикали, нужно создать еще пару переменных, `diffX` и `diffY`. Мы воспользуемся ими в обработчике события `mousemove` для правильного позиционирования элемента:

Листинг DragAndDropExample03.htm


```
var DragDrop = function(){

    var dragging = null,
        diffX = 0,
        diffY = 0;

    function handleEvent(event){

        // получение объектов event и target
        event = EventUtil.getEvent(event);
        var target = EventUtil.getTarget(event);

        // определение типа события
        switch(event.type){
            case "mousedown":
                if (target.className.indexOf("draggable") > -1){
                    dragging = target;
                    diffX = event.clientX - target.offsetLeft;
                    diffY = event.clientY - target.offsetTop;
                }
                break;

            case "mousemove":
                if (dragging !== null){

                    // назначение расположения
                    dragging.style.left = (event.clientX - diffX) + "px";
                    dragging.style.top = (event.clientY - diffY) + "px";

                }
                break;

            case "mouseup":
                dragging = null;
                break;
        }
    };

    // общедоступный интерфейс
    return {
        enable: function(){
            EventUtil.addHandler(document, "mousedown", handleEvent);
            EventUtil.addHandler(document, "mousemove", handleEvent);
            EventUtil.addHandler(document, "mouseup", handleEvent);
        },

        disable: function(){
            EventUtil.removeHandler(document, "mousedown", handleEvent);
            EventUtil.removeHandler(document, "mousemove", handleEvent);
            EventUtil.removeHandler(document, "mouseup", handleEvent);
        }
    }
}();
```

Переменные `diffX` и `diffY` являются закрытыми, потому что они необходимы только функции `handleEvent()`. При возникновении события `mousedown` мы вычисляем их, вычитая значение `offsetLeft` из свойства `clientX` источника и значение `offsetTop` из свойства `clientY`. Это дает нам значения, которые нужно вычесть из координат элемента при обработке события `mousemove`. В результате перетаскивание выполняется более плавно, что улучшает впечатления пользователя.

Добавление пользовательских событий

Функциональность перетаскивания нельзя нормально использовать, если вы не знаете, когда оно выполняется. Пока что мы не в состоянии выяснить, когда перетаскивание началось, выполняется или завершилось. К счастью, это можно сделать с помощью пользовательских событий, чтобы другие части приложения могли использовать перетаскивание.

Поскольку объект `DragDrop` является одиночкой и использует паттерн Модуль, в него нужно внести некоторые изменения, чтобы задействовать тип `EventTarget`. В новом коде мы создаем объект `EventTarget`, добавляем методы `enable()` и `disable()` и, наконец, возвращаем объект `EventTarget`:

Листинг DragAndDropExample04.htm


```
var DragDrop = function(){

    var dragdrop = new EventTarget(),
        dragging = null,
        diffX = 0,
        diffY = 0;

    function handleEvent(event){

        // получение объектов event и target
        event = EventUtil.getEvent(event);
        var target = EventUtil.getTarget(event);

        // определение типа события
        switch(event.type){
            case "mousedown":
                if (target.className.indexOf("draggable") > -1){
                    dragging = target;
                    diffX = event.clientX - target.offsetLeft;
                    diffY = event.clientY - target.offsetTop;
                    dragdrop.fire({type:"dragstart", target: dragging,
                                   x: event.clientX, y: event.clientY});
                }
                break;

            case "mousemove":
                if (dragging !== null){

                    // назначение расположения
                    dragging.style.left = (event.clientX - diffX) + "px";
```

```
                dragging.style.top = (event.clientY - diffY) + "px";

                // генерирование пользовательского события
                dragdrop.fire({type:"drag", target: dragging,
                               x: event.clientX, y: event.clientY});
            }
            break;

        case "mouseup":
            dragdrop.fire({type:"dragend", target: dragging,
                           x: event.clientX, y: event.clientY});
            dragging = null;
            break;
    }
};

// общедоступный интерфейс
dragdrop.enable = function(){
    EventUtil.addHandler(document, "mousedown", handleEvent);
    EventUtil.addHandler(document, "mousemove", handleEvent);
    EventUtil.addHandler(document, "mouseup", handleEvent);
};

dragdrop.disable = function(){
    EventUtil.removeHandler(document, "mousedown", handleEvent);
    EventUtil.removeHandler(document, "mousemove", handleEvent);
    EventUtil.removeHandler(document, "mouseup", handleEvent);
};

return dragdrop;
}();
```

В этом фрагменте мы определяем события `dragstart`, `drag` и `dragend`. Каждое из них задает перетаскиваемый элемент в качестве источника и предоставляет свойства `x` и `y` для указания его текущей позиции. Эти события генерируются для объекта `DragDrop`, который позднее расширяется методами `enable()` и `disable()`. Это небольшое изменение паттерна Модуль обеспечивает поддержку событий объектом `DragDrop`:

Листинг DragAndDropExample04.htm


```
DragDrop.addHandler("dragstart", function(event){
    var status = document.getElementById("status");
    status.innerHTML = "Started dragging " + event.target.id;
});

DragDrop.addHandler("drag", function(event){
    var status = document.getElementById("status");
    status.innerHTML += "<br>Dragged " + event.target.id +
        " перетащен в точку (" + event.x + "," + event.y + ")";
});

DragDrop.addHandler("dragend", function(event){
    var status = document.getElementById("status");
    status.innerHTML += "<br>Dropped " + event.target.id +
        " at (" + event.x + "," + event.y + ")";
});
```

Здесь мы назначаем обработчики всех событий объекта `DragDrop` и отображаем на странице текущее состояние и расположение перетаскиваемого элемента. Как только элемент отпущен, вы можете просмотреть список всех промежуточных действий с момента начала перетаскивания.

Добавление пользовательских событий к объекту `DragDrop` делает его более надежным и позволяет использовать его в веб-приложениях для управления сложным функционалом перетаскивания.

Резюме

JavaScript-функции очень эффективны, потому что являются объектами первого класса. Замыкания и переключение контекста функций поддерживают целый ряд мощных возможностей программирования.

- Вы можете создавать конструкторы, безопасные для областей видимости. Это гарантирует, что конструктор, вызванный без оператора `new`, не изменит неправильный объект контекста.
- Отложенный вызов функции позволяет отсрочить ветвление кода до ее первого вызова.
- Связывание функций позволяет создавать функции, которые всегда выполняются в специфическом контексте, а благодаря каррированию можно создавать функции с предварительно заданными аргументами.
- Объединив связывание и каррирование, можно выполнить любую функцию в любом контексте с любыми аргументами.

ECMAScript 5 поддерживает несколько способов создания объектов, защищенных от изменения.

- Нерасширяемые объекты невозможно расширить новыми свойствами или методами.
- Запечатанные объекты являются нерасширяемыми и не позволяют удалять имеющиеся свойства и методы.
- Замороженные объекты являются запечатанными и не поддерживают перезапись членов.

Используя методы `setTimeout()` и `setInterval()`, можно создавать таймеры.

- Код таймера находится в состоянии предварительного ожидания, пока не наступает нужный момент времени, после чего код добавляется в очередь JavaScript-процесса, чтобы быть выполненным, когда процесс перейдет в состояние бездействия.
- Каждый раз, когда фрагмент кода выполняется полностью, наступает краткий период бездействия, позволяющий браузеру запустить другие процессы.

- Таймеры могут разделять длительные сценарии на небольшие блоки, выполняемые по очереди через определенные интервалы времени. Это позволяет ускорить отклик веб-приложений на действия пользователей.

В JavaScript часто используется паттерн Наблюдатель, реализованный в форме событий. Хотя события часто работают с DOM, вы можете реализовать в собственном коде пользовательские события. Они помогают ослабить связи между разными частями кода, что упрощает его сопровождение и снижает вероятность ошибок при его изменении.

В пользовательском интерфейсе приложений для настольных компьютеров и веб-приложений часто применяется перетаскивание, позволяющее пользователям легко и интуитивно понятно перемещать или конфигурировать элементы. Перетаскивание можно реализовать в JavaScript с помощью событий мыши и некоторых несложных вычислений. Объединив перетаскивание с пользовательскими событиями, можно создать удобную платформу, полезную в самых разных ситуациях.

23 Автономный режим и клиентское хранилище

- Распознавание автономного режима
- Использование автономного кэша
- Хранение данных в браузере

При разработке HTML5 особое внимание было уделено поддержке *автономных* веб-приложений, то есть веб-приложений, способных работать даже без подключения к Интернету. Оно и понятно: всем веб-разработчикам хотелось бы, чтобы их приложения могли на равных конкурировать с традиционными клиентскими приложениями, которые работают, пока включено электропитание.

Реализация поддержки автономного режима в веб-приложениях требует решения нескольких задач. Прежде всего, приложение должно знать, доступно ли подключение к Интернету. Затем приложение должно получить доступ к множеству ресурсов (изображения, код JavaScript, CSS и т. д.), необходимых для нормальной работы. Наконец, приложению нужна область для локального хранения данных, доступная для чтения и записи независимо от наличия или отсутствия подключения к Интернету. Благодаря HTML5 и родственным JavaScript API автономные веб-приложения стали реальностью.

Распознавание автономного режима

Чтобы приложение могло узнать, что устройство работает в автономном режиме, HTML5 предоставляет свойство `navigator.onLine`, которое имеет значение `true` при наличии подключения к Интернету и `false` в противном случае. Идея в том, что браузер определяет, доступна ли сеть, и возвращает соответствующий индикатор. На практике свойство `navigator.onLine` имеет некоторые странности:

- Internet Explorer 6+ и Safari 5+ правильно определяют разрыв соединения с сетью, присваивая свойству `navigator.onLine` значение `false`;
- Firefox 3+ и Opera 10.6+ поддерживают свойство `navigator.onLine`, но пользователь должен переводить браузер в автономный режим вручную, выбирая в меню File пункт Work Offline;
- в Chrome до версии 11 включительно свойство `navigator.onLine` всегда имеет значение `true` (это ошибка, требующая исправления).

Учитывая эти проблемы с совместимостью, свойство `navigator.onLine` не может служить признаком подключения к сети. Тем не менее оно полезно, если во время обработки запросов возникают ошибки. Проверить состояние подключения можно следующим образом:

Листинг OnLineExample01.htm


```
if (navigator.onLine){
    // обычный план действий
} else {
    // действия, специфичные для автономного режима
}
```

Кроме того, в HTML5 доступны события `online` и `offline`, помогающие следить за доступностью сети. Эти события генерируются для объекта `window` соответственно при подключении к сети и отключении от нее:

Листинг OnlineEventsExample01.htm

```
EventUtil.addHandler(window, "online", function(){
    alert("Online");  // Режим подключения
});
EventUtil.addHandler(window, "offline", function(){
    alert("Offline"); // Автономный режим
});
```

Распознавание автономного режима лучше начать с проверки свойства `navigator.onLine` для определения первоначального состояния при загрузке страницы. После этого изменения состояния подключения лучше всего регистрировать с помощью событий. Свойство `navigator.onLine` также изменяется при генерировании событий, но чтобы с его помощью распознавать изменения состояния подключения, его нужно проверять вручную.

Распознавание автономного режима поддерживается в Internet Explorer 6+ (только свойство `navigator.onLine`), Firefox 3, Safari 4, Opera 10.6, Chrome, Safari для iOS 3.2 и WebKit для Android.

Кэш приложений

Кэш приложений (application cache, или appcache) в HTML5 разработан специально для автономных веб-приложений. Он представляет собой область кэша, отделенную от обычного кэша браузера. Чтобы указать, что нужно сохранить в кэше приложений страницы, следует предоставить *файл манифеста* (manifest file), указав в нем ресурсы, которые требуется загрузить и кэшировать. Вот простой файл манифеста:

```
CACHE MANIFEST
#Comment

file.js
file.css
```

В простейшей форме файл манифеста указывает ресурсы, которые нужно загрузить из сети, чтобы они были доступны в автономном режиме.

Возможности настройки этого файла очень широки, но мы не будем их обсуждать. Подробные сведения об этом можно найти, перейдя по ссылке http://html5doctor.com/go-offline-with-application-cache/.

Чтобы связать файл манифеста со страницей, нужно указать путь к нему с помощью атрибута `manifest` элемента `<html>`, например:

```
<html manifest="/offline.manifest">
```

Этот код указывает файл манифеста `/offline.manifest`. Чтобы файл можно было использовать, он должен иметь тип контента `text/cache-manifest`.

Хотя кэш приложений предназначен в основном для кэширования ресурсов, которые должны быть доступны в автономном режиме, он предоставляет API для проверки состояния кэша. Главный объект этого API называется `applicationCache`. Он содержит свойство `status`, которое указывает текущее состояние кэша приложений:

- 0 — со страницей не связан кэш приложений;
- 1 — кэш приложений не обновляется (состояние бездействия);
- 2 — файл манифеста кэша приложений загружается и проверяется на предмет обновлений;
- 3 — кэш приложений загружает из сети ресурсы, указанные в файле манифеста;

- 4 — кэш приложений был обновлен, все ресурсы загружены и могут быть использованы с помощью метода `swapCache()`;
- 5 — файл манифеста кэша приложений больше недоступен, поэтому кэш приложений для страницы недействителен.

Об изменениях состояния кэша приложений сигнализируют следующие события:

- `checking` — генерируется, когда браузер начинает искать обновление для кэша приложений;
- `error` — генерируется, если при проверке или загрузке ресурсов возникает ошибка;
- `noupdate` — генерируется после проверки манифеста кэша приложений на предмет изменений;
- `downloading` — генерируется, когда начинается загрузка ресурсов в кэш приложений;
- `progress` — многократно генерируется в ходе загрузки файлов в кэш приложений;
- `updateready` — генерируется, когда новая версия кэша приложений страницы загружена и может быть использована с помощью метода `swapCache()`;
- `cached` — генерируется, когда кэш приложений заполнен и готов к использованию.

При загрузке страницы эти события обычно генерируются в указанном порядке. Вы также можете запустить проверку обновлений для кэша приложений, вызвав метод `update()`:

```
applicationCache.update();
```

При вызове метода `update()` кэш приложений проверяет, обновлен ли файл манифеста (при этом генерируется событие `checking`), а затем продолжает работу, как если бы страница уже была загружена. Если генерируется событие `cached`, это означает, что новое содержимое кэша приложений готово к использованию и другие действия не требуются. Если генерируется событие `updateready`, это означает, что доступна новая версия кэша приложений, которую нужно подключить методом `swapCache()`:

```
EventUtil.addHandler(applicationCache, "updateready", function(){
    applicationCache.swapCache();
});
```

Кэш приложений HTML5 поддерживается в Firefox 3+, Safari 4+, Opera 10.6, Chrome, Safari для iOS 3.2+ и WebKit для Android. В Firefox до версии 4 включительно при вызове метода `swapCache()` возникает ошибка.

Хранилище данных

С появлением веб-приложений потребовалось хранить пользовательскую информацию непосредственно на клиентском компьютере. Это логично: информация, относящаяся к конкретному пользователю, должна находиться на его компьютере, будь то сведения для входа в систему, предпочтения или другие данные. Соответственно, поставщики веб-приложений начали искать способы сохранения данных на стороне клиента. Первое решение проблемы поступило от корпорации Netscape, которая предложила использовать для этого cookie-файлы, описанные в спецификации *Persistent Client State: HTTP Cookies* («Хранение состояния клиента: HTTP-cookie»), доступной по ссылке http://curl.haxx.se/rfc/cookie_spec.html. Сегодня cookie-файлы — это лишь одна из технологий сохранения данных на стороне клиента.

Cookie-файлы

Cookie-файлы изначально были предназначены для сохранения сведений о сеансе на клиентском компьютере. Спецификация предписывала серверу отправлять в любом ответе на HTTP-запрос HTTP-заголовок `Set-Cookie` со сведениями о сеансе. Например, заголовки ответа сервера могут быть такими:

```
HTTP/1.1 200 OK
Content-type: text/html
Set-Cookie: name=значение
Other-header: значение_другого_заголовка
```

Этот HTTP-ответ задает cookie-файл с именем `"name"` и значением `"значение"`. И имя, и значение при отправке кодируются в формате URL-адреса. Браузер сохраняет эти сведения о сеансе и при каждом последующем запросе отправляет их серверу в HTTP-заголовке `Cookie`, например:

```
GET /index.html HTTP/1.1
Cookie: name=значение
Other-header: значение_другого_заголовка
```

Эти дополнительные сведения сервер может использовать для идентификации клиента, который отправил запрос.

Ограничения

Cookie-файлы по природе связаны с конкретным доменом. Заданный cookie-файл прилагается к запросам, отправляемым в тот домен, в котором файл был создан. Это ограничение гарантирует, что информация, сохраненная в cookie-файле, будет доступна только одобренным получателям и не сможет попасть в другие домены.

Поскольку cookie-файлы хранятся на клиентском компьютере, к ним применяются ограничения, гарантирующие, что их невозможно будет использовать со злым

умыслом и что они не будут занимать много места на диске. Общее количество cookie-файлов на домен ограничено и зависит от браузера:

- в Internet Explorer 6 и более ранних версий поддерживаются 20 cookie-файлов на домен;
- в Internet Explorer 7 и более поздних версий поддерживаются 50 cookie-файлов на домен (первоначально поддерживалось не более 20 cookie-файлов на домен, но позднее в исправлении от Microsoft это число было увеличено);
- Firefox поддерживает 50 cookie-файлов на домен;
- Opera поддерживает 30 cookie-файлов на домен;
- в Safari и Chrome количество cookie-файлов не ограничено.

При создании cookie-файлов сверх этих ограничений браузер начинает удалять имеющиеся cookie-файлы. Internet Explorer и Opera начинают с удаления cookie-файла, который не использовался дольше всего. Firefox, по всей видимости, выбирает удаляемый cookie-файл случайно, поэтому во избежание непредвиденных последствий важно соблюдать ограничения на количество cookie-файлов.

Размер cookie-файла в браузерах также ограничен. В большинстве браузеров это ограничение составляет около 4096 байт, но ради совместимости кода с разными браузерами лучше ограничивать размер cookie-файла значением в 4095 байт или менее. Это ограничение применяется ко всем cookie-файлам в домене, а не к отдельным.

Если попытаться создать cookie-файл с размером, превышающим ограничение, запрос будет проигнорирован. Имейте в виду, что один знак обычно занимает один байт, если не используются многобайтовые знаки.

Части cookie-файла

Cookie-файлы состоят из перечисленных далее частей, хранящихся в браузере.

- **Имя** — уникальное имя, идентифицирующее cookie-файл. Имена cookie нечувствительны к регистру, например имена `myCookie` и `MyCookie` считаются одинаковыми. Тем не менее на практике лучше предполагать, что имена cookie чувствительны к регистру, потому что так их обрабатывают некоторые серверные программы. Имена cookie-файлов нужно кодировать в формате URL-адресов.
- **Значение** — строковое значение, хранящееся в cookie-файле. Его необходимо кодировать в формате URL-адресов.
- **Домен** — домен, для которого действителен cookie-файл. Все запросы, отправляемые в этот домен, будут содержать данные из cookie. Это значение может включать поддомен (например, `www.wrox.com`) или исключать его (например, значение `.wrox.com` действительно для всех поддоменов `wrox.com`). Если домен не задан явно, предполагается тот домен, в котором был задан cookie-файл.

- **Путь** — путь в указанном домене, для которого cookie-файл следует отправить серверу. Например, вы можете указать, что cookie-файл доступен только по адресу `http://www.wrox.com/books/`, и тогда страницы на сайте `http://www.wrox.com` не будут отправлять cookie несмотря на то, что запрос исходит из того же домена.
- **Дата истечения срока действия** — метка времени, указывающая, когда cookie-файл должен быть удален (то есть когда следует прекратить отправлять его серверу). По умолчанию все cookie-файлы удаляются при завершении сеанса браузера, но можно задать и другое время удаления. Это значение задается как дата в формате GMT (Нед, ДД-Мес-ГГГГ ЧЧ:ММ:СС GMT) и указывает точное время, когда cookie-файл должен быть удален. Таким образом, cookie-файл может оставаться на компьютере пользователя даже после закрытия браузера. Если задать уже прошедшую дату, cookie-файл будет удален незамедлительно.
- **Флаг безопасности** — если этот флаг указан, информация из cookie отправляется серверу, только если используется SSL-соединение. Например, при запросе `https://www.wrox.com` она отправляется, а при запросе `http://www.wrox.com` — нет.

Каждый из этих элементов данных указывается в составе заголовка `Set-Cookie`, при этом в качестве разделителя используется сочетание точки с запятой и пробела, например:

```
HTTP/1.1 200 OK
Content-type: text/html
Set-Cookie: name=значение; expires=Mon, 22-Jan-07 07:10:24 GMT;
domain=.wrox.com
Other-header: значение_другого_заголовка
```

Этот заголовок определяет cookie-файл с именем `"name"`, который должен устареть 22 января 2007 года в 7:10:24 по GMT и действителен для домена `www.wrox.com` и любых других поддоменов `wrox.com`, таких как `p2p.wrox.com`.

Флаг безопасности — единственный элемент cookie-файла, который не является парой из имени и значения, это просто слово `"secure"`. Рассмотрим пример:

```
HTTP/1.1 200 OK
Content-type: text/html
Set-Cookie: name=значение; domain=.wrox.com; path=/; secure
Other-header: значение_другого_заголовка
```

В этом фрагменте создается cookie-файл, действительный для всех поддоменов `wrox.com` и всех страниц в этом домене (что указано с помощью аргумента `path`). Поскольку задан флаг `secure`, этот cookie-файл может передаваться только по SSL-соединению.

Имейте в виду, что домен, путь, дата истечения срока действия и флаг безопасности лишь указывают браузеру, когда следует отправлять cookie-файл вместе с запросом. Эти аргументы не отправляются серверу с информацией cookie-файла; отправляются только пары имен и значений.

Cookie-файлы в JavaScript

Работать с cookie-файлами в JavaScript не просто из-за на редкость неудачного интерфейса — BOM-свойства `document.cookie`. Оно уникально тем, что работает совершенно по-разному в зависимости от того, как используется. При чтении свойство `document.cookie` возвращает строку, содержащую все cookie-файлы, доступные странице (на основе домена, пути, даты истечения срока действия и параметров безопасности). Эта строка представляет собой набор пар имен и значений, разделенных точками с запятой, например:

```
имя1=значение1;имя2=значение2;имя3=значение3
```

Все имена и значения кодируются в формате URL-адресов, поэтому их нужно декодировать с помощью метода `decodeURIComponent()`.

При записи можно присвоить свойству `document.cookie` новую cookie-строку, которая будет интерпретирована и добавлена в существующий набор cookie-файлов. При установке свойства `document.cookie` никакие cookie-файлы не перезаписываются, если задаваемое имя cookie-файла не используется. Для задания cookie-файла используется такой же формат, что и в заголовке `Set-Cookie`:

```
name=значение; expires=срок_действия; path=путь_домена;
domain=имя_домена; secure
```

Из этих параметров обязательны только имя и значение cookie-файла. Вот простой пример:

```
document.cookie = "name=Nicholas";
```

Этот код создает cookie-файл сеанса с именем `"name"` и значением `"Nicholas"`. Этот cookie-файл отправляется серверу при каждом клиентском запросе и удаляется при закрытии браузера. Хотя такой способ вполне приемлем, потому что никакие знаки в имени или значении кодировать не нужно, при задании cookie-файла лучше всегда использовать метод `encodeURIComponent()`, например:

```
document.cookie = encodeURIComponent("name") + "=" +
                  encodeURIComponent("Nicholas");
```

Чтобы указать дополнительные сведения о создаваемом cookie-файле, просто добавьте их к строке в том же формате, что и в заголовке `Set-Cookie`, например:

```
document.cookie = encodeURIComponent("name") + "=" +
                  encodeURIComponent("Nicholas") + "; domain=.wrox.com;
                  path=/";
```

Поскольку читать и записывать cookie-файлы в JavaScript непросто, для этого часто используют вспомогательные функции. Базовых операций над cookie-файлами три: чтение, запись и удаление. Они представлены в объекте `CookieUtil` следующим образом:

Листинг CookieUtil.js


```
var CookieUtil = {

    get: function (name){
        var cookieName = encodeURIComponent(name) + "=",
            cookieStart = document.cookie.indexOf(cookieName),
            cookieValue = null;

        if (cookieStart > -1){
            var cookieEnd = document.cookie.indexOf(";", cookieStart);
            if (cookieEnd == -1){
                cookieEnd = document.cookie.length;
            }
            cookieValue = decodeURIComponent(document.cookie.substring(
                cookieStart + cookieName.length, cookieEnd));
        }

        return cookieValue;
    },

    set: function (name, value, expires, path, domain, secure) {
        var cookieText = encodeURIComponent(name) + "=" +
                         encodeURIComponent(value);

        if (expires instanceof Date) {
            cookieText += "; expires=" + expires.toGMTString();
        }

        if (path) {
            cookieText += "; path=" + path;
        }

        if (domain) {
            cookieText += "; domain=" + domain;
        }

        if (secure) {
            cookieText += "; secure";
        }

        document.cookie = cookieText;
    },

    unset: function (name, path, domain, secure){
        this.set(name, "", new Date(0), path, domain, secure);
    }

};
```

Метод `CookieUtil.get()` получает значение cookie-файла с указанным именем. Для этого он ищет в свойстве `document.cookie` имя cookie-файла, за которым следует знак равенства. Если ему удается найти этот шаблон, вызывается метод `indexOf()` для поиска ближайшей точки с запятой (которая обозначает конец cookie). Если точка с запятой не обнаруживается, это означает, что cookie является последним

в строке и всю оставшуюся строку следует считать значением cookie. Это значение декодируется с помощью метода `decodeURIComponent()` и возвращается. Если обнаружить cookie не удается, возвращается значение `null`.

Метод `CookieUtil.set()` задает cookie для страницы. В качестве аргументов он принимает имя cookie, значение cookie, необязательный объект `Date`, указывающий, когда нужно удалить cookie, необязательный URL-путь для cookie, необязательный домен cookie и необязательное логическое значение, указывающее, следует ли добавить флаг `secure`. Аргументы указываются в том порядке, в котором они используются чаще всего, при этом обязательны только первые два. Внутри этого метода имя и значение cookie кодируются в формате URL-адресов с помощью метода `encodeURIComponent()`, после чего проверяются другие параметры. Если аргументом `expires` является объект `Date`, к строке добавляется параметр `expires`, при этом для правильного форматирования даты вызывается метод `toGMTString()` объекта `Date`. В оставшейся части метода мы просто составляем строку cookie и присваиваем ее свойству `document.cookie`.

Непосредственного способа удалить существующий cookie-файл не существует. Вместо этого нужно еще раз задать cookie с теми же путем, доменом и параметрами безопасности и назначить в качестве даты истечения его срока действия какой-то момент в прошлом. Это и делает метод `CookieUtil.unset()`. Он принимает четыре аргумента: имя удаляемого cookie, необязательный путь, необязательный домен и необязательный аргумент безопасности.

Эти аргументы передаются в метод `CookieUtil.set()` вместе с пустой строкой и датой истечения срока действия, равной 1 января 1970 года (объект `Date` инициализируется нулевым значением счетчика миллисекунд). В результате cookie-файл удаляется.

Эти методы можно использовать следующим образом:

Листинг CookieExample01.htm


```
// задание cookie-файлов
CookieUtil.set("name", "Nicholas");
CookieUtil.set("book", "Professional JavaScript");

// чтение значений
alert(CookieUtil.get("name"));    // "Nicholas"
alert(CookieUtil.get("book"));    // "Professional JavaScript"

// удаление cookie-файлов
CookieUtil.unset("name");
CookieUtil.unset("book");

// задание cookie-файла с путем, доменом и датой истечения срока действия
CookieUtil.set("name", "Nicholas", "/books/projs/", "www.wrox.com",
               new Date("January 1, 2010"));

// удаление того же cookie-файла
CookieUtil.unset("name", "/books/projs/", "www.wrox.com");
```

```
// задание защищенного cookie-файла
CookieUtil.set("name", "Nicholas", null, null, null, true);
```

Эти методы упрощают использование cookie-файлов для хранения данных на клиенте, обеспечивая синтаксический анализ и конструирование строки cookie.

Вложенные cookie-файлы

Чтобы обойти действующее во многих браузерах ограничение на количество cookie-файлов на домен, некоторые разработчики используют *вложенные cookie* (subcookies). Так называют меньшие фрагменты данных, хранящиеся в одном cookie. Идея в том, чтобы хранить в одном cookie несколько пар имен и значений. Наиболее популярный формат вложенных cookie таков:

```
name=имя1=значение1&имя2=значение2&имя3=значение3&имя4=значение4
```

Вложенные cookie обычно форматируют как строку запроса. Затем эти значения можно сохранить в единственном cookie, а не использовать отдельный cookie для хранения каждой пары имени и значения. Благодаря этому разработчики веб-сайтов и веб-приложений могут хранить больше структурированных данных, не достигая предельного количества cookie на домен.

Для работы с вложенными cookie нужен другой набор методов. Синтаксический анализ и сериализация вложенных cookie выполняются иначе и более сложны из-за предполагаемого способа использования таких cookie. Например, чтобы получить вложенный cookie, требуется выполнить те же базовые действия, но перед декодированием значения нужно найти информацию вложенного cookie:

Листинг SubCookieUtil.js


```
var SubCookieUtil = {

    get: function (name, subName){
        var subCookies = this.getAll(name);
        if (subCookies){
            return subCookies[subName];
        } else {
            return null;
        }
    },

    getAll: function(name){
        var cookieName = encodeURIComponent(name) + "=",
            cookieStart = document.cookie.indexOf(cookieName),
            cookieValue = null,
            cookieEnd,
            subCookies,
            i,
            parts,
            result = {};
```

```
        if (cookieStart > -1){
            cookieEnd = document.cookie.indexOf(";", cookieStart);
            if (cookieEnd == -1){
                cookieEnd = document.cookie.length;
            }
            cookieValue = document.cookie.substring(cookieStart +
                          cookieName.length, cookieEnd);

            if (cookieValue.length > 0){
                subCookies = cookieValue.split("&");

                for (i=0, len=subCookies.length; i < len; i++){
                    parts = subCookies[i].split("=");
                    result[decodeURIComponent(parts[0])] =
                         decodeURIComponent(parts[1]);
                }

                return result;
            }
        }

        return null;
    },

    // другой код
};
```

Для получения вложенных cookie в этом коде используются методы `get()` и `getAll()`. Первый из них получает значение одного вложенного cookie, тогда как второй получает все вложенные cookie, возвращая их в виде объекта, свойствами которого являются имена cookie, а значениями — значения cookie. Метод `get()` принимает два аргумента: имя cookie и имя вложенного cookie. Он просто вызывает метод `getAll()` для получения всех вложенных cookie, а затем возвращает только нужный cookie (или `null`, если cookie не существует).

Метод `SubCookieUtil.getAll()` очень похож на `CookieUtil.get()` тем, как он разбирает значение cookie. Его отличие состоит в том, что значение cookie не декодируется немедленно. Вместо этого оно разделяется по знаку амперсанда для сохранения всех вложенных cookie в массиве. Затем каждый вложенный cookie разделяется по знаку равенства, чтобы первым элементом в массиве `parts` было имя вложенного cookie, а вторым — его значение. Оба элемента декодируются с помощью метода `decodeURIComponent()` и назначаются объекту `result`, который возвращается как значение метода. Если cookie-файл не существует, возвращается значение `null`.

Эти методы можно использовать следующим образом:

Листинг SubCookiesExample01.htm


```
// предполагается, что
// document.cookie=data=name=Nicholas&book=Professional%20JavaScript

// получение всех вложенных cookie-фрагментов
```

```
var data = SubCookieUtil.getAll("data");
alert(data.name);    // "Nicholas"
alert(data.book);    // "Professional JavaScript"

// индивидуальное получение вложенных cookie-фрагментов
alert(SubCookieUtil.get("data", "name"));  // "Nicholas"
alert(SubCookieUtil.get("data", "book"));  // "Professional JavaScript"
```

Для записи вложенных cookie-фрагментов хорошо подходят методы `set()` и `setAll()`:

Листинг SubCookieUtil.js

```
var SubCookieUtil = {

    set: function (name, subName, value, expires, path, domain, secure) {
        var subcookies = this.getAll(name) || {};
        subcookies[subName] = value;
        this.setAll(name, subcookies, expires, path, domain, secure);
    },

    setAll: function(name, subcookies, expires, path, domain, secure){

        var cookieText = encodeURIComponent(name) + "=",
            subcookieParts = new Array(),
            subName;

        for (subName in subcookies){
            if (subName.length > 0 && subcookies.hasOwnProperty(subName)){
                subcookieParts.push(encodeURIComponent(subName) + "=" +
                    encodeURIComponent(subcookies[subName]));
            }
        }

        if (cookieParts.length > 0){
            cookieText += subcookieParts.join("&");

            if (expires instanceof Date) {
                cookieText += "; expires=" + expires.toGMTString();
            }

            if (path) {
                cookieText += "; path=" + path;
            }

            if (domain) {
                cookieText += "; domain=" + domain;
            }

            if (secure) {
                cookieText += "; secure";
            }
        } else {
            cookieText += "; expires=" + (new Date(0)).toGMTString();
        }

        document.cookie = cookieText;
```

```
    },

    // другой код
};
```

Метод `set()` принимает семь аргументов: имя cookie, имя вложенного cookie, значение вложенного cookie, необязательный объект `Date`, содержащий дату и время истечения срока действия cookie, необязательный путь для cookie, необязательный домен cookie и необязательный логический флаг безопасности. Все необязательные аргументы относятся к самому cookie, а не к вложенному cookie. Чтобы можно было сохранить несколько вложенных cookie в одном cookie, их путь, домен и флаг безопасности должны быть одинаковыми. Дата истечения срока действия относится ко всему cookie и может быть задана при записи отдельного вложенного cookie. В теле метода мы первым делом получаем все вложенные cookie для указанного имени cookie. Если метод `getAll()` возвращает `null`, переменной `subcookies` с помощью логического оператора ИЛИ назначается новый объект. После этого объекту `subcookies` присваивается значение вложенного cookie и этот объект передается в метод `setAll()`.

Метод `setAll()` принимает шесть аргументов: имя cookie, объект, содержащий все вложенные cookie, и остальные необязательные аргументы, используемые в методе `set()`. В теле метода свойства второго аргумента перебираются в цикле `for-in`. Для сохранения нужных данных служит метод `hasOwnProperty()`, который гарантирует, что во вложенных cookie будут сериализованы только свойства экземпляра. Поскольку именем свойства может быть пустая строка, перед добавлением значения к результату также проверяется длина имени свойства. Все пары из имен и значений вложенных cookie добавляются в массив `subcookieParts`, чтобы позднее их можно было легко объединить с помощью амперсанда, используя метод `join()`. Остальной код не отличается от метода `CookieUtil.set()`.

Эти методы можно использовать следующим образом:

Листинг SubCookiesExample01.htm


```
//предполагается, что
// document.cookie=data=name=Nicholas&book=Professional%20JavaScript

// задание двух вложенных cookie
SubCookieUtil.set("data", "name", "Nicholas");
SubCookieUtil.set("data", "book", "Professional JavaScript");

// задание всех вложенных cookie с датой истечения срока действия
SubCookieUtil.setAll("data", { name: "Nicholas",
    book: "Professional JavaScript" }, new Date("January 1, 2010"));

// изменение значения name и даты истечения срока действия cookie
SubCookieUtil.set("data", "name", "Майкл", new Date("February 1, 2010"));
```

Нам осталось обсудить методы удаления вложенных cookie. Для удаления обычного cookie достаточно задать уже прошедшую дату истечения срока действия, но

вложенный cookie удалить сложнее. Для этого нужно получить все вложенные cookie, содержащиеся в обычном cookie, удалить тот из них, который не нужен, а затем сохранить оставшиеся значения вложенных cookie. Рассмотрим следующий код:

Листинг SubCookieUtil.js

```
var SubCookieUtil = {

    // другой код

    unset: function (name, subName, path, domain, secure){
        var subcookies = this.getAll(name);
        if (subcookies){
            delete subcookies[subName];
            this.setAll(name, subcookies, null, path, domain, secure);
        }
    },

    unsetAll: function(name, path, domain, secure){
        this.setAll(name, null, new Date(0), path, domain, secure);
    }

};
```

Метод `unset()` из этого фрагмента удаляет один вложенный cookie, оставляя остальное содержимое неизменным, а метод `unsetAll()` эквивалентен методу `CookieUtil.unset()`, который удаляет cookie полностью. Как и в методах `set()` и `setAll()`, путь, домен и флаг безопасности должны соответствовать параметрам, с которыми cookie был создан. Эти методы можно использовать следующим образом:

```
// удаление вложенного cookie с именем "name"
SubCookieUtil.unset("data", "name");

// удаление всего cookie
SubCookieUtil.unsetAll("data");
```

Если вас беспокоит ограничение количества cookie на домен, вложенные cookie могут оказаться привлекательной альтернативой, но вам придется более тщательно отслеживать размеры cookie-файлов, чтобы не превзойти предельный размер отдельного cookie.

Замечания по поводу cookie-файлов

Существуют также *cookie-файлы, используемые только в HTTP* (HTTP-only cookie). Такие cookie можно задать и в браузере, и с сервера, а прочитать — только с сервера, потому что получить их значения с помощью JavaScript нельзя.

Поскольку все cookie отправляются браузером как заголовки запроса, хранение большого объема информации в cookie может сказаться на скорости обработки запросов, адресованных в конкретный домен. Чем больше данных содержит cookie, тем дольше обрабатывается запрос. Несмотря на то что браузер налагает ограничения

на размер cookie, во избежание проблем с быстродействием имеет смысл хранить в cookie как можно меньше информации.

Из-за целого ряда ограничений cookie-файлы плохо подходят для хранения больших объемов данных. К счастью, существуют более эффективные альтернативы.

Хранить в cookie важные или конфиденциальные данные вроде номеров кредитных карт или личных адресов не рекомендуется, потому что содержимое cookie может быть доступно другим пользователям.

Пользовательские данные в Internet Explorer

В Internet Explorer 5 корпорация Microsoft реализовала возможность хранения пользовательских данных, основанную на поведении. Вы можете хранить до 128 Кбайт данных на документ и до 1 Мбайт данных на домен. Чтобы использовать эту возможность, нужно сначала задать для элемента с помощью CSS поведение `userData`:

```
<div style="behavior:url(#default#userData)" id="dataStore"></div>
```

Как только для элемента задано поведение `userData`, вы можете сохранить в нем данные с помощью метода `setAttribute()`. Чтобы зафиксировать данные в кэше браузера, необходимо вызвать метод `save()`, передав ему имя хранилища для сохранения данных. Это имя, которое может быть любым, используется для различения наборов данных. Рассмотрим пример:

Листинг UserDataExample01.htm


```
var dataStore = document.getElementById("dataStore");
dataStore.setAttribute("name", "Nicholas");
dataStore.setAttribute("book", "Professional JavaScript");
dataStore.save("BookInfo");
```

Этот код сохраняет в элементе `<div>` два блока данных. После сохранения данных с помощью метода `setAttribute()` здесь вызывается метод `save()` с именем хранилища данных `"BookInfo"`. При загрузке страницы в следующий раз вы сможете получить данные, вызвав метод `load()` с именем хранилища данных:

Листинг UserDataExample01.htm

```
dataStore.load("BookInfo");
alert(dataStore.getAttribute("name"));    // "Nicholas"
alert(dataStore.getAttribute("book"));    // "Professional JavaScript"
```

Метод `load()` получает всю информацию из хранилища данных `"BookInfo"` и делает ее доступной через элемент; пока информация не загружена явно, она недоступна. Если вызвать метод `getAttribute()` для имени, которое не существует или не было загружено, возвращается значение `null`.

Вы можете явно удалить данные из элемента с помощью метода `removeAttribute()`, передав ему имя атрибута. После удаления нужно снова вызвать метод `save()`, чтобы зафиксировать изменения:

Листинг UserDataExample01.htm

```
dataStore.removeAttribute("name");
dataStore.removeAttribute("book");
dataStore.save("BookInfo");
```

Этот код удаляет два атрибута, а затем сохраняет измененные данные в кэше.

Ограничения доступа к пользовательским данным в Internet Explorer похожи на ограничения на работу с cookie. Чтобы можно было получить доступ к хранилищу данных, страница со сценарием должна находиться в том же домене по тому же пути и использовать тот же протокол, что и сценарий, сохранивший данные в хранилище. В отличие от cookie-файлов, вы не можете сделать пользовательские данные доступными более широкой аудитории. Пользовательские данные отличаются от cookie еще и тем, что по умолчанию они сохраняются между сеансами, а срок их действия не истекает, поэтому для освобождения места нужно явно удалить данные методом `removeAttribute()`.

Как и cookie-файлы, пользовательские данные в Internet Explorer не защищены, поэтому их не следует применять для хранения конфиденциальных сведений.

Веб-хранилище

Впервые веб-хранилище было описано в спецификации Web Applications 1.0, подготовленной рабочей группой Web Hypertext Application Technical Working Group (WHAT-WG). Первоначальное описание из этой спецификации позднее было включено в HTML5, а в итоге стало отдельной спецификацией. Веб-хранилище устраняет некоторые ограничения cookie в тех ситуациях, когда данные нужны только на стороне клиента и их не требуется многократно отправлять серверу. Основные цели использования веб-хранилища таковы:

- хранение данных сеанса вне cookie-файлов;
- хранение больших объемов данных между сеансами.

Оригинальная спецификация Web Storage включает определения двух объектов: `localStorage` и `globalStorage`. Они доступны в Internet Explorer 8+, Firefox 3.5+, Safari 3.1+, Chrome 4+ и Opera 10.5+ как свойства объекта `window`.

В Firefox 2 и 3 доступны частичные реализации веб-хранилища, основанные на более раннем проекте, в котором вместо localStorage использовался объект globalStorage.

Тип Storage

Тип `Storage` позволяет хранить пары имен и значений, при этом максимальный объем хранилища определяется браузером. Экземпляр `Storage` работает подобно любым другим объектам и имеет следующие дополнительные методы:

- `clear()` — удаляет все значения (не реализован в Firefox);
- `getItem(имя)` — получает значение, соответствующее указанному имени;
- `key(индекс)` — получает имя значения в указанной позиции;
- `removeItem(имя)` — удаляет пару имени и значения, указанную по имени;
- `setItem(имя, значение)` — задает значение для указанного имени.

Методы `getItem()`, `removeItem()` и `setItem()` можно вызывать напрямую или косвенно, манипулируя объектом `Storage`. Поскольку элементы хранятся в этом объекте как свойства, вы можете читать их и задавать их значения, используя точечную или скобочную нотацию, а также удалять их с помощью оператора `delete`. Тем не менее рекомендуется применять для этого методы, чтобы не перезаписать один из уже доступных членов объекта с ключом.

Узнать количество пар имен и значений в объекте `Storage` можно с помощью свойства `length`. Объем всех данных в объекте определить невозможно, но в Internet Explorer 8 доступно свойство `remainingSpace`, которое возвращает объем доступного в хранилище места в байтах.

Объект Storage может хранить только строки. Нестроковые данные перед сохранением в нем преобразуются в строку.

Объект sessionStorage

Объект `sessionStorage` хранит данные только в течение сеанса, то есть до закрытия браузера (в этом смысле он похож на сеансовый cookie). Данные, сохраненные в объекте `sessionStorage`, остаются в нем при обновлениях страницы и даже могут оставаться нетронутыми при сбое и перезапуске браузера (в Firefox и WebKit, но не в Internet Explorer).

Поскольку объект `sessionStorage` связан с сеансом сервера, он недоступен, если файл используется локально. Данные, сохраненные в объекте `sessionStorage`, доступны только со страницы, которая первоначально поместила их в него, из-за чего в многостраничных приложениях от него мало пользы.

Объект `sessionStorage` является экземпляром типа `Storage`, поэтому вы можете назначить ему данные, вызвав метод `setItem()` или напрямую задав новое свойство. Вот соответствующие примеры:

Листинг SessionStorageExample01.htm

```
// сохранение данных с помощью метода
sessionStorage.setItem("name", "Nicholas");
```

```
// сохранение данных с помощью свойства
sessionStorage.book = "Professional JavaScript";
```

Запись данных в хранилище немного различается в зависимости от браузера. В Firefox и WebKit она осуществляется синхронно, так что данные, добавленные в хранилище, фиксируются сразу. В Internet Explorer данные сохраняются в хранилище асинхронно, то есть после их назначения свойству они могут записываться на диск с некоторой задержкой. При небольших объемах данных это различие несущественно, но при значительных объемах процесс JavaScript возобновляется быстрее в Internet Explorer, чем в других браузерах, потому что фактическая запись на диск делегируется.

В Internet Explorer 8 можно форсировать запись на диск, вызвав метод `begin()` перед назначением любых новых данных свойствам и метод `commit()` — после, например:

```
// только для IE8
sessionStorage.begin();
sessionStorage.name = "Nicholas";
sessionStorage.book = "Professional JavaScript";
sessionStorage.commit();
```

В этом фрагменте значения свойств `name` и `book` записываются, как только вызывается метод `commit()`. Вызов метода `begin()` гарантирует, что во время выполнения этого кода данные не будут записываться на диск. Если данных немного, такой транзакционный подход не требуется, но при записи больших объемов данных он приобретает смысл.

При наличии данных в объекте `sessionStorage` их можно получить, вызвав метод `getItem()` или прочитав свойство напрямую:

Листинг SessionStorageExample01.htm


```
// получение данных с помощью метода
var name = sessionStorage.getItem("name");

// получение данных с помощью свойства
var book = sessionStorage.book;
```

Перебрать значения в объекте `sessionStorage` можно с помощью свойства `length` и метода `key()`:

Листинг SessionStorageExample01.htm

```
for (var i=0, len = sessionStorage.length; i < len; i++){
    var key = sessionStorage.key(i);
    var value = sessionStorage.getItem(key);
    alert(key + "=" + value);
}
```

Для последовательного доступа к парам имен и значений в объекте `sessionStorage` можно сначала получить имя элемента данных в указанной позиции методом

`key()`, а затем вызвать для получения значения метод `getItem()`, передав в него имя элемента.

Вы также можете перебрать значения в объекте `sessionStorage` в цикле `for-in`:

```
for (var key in sessionStorage){
    var value = sessionStorage.getItem(key);
    alert(key + "=" + value);
}
```

На каждой итерации цикла переменной `key` назначается очередное имя из объекта `sessionStorage`, но встроенные методы, как и свойство `length`, не возвращаются.

Удалить данные из объекта `sessionStorage` можно, применив оператор `delete` к соответствующему свойству или вызвав метод `removeItem()`:

Листинг SessionStorageExample01.htm

```
// удаление значения с помощью оператора delete - не работает в WebKit
delete sessionStorage.name;

// удаление значения с помощью метода
sessionStorage.removeItem("book");
```

На день написания этого фрагмента оператор `delete` не удаляет данные в WebKit. Метод `removeItem()` во всех поддерживающих его браузерах работает правильно.

Объект `sessionStorage` рекомендуется использовать для хранения небольших фрагментов данных, действительных только в течение сеанса. Для хранения данных между сеансами лучше подходит объект `globalStorage` или `localStorage`.

Объект globalStorage

Объект `globalStorage` реализован в Firefox 2 и, будучи частью оригинальной спецификации Web Storage, предназначен для хранения данных между сеансами со специфическими ограничениями доступа. Чтобы использовать его, нужно указать домены, для которых данные должны быть доступны. Это делается с помощью свойства и скобочной нотации:

Листинг GlobalStorageExample01.htm


```
// сохранение значения
globalStorage["wrox.com"].name = "Nicholas";

// получение значения
var name = globalStorage["wrox.com"].name;
```

Здесь осуществляется доступ к области хранения для домена `wrox.com`. Сам объект `globalStorage` не является экземпляром `Storage`, но им является выражение `globalStorage["wrox.com"]`, которое можно использовать соответствующим образом.

Эта область хранилища доступна из домена `wrox.com` и всех его поддоменов. Чтобы ограничить доступ конкретным поддоменом, следует явно указать его:

Листинг GlobalStorageExample01.htm

```
// сохранение значения
globalStorage["www.wrox.com"].name = "Nicholas";

// получение значения
var name = globalStorage["www.wrox.com"].name;
```

Указанная здесь область хранилища доступна только со страницы из поддомена `www.wrox.com`, но не из других поддоменов.

Некоторые браузеры поддерживают более общие ограничения доступа, разрешая доступ к данным только для доменов верхнего уровня или позволяя сделать данные доступными глобально, например:

```
// сохранение данных с доступом для всех - НЕ ДЕЛАЙТЕ ТАК!
globalStorage[""].name = "Nicholas";

// сохранение данных с доступом только для доменов,
// заканчивающихся на .net, - НЕ ДЕЛАЙТЕ ТАК!
globalStorage["net"].name = "Nicholas";
```

Во избежание проблем с безопасностью рекомендуется не использовать общедоступные хранилища данных. Кроме того, из соображений безопасности эта возможность может быть удалена или существенно ограничена в будущем, так что применять ее в приложениях не следует. Иначе говоря, при использовании объекта `globalStorage` всегда указывайте имя домена.

Доступ к областям, заданным с помощью объекта `globalStorage`, ограничен доменом, протоколом и портом страницы, которая инициирует запрос. Например, если данные для домена `wrox.com` сохранены с использованием протокола HTTPS, получить их по протоколу HTTP из этого же домена не удастся. Аналогичным образом страница, доступ к которой осуществляется через порт 80, не может иметь общие данные со страницей из того же домена, которая использует тот же протокол, но обращается к порту 8080. Это похоже на политику одного источника в Ajax.

Все свойства объекта `globalStorage` являются экземплярами типа `Storage`, поэтому их можно использовать следующим образом:

Листинг GlobalStorageExample01.htm


```
globalStorage["www.wrox.com"].name = "Nicholas";
globalStorage["www.wrox.com"].book = "Professional JavaScript";

globalStorage["www.wrox.com"].removeItem("name");

var book = globalStorage["www.wrox.com"].getItem("book");
```

Если нужное имя домена неизвестно, может быть безопаснее использовать свойство `location.host`, например:

Листинг GlobalStorageExample01.htm

```
globalStorage[location.host].name = "Nicholas";
var book = globalStorage[location.host].getItem("book");
```

Данные, хранящиеся в свойствах объекта `globalStorage`, остаются на диске, пока не будут удалены с помощью метода `removeItem()` или оператора `delete` или пока пользователь не очистит кэш браузера. Благодаря этому объект `globalStorage` идеально подходит для хранения документов или параметров пользователя на клиентском компьютере.

Объект localStorage

Переработанная спецификация HTML5 предписывает использовать для хранения клиентских данных объект `localStorage` вместо `globalStorage`. В отличие от `globalStorage`, для объекта `localStorage` вы не можете указать никакие правила доступа, потому что они уже заданы. Чтобы можно было получить доступ к тому же объекту `localStorage`, страницы должны быть выданы из того же домена (поддомены не допускаются) с использованием того же протокола и того же порта. По сути, это эквивалентно выражению `globalStorage[location.host]`.

Поскольку объект `localStorage` является экземпляром типа `Storage`, его можно использовать так же, как и объект `sessionStorage`. Вот несколько примеров:

Листинг LocalStorageExample01.htm

```
// сохранение данных с помощью метода
localStorage.setItem("name", "Nicholas");

// сохранение данных с помощью свойства
localStorage.book = "Professional JavaScript";

// получение данных с помощью метода
var name = localStorage.getItem("name");

// получение данных с помощью свойства
var book = localStorage.book;
```

Правила хранения данных в объектах `localStorage` и `globalStorage` одинаковы: данные хранятся, пока не будут явно удалены с помощью JavaScript или пока пользователь не очистит кэш браузера.

Поскольку некоторые браузеры поддерживают только объект `globalStorage`, для написания кроссбраузерного кода можно использовать следующую функцию:

Листинг GlobalAndLocalStorageExample01.htm

```
function getLocalStorage(){
    if (typeof localStorage == "object"){
        return localStorage;
```

```
    } else if (typeof globalStorage == "object"){
        return globalStorage[location.host];
    } else {
        // локальное хранилище недоступно
        throw new Error("Local storage not available.");
    }
}
```

Чтобы определить правильное место для хранения данных, достаточно просто вызвать эту функцию:

Листинг GlobalAndLocalStorageExample01.htm

```
var storage = getLocalStorage();
```

После определения нужного объекта `Storage` вы можете легко сохранять и получать данные с одинаковыми правилами доступа во всех браузерах, которые поддерживают веб-хранилище.

Событие storage

При изменении объекта `Storage` для документа генерируется событие `storage`. Это происходит при задании значений с помощью свойств или метода `setItem()`, при удалении значений с помощью оператора `delete` или метода `removeItem()` и при каждом вызове метода `clear()`. Объект `event` этого события имеет четыре свойства:

- `domain` — домен, для которого было изменено хранилище;
- `key` — заданный или измененный ключ;
- `newValue` — значение, присвоенное ключу, или `null`, если ключ был удален;
- `oldValue` — значение до изменения ключа.

Из этих четырех свойств в Internet Explorer 8 и Firefox реализовано только свойство `domain`. WebKit на день написания этого фрагмента не поддерживает событие `storage`.

Для прослушивания события `storage` можно использовать следующий код:

Листинг StorageEventExample01.htm


```
EventUtil.addHandler(document, "storage", function(event){
    alert("Storage changed for " + event.domain);
});
```

Событие `storage` генерируется при любых изменениях объектов `sessionStorage` и `localStorage`, не делая различий между ними.

Пределы и ограничения

Как и другие решения для хранения данных на стороне клиента, веб-хранилище имеет ограничения, которые специфичны для браузера. Вообще говоря, ограничение

размера клиентских данных задается по отдельности для каждого источника (протокол, домен и порт), так что каждому источнику выделяется фиксированный объем места для хранения данных. Ограничение применяется на основе анализа источника страницы с данными.

Большинство браузеров для настольных компьютеров выделяют в локальном хранилище 5 Мбайт на источник. В Chrome и Safari, а также в Safari для iOS и WebKit для Android это ограничение составляет 2,5 Мбайт.

Ограничения объекта `sessionStorage` зависят от браузера. Во многих браузерах размер данных в `sessionStorage` не ограничивается, но в Chrome, Safari, Safari для iOS и WebKit для Android действует ограничение в 2,5 Мбайт, а в Internet Explorer 8+ и Opera — 5 Мбайт.

Дополнительные сведения об этих ограничениях см. на странице тестирования поддержки веб-хранилища по адресу http://dev-test.nemikor.com/web-storage/support-test/.

IndexedDB

Indexed Database API (IndexedDB) служит для хранения структурированных данных в браузере. Он был разработан как альтернатива Web SQL Database API, который уже признан устаревшим и не рассматривается в этой книге. Целью его разработчиков было создание API, позволяющего легко сохранять и получать JavaScript-объекты и при этом поддерживающего запросы и поиск данных.

IndexedDB почти полностью асинхронен. Большинство операций представляют собой запросы, которые выполняются по прошествии некоторого времени и завершаются успехом или ошибкой. Почти каждая операция IndexedDB требует подключения обработчиков событий `error` и `success`, чтобы можно было определить ее результат.

Когда IndexedDB API будет реализован полностью, его главный (глобальный) объект будет называться `indexedDB`, но пока API еще дорабатывается, поэтому в разных браузерах используются префиксы их производителей. Так, в Internet Explorer 10 этот объект называется `msIndexedDB`, в Firefox 4 — `mozIndexedDB`, а в Chrome — `webkitIndexedDB`. В этом разделе ради ясности используется имя `indexedDB`, так что перед каждым примером может потребоваться добавить следующий код:

Листинг IndexedDBExample01.htm

```
var indexedDB = window.indexedDB || window.msIndexedDB ||
    window.mozIndexedDB || window.webkitIndexedDB;
```

Базы данных

IndexedDB — это база данных, похожая на базы данных, которые вы, вероятно, уже использовали, такие как MySQL или Web SQL Database. Важное отличие IndexedDB состоит в том, что в ней для хранения данных используются не таблицы,

а хранилища объектов. База данных IndexedDB — это просто набор хранилищ объектов, объединенных общим именем.

Чтобы использовать базу данных, нужно сначала открыть ее методом `indexedDB.open()`, передав в него ее имя. Если база данных с этим именем уже существует, она будет открыта; если такой базы данных нет, она будет создана и открыта. Метод `indexDB.open()` возвращает экземпляр `IDBRequest`, к которому можно подключить обработчики событий `error` и `success`. Вот пример:

Листинг IndexedDBExample01.htm


```
var request, database;

request = indexedDB.open("admin");
request.onerror = function(event){
    // при открытии базы данных произошла какая-то ошибка
    alert("Something bad happened while trying to open: " +
          event.target.errorCode);
};
request.onsuccess = function(event){
    database = event.target.result;
};
```

В обоих обработчиках событий свойство `event.target` указывает на переменную `request`, так что они взаимозаменяемы. Если вызывается обработчик события `success`, объект базы данных (`IDBDatabase`) доступен как свойство `event.target.result`, и мы сохраняем его в переменной `database`. С этого момента все запросы базы данных выполняются через переменную `database`. Если происходит ошибка, суть проблемы можно узнать по коду ошибки, сохраненному в свойстве `event.target.errorCode` (приведенные коды ошибок действительны для всех операций):

- `IDBDatabaseException.UNKNOWN_ERR` (1) — ошибка не относится ни к одной из имеющихся категорий;
- `IDBDatabaseException.NON_TRANSIENT_ERR` (2) — операция не разрешена;
- `IDBDatabaseException.NOT_FOUND_ERR` (3) — не удалось найти базу данных, для которой нужно выполнить операцию;
- `IDBDatabaseException.CONSTRAINT_ERR` (4) — нарушено ограничение базы данных;
- `IDBDatabaseException.DATA_ERR` (5) — данные для транзакции не соответствуют требованиям;
- `IDBDatabaseException.NOT_ALLOWED_ERR` (6) — операция не разрешена;
- `IDBDatabaseException.TRANSACTION_INACTIVE_ERR` (7) — предпринята попытка выполнить завершенную транзакцию повторно;
- `IDBDatabaseException.ABORT_ERR` (8) — запрос отменен;
- `IDBDatabaseException.READ_ONLY_ERR` (9) — предпринята попытка записать или иным образом изменить данные в режиме «только для чтения»;

- `IDBDatabaseException.TIMEOUT_ERR` (10) — не удалось завершить операцию в отведенное время;
- `IDBDatabaseException.QUOTA_ERR` (11) — недостаточно места на диске.

По умолчанию у базы данных нет версии, так что имеет смысл сначала задать ее. Для этого вызовите метод `setVersion()`, передав ему версию как строку. В результате будет создан объект запроса, которому нужно назначить обработчики событий:

Листинг IndexedDBExample01.htm


```
if (database.version != "1.0"){
    request = database.setVersion("1.0");
    request.onerror = function(event){
        // при попытке задать версию произошла ошибка
        alert("Something bad happened while trying to set version: " +
              event.target.errorCode);
    };
    request.onsuccess = function(event){
        // инициализация базы данных завершена
        alert("Database initialization complete. Database name: " +
              database.name + "; Version: " + database.version);
    };
} else {
    // база данных уже инициализирована
    alert("Database already initialized. Database name: " +
          database.name + "; Version: " + database.version);
}
```

Здесь предпринимается попытка назначить базе данных версию 1.0. Первая строка проверяет, задана ли уже версия. Если нет, вызывается метод `setVersion()`, который создает запрос на изменение версии. Если запрос выполняется успешно, выводится оповещение о том, что версия изменена. В реальном коде здесь следовало бы настроить хранилища объектов (подробности см. в следующем разделе).

Если база данных имеет версию 1.0, выводится оповещение о том, что она уже инициализирована. Так можно узнать, настроены ли для базы данных, которую вы намереваетесь использовать, подходящие хранилища объектов. В течение жизненного цикла веб-приложения по мере обновления и изменения структур данных версия базы данных может измениться много раз.

Хранилища объектов

Как только подключение к базе данных установлено, можно взаимодействовать с хранилищами объектов. Если версия базы данных отличается от ожидаемой, вероятно, вам придется создать такое хранилище, но перед этим важно подумать о том, какие данные вам нужно в нем хранить.

Предположим, что вам нужно хранить пользовательские записи, содержащие имя пользователя, пароль и т. д. Объект для хранения одной записи может быть таким:

```
var user = {
    username: "007",
    firstName: "James",
    lastName: "Bond",
    password: "foo"
};
```

С одного взгляда на этот объект ясно, что ключом в хранилище следует сделать свойство `username`. Имя пользователя должно быть глобально уникальным, и в большинстве случаев именно его вы будете использовать для доступа к данным. Это важно потому, что ключ нужно указать при создании хранилища объектов. Вот как вы создали бы его для этих пользователей:

Листинг IndexedDBExample02.htm

```
var store = db.createObjectStore("users", { keyPath: "username" });
```

Свойство `keyPath` во втором аргументе указывает имя свойства хранящихся объектов, которое должно использоваться как ключ.

Теперь у нас есть ссылка на хранилище объектов и мы можем заполнить его данными методом `add()` или `put()`. Оба они принимают в качестве аргумента объект, который нужно сохранить, и сохраняют его в хранилище. Разница между методами наблюдается, только если объект с таким же ключом уже есть в хранилище. В этом случае метод `add()` генерирует ошибку, а метод `put()` просто перезаписывает объект. Проще говоря, метод `add()` служит для вставки новых значений, а метод `put()` — для обновления значений. Так, инициализировать хранилище объектов в первый раз можно следующим образом:

Листинг IndexedDBExample02.htm


```
// users - это массив новых пользователей
var i=0,
    len = users.length;

while(i < len){
    store.add(users[i++]);
}
```

При каждом вызове метода `add()` или `put()` создается запрос на обновление хранилища объектов. Если вам нужно убедиться, что запрос выполнен успешно, можно сохранить объект запроса в переменной и назначить обработчики событий `error` и `success`:

```
// users - это массив новых пользователей
var i=0,
    request,
    requests = [],
    len = users.length;

while(i < len){
```

```
    request = store.add(users[i++]);
    request.onerror = function(){
        // обработка ошибки
    };
    request.onsuccess = function(){
        // код, выполняемый при успешном завершении операции
    };
    requests.push(request);
}
```

Как только хранилище объектов создано и заполнено данными, можно приступить к выполнению запросов.

Транзакции

После создания хранилища объектов все последующие операции с ним выполняются как *транзакции*. Чтобы создать транзакцию, нужно вызвать метод `transaction()` для объекта базы данных. Каждый раз, когда требуется прочитать или изменить данные, все изменения следует сгруппировать в транзакцию. Создать простейшую транзакцию можно так:

```
var transaction = db.transaction();
```

Если аргументы не указаны, все хранилища объектов в базе данных доступны только для чтения. Для оптимизации можно указать имена одного или нескольких хранилищ объектов, к которым требуется получить доступ:

```
var transaction = db.transaction("users");
```

Этот код загружает для выполнения транзакции только сведения о хранилище объектов `users`. Если вам нужен доступ к нескольким хранилищам объектов, можно передать в метод `transaction()` массив строк:

```
var transaction = db.transaction(["users", "anotherStore"]);
```

Как уже отмечалось, каждая из этих транзакций получает доступ к базе данных в режиме «только для чтения». Изменить режим доступа можно с помощью второго аргумента — константы `READ_ONLY` (0), `READ_WRITE` (1) или `VERSION_CHANGE` (2), доступной у объекта `IDBTransaction`. Этот объект реализован в Internet Explorer 10+ и Firefox 4+, но в Chrome его аналог называется `webkitIDBTransaction`, поэтому для стандартизации интерфейса необходим следующий код:

Листинг IndexedDBExample03.htm

```
var IDBTransaction = window.IDBTransaction || window.webkitIDBTransaction;
```

После этого можно использовать метод `transaction()` следующим образом:

Листинг IndexedDBExample03.htm

```
var transaction = db.transaction("users", IDBTransaction.READ_WRITE);
```

Эта транзакция может читать хранилище объектов `users` и записывать данные в него.

Когда у вас есть ссылка на транзакцию, доступ к конкретному хранилищу объектов обеспечивается методом `objectStore()`, которому передают имя хранилища. После этого можно использовать методы `add()` и `put()`, как было показано, а также метод `get()` для получения значений, `delete()` для удаления объекта и `clear()` для удаления всех объектов. Методы `get()` и `delete()` принимают в качестве аргумента ключ объекта, и все эти пять методов создают объект запроса, например:

Листинг IndexedDBExample02.htm


```
var request = db.transaction("users").objectStore("users").get("007");
request.onerror = function(event){
    // не удалось получить объект
    alert("Did not get the object!");
};
request.onsuccess = function(event){
    var result = event.target.result;
    alert(result.firstName);    // "James"
};
```

Поскольку в рамках одной транзакции можно выполнить любое количество запросов, у самого объекта транзакции есть обработчики событий `error` и `complete`. Они используются для получения сведений о состоянии транзакции:

```
transaction.onerror = function(event){
    // транзакция была отменена
};

transaction.oncomplete = function(event){
    // транзакция выполнена успешно
};
```

Помните, что объект `event` события `complete` не предоставляет доступ к данным, возвращенным запросом `get()`, так что для таких запросов вам все равно нужно обрабатывать событие `success`.

Запросы с курсорами

С помощью транзакции можно напрямую получить один элемент с известным ключом. Если требуется получить несколько элементов, нужно создать в транзакции *курсор* (cursor) — указатель в наборе результатов. В отличие от обычных запросов баз данных, курсор не собирает весь набор результатов, а указывает на первый результат и не ищет следующий, пока не получит команду это сделать.

Чтобы создать курсор, нужно вызвать метод `openCursor()` для хранилища объектов. Как и другие IndexedDB-операции, этот метод возвращает запрос, так что вы должны назначить обработчики событий `success` и `error`, например:

Листинг IndexedDBExample04.htm

```
var store = db.transaction("users").objectStore("users"),
    request = store.openCursor();

request.onsuccess = function(event){
    // код, выполняемый при успешном завершении операции
};

request.onfailure = function(event){
    // обработка сбоя
};
```

Когда вызывается обработчик события `success`, следующий элемент в хранилище объектов доступен с помощью свойства `event.target.result`, которое содержит экземпляр `IDBCursor` при наличии следующего элемента или значение `null`, если элементов больше нет. Перечислим свойства экземпляра `IDBCursor`:

- `direction` — число, указывающее направление перемещения курсора. По умолчанию это свойство имеет значение `IDBCursor.NEXT` (0), которое предписывает переместиться к следующему элементу. Также можно использовать значение `IDBCursor.NEXT_NO_DUPLICATE` (1) для перемещения к следующему элементу с игнорированием дубликатов, `IDBCursor.PREV` (2) для перемещения к предыдущему элементу и `IDBCursor.PREV_NO_DUPLICATE` (3) для перемещения к предыдущему элементу с игнорированием дубликатов.
- `key` — ключ объекта.
- `value` — фактический объект.
- `primaryKey` — ключ, используемый курсором. Может быть ключом объекта или индексным ключом (см. далее).

Получить информацию об одном результате можно следующим образом:

```
request.onsuccess = function(event){
    var cursor = event.target.result;
    if (cursor){    // проверка обязательна
        console.log("Key: " + cursor.key + "; Value: " +
                    JSON.stringify(cursor.value));
    }
};
```

Значение `cursor.value` в этом примере является объектом, поэтому перед отображением оно кодируется в формате JSON.

С помощью курсора можно обновить отдельную запись. Метод `update()` обновляет текущее значение курсора указанным объектом. Как и другие подобные операции, метод `update()` создает новый запрос, поэтому чтобы узнать результат, нужно назначить обработчики событий `success` и `error`:

```
request.onsuccess = function(event){
    var cursor = event.target.result,
```

```
        value,
        updateRequest;

    if (cursor){    // проверка обязательна
        if (cursor.key == "foo"){
            value = cursor.value;          // получение текущего значения
            value.password = "magic!";     // обновление пароля

            // запрос на сохранение обновления
            updateRequest = cursor.update(value);
            updateRequest.onsuccess = function(){
                // код, выполняемый при успешном завершении операции
            };
            updateRequest.onfailure = function(){
                // обработка сбоя
            };
        }
    }
};
```

Вы также можете удалить элемент в текущей позиции с помощью метода `delete()`. Как и метод `update()`, он тоже создает запрос:

```
request.onsuccess = function(event){
    var cursor = event.target.result,
        value,
        deleteRequest;

    if (cursor){    // проверка обязательна
        if (cursor.key == "foo"){

            // запрос на удаление значения
            deleteRequest = cursor.delete();
            deleteRequest.onsuccess = function(){
                // код, выполняемый при успешном завершении операции
            };
            deleteRequest.onfailure = function(){
                // обработка сбоя
            };
        }
    }
};
```

Если у транзакции нет разрешения на изменение хранилища объектов, методы `update()` и `delete()` генерируют ошибки.

Каждый курсор выполняет по умолчанию только один запрос. Чтобы выполнить другой запрос, необходимо вызвать один из следующих методов:

- `continue(`*ключ*`)` — перемещает курсор к следующему элементу в наборе результатов. Аргумент *ключ* не обязателен. Если он не указан, курсор просто перемещается к следующему элементу; если он указан, курсор перемещается к указанному ключу.

- `advance(`*количество*`)` — перемещает курсор вперед на указанное количество элементов.

Каждый из этих методов предписывает курсору повторно использовать тот же запрос, так что при этом задействуются те же самые обработчики событий `success` и `failure`, пока они не станут ненужными. Например, следующий код перебирает все элементы в хранилище объектов:

```
request.onsuccess = function(event){
    var cursor = event.target.result;
    if (cursor){    // проверка обязательна
        console.log("Key: " + cursor.key + "; Value: " +
                    JSON.stringify(cursor.value));
        cursor.continue();    // переход к следующему значению
    } else {
        console.log("Готово!");
    }
};
```

Вызов `continue()` инициирует другой запрос, в результате которого снова вызывается обработчик события `success`. По исчерпании элементов этот обработчик вызывается в последний раз, при этом свойство `event.target.result` равно `null`.

Диапазоны ключей

Курсоры могут показаться недостаточно эффективными, потому что они ограничивают способы получения данных. Чтобы сделать работу с курсорами немного более управляемой, используют *диапазоны ключей* (key ranges), которые представляются экземплярами типа `IDBKeyRange`. Это стандартное имя типа, поддерживаемое в Internet Explorer 10+ и Firefox 4+, тогда как в Chrome он называется `webkitIDBKeyRange`. Как и при работе с другими типами, связанными с IndexedDB, сначала нужно создать его локальную копию с учетом этого различия:

```
var IDBKeyRange = window.IDBKeyRange || window.webkitIDBKeyRange;
```

Указать диапазон ключей можно четырьмя разными способами. Первый — вызвать метод `only()`, передав ему ключ, который нужно получить:

```
var onlyRange = IDBKeyRange.only("007");
```

В данном случае будет получено только значение с ключом `"007"`. Создание курсора с таким диапазоном аналогично непосредственному доступу к хранилищу объектов и вызову `get("007")`.

Диапазон второго типа определяет нижнюю границу набора результатов, то есть указывает для курсора начальный элемент. Например, если указать следующий диапазон ключей, курсор переберет элементы до конца начиная с ключа `"007"`:

```
// проход к концу начиная с элемента "007"
var lowerRange = IDBKeyRange.lowerBound("007");
```

Если нужно начать с элемента, следующего за значением с ключом "007", можно передать в метод значение true в качестве второго аргумента:

```
// проход к концу начиная после элемента "007"
var lowerRange = IDBKeyRange.lowerBound("007", true);
```

Диапазон третьего типа определяет верхнюю границу набора результатов, то есть ключ конечного значения. Такой диапазон задается с помощью метода upperBound(). В следующем фрагменте курсор остановится, когда дойдет до значения с ключом "ace":

```
// проход с начала до элемента "ace"
var upperRange = IDBKeyRange.upperBound("ace");
```

Если включать указанный ключ в набор результатов не нужно, передайте в метод в качестве второго аргумента значение true:

```
// проход с начала до элемента перед "ace"
var upperRange = IDBKeyRange.upperBound("ace", true);
```

Указать нижнюю и верхнюю границы можно с помощью метода bound(). Он принимает четыре аргумента: ключ нижней границы, ключ верхней границы, необязательное логическое значение, предписывающее пропустить нижнюю границу, и необязательное логическое значение, предписывающее пропустить верхнюю границу. Вот несколько примеров:

```
// проход до элемента "ace" начиная с элемента "007"
var boundRange = IDBKeyRange.bound("007", "ace");

// проход до элемента "ace" начиная с элемента после "007"
var boundRange = IDBKeyRange.bound("007", "ace", true);

// проход до элемента перед "ace" начиная с элемента после "007"
var boundRange = IDBKeyRange.bound("007", "ace", true, true);

// проход до элемента перед "ace" начиная с элемента "007"
var boundRange = IDBKeyRange.bound("007", "ace", false, true);
```

Если затем передать определенный диапазон в метод openCursor(), будет создан курсор, остающийся в заданных пределах:

```
var store = db.transaction("users").objectStore("users"),
    range = IDBKeyRange.bound("007", "ace");
    request = store.openCursor(range);

request.onsuccess = function(event){
    var cursor = event.target.result;
    if (cursor){    // проверка обязательна
        console.log("Key: " + cursor.key + "; Value: " +
```

```
                JSON.stringify(cursor.value));
        cursor.continue();    // переход к следующему элементу
    } else {
        console.log("Готово!");
    }
};
```

Этот код выводит значения между ключами `"007"` и `"ace"`. Очевидно, что их будет меньше, чем в примере из предыдущего раздела.

Указание направления перемещения курсора

На самом деле у метода `openCursor()` два аргумента. Первым является экземпляр `IDBKeyRange`, а вторым — числовая константа, указывающая направление перемещения курсора. Значения этих констант доступны в объекте `IDBCursor`, реализации которого в Firefox 4+ и Chrome различаются, так что первым делом нужно привести их «к общему знаменателю»:

```
var IDBCursor = window.IDBCursor || window.webkitIDBCursor;
```

Обычно курсор начинает обработку с первого элемента в хранилище объектов и продвигается к последнему с каждым вызовом метода `continue()` или `advance()`. По умолчанию такие курсоры используют константу направления `IDBCursor.NEXT`. При наличии дубликатов в хранилище может потребоваться пропускающий их курсор. Его можно реализовать, передав в метод `openCursor()` в качестве второго аргумента значение `IDBCursor.NEXT_NO_DUPLICATE`:

```
var store = db.transaction("users").objectStore("users"),
    request = store.openCursor(null, IDBCursor.NEXT_NO_DUPLICATE);
```

Обратите внимание, что первым аргументом здесь является значение `null`, которое предписывает использовать предлагаемый по умолчанию диапазон ключей, охватывающий все значения. Этот курсор перебирает элементы в хранилище объектов от первого до последнего, пропуская при этом дубликаты.

Вы также можете создать курсор, который перемещается по хранилищу от последнего элемента к первому. Для этого следует передать в метод `openCursor()` значение `IDBCursor.PREV` или `IDBCursor.PREV_NO_DUPLICATE` (последнее указывает пропускать дубликаты):

Листинг IndexedDBExample05.htm

```
var store = db.transaction("users").objectStore("users"),
    request = store.openCursor(null, IDBCursor.PREV);
```

Если курсор открыт с помощью значения `IDBCursor.PREV` или `IDBCursor.PREV_NO_DUPLICATE`, при каждом вызове метода `continue()` или `advance()` курсор будет перемещаться по хранилищу объектов назад, а не вперед.

Индексы

Возможно, в некоторых наборах данных вы захотите указать более одного ключа для хранилища объектов. Например, если вы отслеживаете пользователей по идентификатору и имени, может потребоваться получать доступ к записям с помощью любого из этих двух элементов. Для этого имеет смысл использовать идентификатор пользователя как первичный ключ и создать индекс для имени пользователя.

Чтобы создать индекс, сначала следует получить ссылку на хранилище объектов, а затем вызвать метод `createIndex()`, например:

```
var store = db.transaction("users").objectStore("users"),
    index = store.createIndex("username", "username", { unique: true });
```

Первым аргументом метода `createIndex()` является имя индекса, вторым — имя свойства для индексации, третьим — объект `options`, содержащий ключ `unique`. Этот параметр нужно указывать всегда, чтобы было ясно, уникален ли ключ среди всех записей. Поскольку имя пользователя дублироваться не может, этот индекс уникален.

Метод `createIndex()` возвращает экземпляр `IDBIndex`. Этот же экземпляр можно получить, вызвав для хранилища объектов метод `index()`. Например, использовать уже существующий индекс `"username"` можно так:

```
var store = db.transaction("users").objectStore("users"),
    index = store.index("username");
```

Индекс во многом похож на хранилище объектов. Вы можете создать для индекса новый курсор методом `openCursor()`, который отличается от одноименного метода хранилища объектов лишь тем, что свойству `result.key` присваивается индексный, а не первичный ключ, например:

```
var store = db.transaction("users").objectStore("users"),
    index = store.index("username"),
    request = index.openCursor();

request.onsuccess = function(event){
    // код, выполняемый при успешном завершении операции
};
```

Для индекса также можно создать специальный курсор, возвращающий для каждой записи только первичный ключ, используя метод `openKeyCursor()`, который принимает те же аргументы, что и метод `openCursor()`. Имейте в виду, что свойство `event.result.key` представляет индексный ключ, а `event.result.value` — первичный ключ, а не всю запись.

```
var store = db.transaction("users").objectStore("users"),
    index = store.index("username"),
    request = index.openKeyCursor();
```

```
request.onsuccess = function(event){
    // код, выполняемый при успешном завершении операции
    // event.result.key - индексный ключ
    // event.result.value - первичный ключ
};
```

Вы также можете получить из индекса одно значение, вызвав метод `get()` и передав в него индексный ключ; в результате будет создан новый запрос:

```
var store = db.transaction("users").objectStore("users"),
    index = store.index("username"),
    request = index.get("007");

request.onsuccess = function(event){
    // код, выполняемый при успешном завершении операции
};
request.onfailure = function(event){
    // обработка сбоя
};
```

Чтобы получить только первичный ключ для указанного индексного ключа, используйте метод `getKey()`. Он также создает новый запрос, но свойство `result.value` при этом содержит первичный ключ, а не всю запись:

```
var store = db.transaction("users").objectStore("users"),
    index = store.index("username"),
    request = index.getKey("007");

request.onsuccess = function(event){
    // код, выполняемый при успешном завершении операции
    // event.result.key - индексный ключ
    // event.result.value - первичный ключ
};
```

В обработчике события `success` в этом примере значением `event.result.value` был бы идентификатор пользователя.

Информацию об индексе можно получить в любой момент с помощью свойств объекта `IDBIndex`:

- `name` — имя индекса;
- `keyPath` — пусть к свойству, переданный в метод `createIndex()`;
- `objectStore` — хранилище объектов, с которым работает данный индекс;
- `unique` — логическое значение, указывающее, уникален ли индексный ключ.

Само хранилище объектов также отслеживает индексы по имени с помощью свойства `indexNames`. Это позволяет легко узнать, какие индексы уже есть у объекта:

```
var store = db.transaction("users").objectStore("users"),
    indexNames = store.indexNames,
```

```
    index,
    i = 0,
    len = indexNames.length;

while(i < len){
    index = store.index(indexNames[i++]);
    console.log("Index name: " + index.name + "; KeyPath: " +
        index.keyPath + "; Unique: " + index.unique);
}
```

Этот код перебирает все индексы и выводит информацию о них на консоль.

Индекс можно удалить, вызвав для хранилища объектов метод `deleteIndex()` с именем индекса:

```
var store = db.transaction("users").objectStore("users");
store.deleteIndex("username");
```

Поскольку удаление индекса не влияет на данные в хранилище объектов, эта операция выполняется без каких-либо функций обратного вызова.

Проблемы параллельного доступа

Хотя IndexedDB является асинхронным API, ему все равно присущи проблемы параллельного доступа. Если одна веб-страница одновременно открыта в двух разных вкладках браузера, какой-то из ее экземпляров может попытаться обновить базу данных, когда второй экземпляр не готов к этому. Проблема связана с заданием новой версии базы данных, так что метод `setVersion()` может быть выполнен, только если базу данных использует лишь одна вкладка браузера.

При открытии базы данных в первый раз важно назначить обработчик события `versionchange`. Эта функция обратного вызова выполняется, когда метод `setVersion()` вызывает другая вкладка из того же источника. В ответ на это событие лучше всего немедленно закрыть базу данных, чтобы можно было обновить ее версию, например:

```
var request, database;

request = indexedDB.open("admin");
request.onsuccess = function(event){
    database = event.target.result;

    database.onversionchange = function(){
        database.close();
    };
};
```

Обработчик события `versionchange` следует назначать после каждого успешного открытия базы данных.

При вызове метода `setVersion()` также важно назначить запросу обработчик события `blocked`. Он вызывается, когда вы пытаетесь обновить версию базы данных,

в то время как она открыта в другой вкладке. В этом случае имеет смысл попросить пользователя закрыть все другие вкладки, прежде чем снова вызывать метод `setVersion()`, например:

```
var request = database.setVersion("2.0");
request.onblocked = function(){
    // закройте все остальные вкладки и повторите попытку
    alert("Please close all other tabs and try again.");
};

request.onsuccess = function(){
    // код, выполняемый при успешном завершении операции; продолжение работы
};
```

Помните, что обработчик события `versionchange` вызывается и для других вкладок.

Если всегда назначать эти обработчики событий, веб-приложение сможет эффективнее обрабатывать проблемы параллельного доступа, связанные с IndexedDB.

Пределы и ограничения

У IndexedDB много тех же ограничений, что и у веб-хранилища. Так, базы данных IndexedDB привязаны к источнику (протокол, домен и порт) страницы, из-за чего информация не может быть общей для нескольких доменов. Это означает, например, что с поддоменами `www.wrox.com` и `p2p.wrox.com` используются разные хранилища данных.

Объем хранимых данных на источник также ограничен. В Firefox 4+ это ограничение сейчас составляет 50 Мбайт, а в Chrome — 5 Мбайт. Firefox для мобильных устройств ограничивает объем данных значением 5 Мбайт, а при превышении этой квоты запрашивает у пользователя разрешение на сохранение дополнительных данных.

Кроме того, в Firefox запрещен доступ к базам данных IndexedDB для локальных файлов, тогда как в Chrome такого ограничения нет. Таким образом, для запуска примеров из этой книги на локальном компьютере используйте Chrome.

Резюме

Автономные веб-приложения и клиентские хранилища данных — две важные для будущего Сети технологии. Распознав переход приложения в автономный режим, браузер может генерировать события, чтобы приложение могло выполнить необходимые действия. Указать файлы, которые должны быть доступны в автономном режиме, можно с помощью кэша приложений. Для определения состояния кэша приложений и регистрации его изменений можно использовать соответствующий JavaScript API.

Перечислим некоторые аспекты клиентского хранилища:

- Раньше для хранения информации на клиентском компьютере можно было использовать только cookie-файлы — небольшие фрагменты данных, задаваемые на клиенте или сервере и передаваемые с каждым запросом.
- В JavaScript для доступа к cookie используется свойство `document.cookie`.
- Из-за ряда ограничений cookie-файлы подходят для хранения только сравнительно небольших объемов данных.

Internet Explorer поддерживает поведение `userData`, которое можно применить к элементу на странице:

- После применения поведения `userData` элемент может загрузить данные из именованного хранилища. Для работы с ними используются методы `getAttribute()`, `setAttribute()` и `removeAttribute()`.
- Чтобы данные не были удалены при завершении сеанса, их нужно явно сохранить в именованном хранилище данных с помощью метода `save()`.

Также можно сохранять данные в веб-хранилище, используя объекты `sessionStorage` и `localStorage`. Первый из них хранит данные в течение сеанса браузера, по завершении которого данные удаляются. Второй используется для сохранения данных между сеансами и соблюдает политики безопасности обмена данными между доменами.

IndexedDB — это механизм хранения структурированной информации. Она напоминает базу данных SQL-типа, но в IndexedDB данные хранятся не в таблицах, а в хранилищах объектов. Чтобы создать такое хранилище, нужно сначала определить ключ, а уже затем можно добавлять данные. Для запроса конкретных элементов данных из хранилищ объектов используются курсоры, а для ускорения просмотра конкретных свойств можно создавать индексы.

Таким образом, с помощью JavaScript можно хранить на клиентском компьютере значительный объем данных. Будьте, однако, внимательны, чтобы не сохранить по неосторожности конфиденциальную информацию, потому что кэш данных не шифруется.

24 Наилучшие методики

- Удобство сопровождения кода
- Повышение быстродействия кода
- Развертывание кода

Начиная с 2000 года разработка веб-приложений развивалась стремительными темпами. Область, которая была виртуальным Диким Западом и в которой было приемлемо практически все, стала полноценной дисциплиной с общепризнанными оптимальными методиками программирования, основанными на надежных исследованиях. По мере трансформации простых веб-сайтов в сложные веб-приложения и превращения энтузиастов в высокооплачиваемых профессионалов копилось множество данных о том, какие приемы и подходы работают, а какие нет. Исследования и гипотезы очень позитивно сказались на развитии JavaScript, и теперь нам известно множество наилучших методик программирования веб-приложений.

Удобство сопровождения кода

На ранних веб-сайтах JavaScript-код использовался преимущественно для создания несложных эффектов и проверки форм. Современные веб-приложения содержат тысячи строк JavaScript-кода, управляющего разнообразными сложными процессами. Это требует, чтобы разработчики при написании кода учитывали удобство его сопровождения в будущем. Они приносят прибыль своим компаниям и делают это, не просто создавая программные продукты вовремя, но и наращивая объемы интеллектуальной собственности, которая продолжает приносить прибыль длительное время после выпуска продукта.

Писать удобный для сопровождения код очень важно, потому что большинство разработчиков проводят много времени, изучая и модернизируя чужой код. Новые приложения редко разрабатывают с нуля — гораздо чаще за основу берутся уже имеющиеся компоненты. Удобство сопровождения кода помогает другим разработчикам делать свою работу как можно лучше.

Концепция удобства сопровождения кода не уникальна для JavaScript. Многие ее аспекты относятся ко всем языкам программирования, хотя есть и специфичные для JavaScript.

Какой код удобно сопровождать?

Удобство сопровождения кода включает несколько аспектов. Говорят, что код удобно сопровождать, если он:

- **понятный** — цель написания кода и использованный при этом подход можно понять без пояснений автора кода;
- **интуитивный** — смысл кода ясен независимо от его сложности;
- **адаптируемый** — при изменении данных не требуется переписывать значительный объем кода;
- **расширяемый** — архитектура кода допускает расширение его функционала в будущем;
- **удобен для отладки** — при возникновении проблемы код позволяет достаточно точно определить ее источник.

Умение писать удобный для сопровождения JavaScript-код — важный навык, который среди прочих отличает профессионального веб-разработчика, по-настоящему владеющего своим мастерством, от энтузиаста, способного «состряпать» за выходные худо-бедно работающий сайт.

Конвенции кодирования

Один из простейших способов начать писать удобный для сопровождения JavaScript-код — это принять те или иные конвенции. Конвенции кодирования доступны для большинства языков программирования, и в Интернете можно легко найти тысячи документов по этой теме. Профессиональные организации уже давно предлагают разработчикам использовать конвенции кодирования, чтобы упростить сопровождение кода. Лучшие проекты с открытым исходным кодом включают строгие требования к его написанию, помогающие всем участникам сообщества понимать организацию кода.

Конвенции кодирования особо важны для JavaScript из-за того, что он настолько гибок. В отличие от большинства объектно-ориентированных языков, JavaScript

не вынуждает разработчиков определять все как объекты и поддерживает много стилей программирования: от традиционной объектно-ориентированной парадигмы до декларативных и функциональных подходов. Быстро просмотрев несколько JavaScript-библиотек с открытым исходным кодом, можно легко обнаружить разнообразие подходов к созданию объектов, определению методов и управлению средой.

В последующих разделах описаны общие принципы разработки конвенций кодирования. Они очень важны сами по себе, хотя конкретные способы их реализации могут зависеть от специфических требований.

Удобочитаемость кода

Чтобы код было удобно сопровождать, он должен быть удобочитаемым, а это напрямую зависит от форматирования кода в текстовом файле, в том числе от отступов. Если все разработчики используют одну схему отступов во всем проекте, читать код гораздо проще. Отступы обычно создают с помощью пробелов, а не знака табуляции, который отображается в текстовых редакторах по-разному. Популярный размер отступа — четыре пробела, хотя при желании можно уменьшить его или увеличить.

Кроме того, удобочитаемость кода зависит от комментариев. В большинстве языков программирования принято комментировать каждый метод, но из-за того, что JavaScript позволяет создавать функции в любом месте кода, этим правилом часто пренебрегают. Но это лишь подчеркивает важность документирования каждой JavaScript-функции. Перечислим элементы кода, которые следует комментировать:

- **Функции и методы** — все функции и методы должны включать комментарии, описывающие их назначение и, возможно, алгоритмы решения задач. Также важно указывать сделанные предположения, смысл аргументов и описывать возвращаемое функцией значение, если оно имеется (поскольку это нельзя выяснить по определению функции).
- **Большие фрагменты кода** — перед многострочными фрагментами кода, которые используются для решения конкретных задач, следует добавлять комментарии с описанием этих задач.
- **Сложные алгоритмы** — если вы используете уникальный подход к решению проблемы, объясните в комментарии, в чем он заключается. Это не только поможет другим людям, которые будут читать ваш код, но и упростит жизнь вам самим.
- **Трюки и хитрости** — из-за различий браузеров код JavaScript обычно использует некоторые хитрости. Не думайте, что кто-то другой, кто будет читать код, поймет, для чего они используются. Если вы решаете какую-то задачу нетрадиционным образом, потому что один из браузеров не поддерживает обычный подход, укажите это в комментарии. Благодаря этому другие разработчики не будут пытаться «исправить» ваш код, внося в него ошибки, которые вы уже исправили.

Отступы и комментарии делают код более удобочитаемым, упрощая его дальнейшее сопровождение.

Имена переменных и функций

Простота понимания и сопровождения кода во многом зависит от грамотного выбора имен переменных и функций. Многие JavaScript-разработчики поначалу воспринимали программирование как хобби, поэтому по привычке иногда используют бессмысленные имена переменных вроде `"foo"` или `"bar"` и имена функций вроде `"doSomething"`. Если вы желаете заниматься разработкой профессионально, вы должны отказаться от таких привычек, иначе сопровождать ваш код будет гораздо труднее. Рассмотрим общие правила выбора имен:

- В качестве имен переменных следует использовать существительные, такие как `"car"` или `"person"`.
- Имена функций следует начинать с глагола, например `getName()`. Имена функций, которые возвращают логические значения, обычно начинают словом `is`, например `isEnabled()`.
- Следует использовать логичные имена переменных и функций, не беспокоясь об их длине. Длину имен можно сократить путем постобработки и сжатия (см. далее).

Не используйте имена, которые ничего не говорят о данных, содержащихся в переменных. Если имена подобраны удачно, код читается как рассказ и понятен даже при беглом просмотре.

Прозрачность типов переменных

Поскольку JavaScript-переменные типизированы слабо, легко забыть, какие данные должна содержать переменная. Правильный подбор имен отчасти сглаживает остроту этой проблемы, но иногда этого недостаточно. Указать тип данных переменной можно тремя способами.

Первый способ основан на инициализации. Его суть в том, что при определении переменной вы инициализируете ее значением, указывающим, как она будет использоваться в будущем. Например, переменную, которая будет содержать логическое значение, следует инициализировать значением `true` или `false`, а числовую переменную — числом:

```
// указание типов переменных путем инициализации
var found = false;        // логическое значение
var count = -1;           // число
var name = "";            // строка
var person = null;        // объект
```

Инициализация переменной конкретным значением ясно указывает на ее тип. Недостаток этого подхода в том, что его нельзя использовать с аргументами в объявлениях функций.

Второй способ указать тип переменной — это использовать венгерскую нотацию. В этом случае для указания типа переменной добавляют к ее имени один или несколько символов. Эта нотация часто используется в языках сценариев и какое-то время была популярна в JavaScript. Наиболее традиционный вариант венгерской нотации для JavaScript предполагает добавление одного символа для базовых типов данных, например: `"o"` — для объектов; `"s"` — для строк; `"i"` — для целых чисел; `"f"` — для чисел с плавающей точкой; `"b"` — для логических значений:

```
// указание типов переменных с помощью венгерской нотации
var bFound;               // логическое значение
var iCount;               // целое число
var sName;                // строка
var oPerson;              // объект
```

Венгерская нотация в JavaScript привлекательна тем, что ее можно использовать с аргументами функций. Ее недостаток в том, что она затрудняет чтение кода, нарушая его интуитивное восприятие цельными блоками. По этой причине некоторые разработчики отказались от венгерской нотации.

Последний способ указания типов переменных — использование комментариев типов. Такие комментарии добавляются сразу после имен переменных, но перед их инициализацией, например:

```
// указание типов переменных с помощью комментариев
var found    /*:логическое значение*/   = false;
var count    /*:целое число*/           = 10;
var name     /*:строка*/                = "Nicholas";
var person   /*:объект*/                = null;
```

Комментарии типов добавляют в код сведения о типах переменных, но при этом сохраняют общую удобочитаемость кода. Недостаток таких комментариев в том, что они мешают комментировать крупные блоки кода с помощью многострочных комментариев, потому что комментарии типов тоже являются многострочными:

```
// следующий подход не сработает
/*
var found    /*:логическое значение*/   = false;
var count    /*:целое число*/           = 10;
var name     /*:строка*/                = "Nicholas";
var person   /*:объект*/                = null;
*/
```

Здесь мы хотим закомментировать все переменные с помощью многострочного комментария, но комментарии типов препятствуют этому, потому что первое сочетание символов `/*` во второй строке сопоставляется с первым сочетанием `*/` в третьей строке, что приводит к синтаксической ошибке. Если нужно закомментировать строки кода с комментариями типов, лучше использовать однострочные комментарии в каждой строке (многие редакторы поддерживают такую возможность).

Таковы наиболее популярные способы, позволяющие указать типы данных переменных. Каждый из них имеет преимущества и недостатки. Выберите способ, наиболее подходящий для проекта, и используйте его согласованным образом.

Слабая связанность

Если части приложения чересчур зависят одна от другой, код становится сильно связанным и сложным для сопровождения. Типичный пример этого имеет место, если объекты напрямую ссылаются друг на друга так, что изменение одного всегда требует изменения другого. Сильно связанные программы трудно сопровождать, их фрагменты часто приходится переписывать.

Сильная связанность в веб-приложениях формируется в результате некомпетентного использования соответствующих технологий. Важно знать о том, когда и как это происходит, и стараться поддерживать слабую связанность кода.

Ослабление связанности HTML и JavaScript

Один из наиболее частых вариантов связанности имеет место между кодом HTML и JavaScript. В веб-приложениях эти технологии работают на разных уровнях: HTML служит для представления данных, а JavaScript — для реализации поведения. Есть несколько способов объединения этих технологий, но некоторые из них связывают HTML и JavaScript слишком сильно.

Сильная связанность наблюдается, если JavaScript-код встраивается в HTML с помощью элемента `<script>` или если обработчики событий назначаются HTML-атрибутам, например:

```
<!-- использование тега <script> - сильная связанность HTML и JavaScript -->
<script type="text/javascript">
  document.write("Hello world!");
</script>

<!-- назначение обработчика события атрибуту -
     сильная связанность HTML и JavaScript -->
<input type="button" value="Click Me" onclick="doSomething()" />
```

Хотя оба эти примера технически корректны, на практике они сильно связывают HTML-код, представляющий данные, с JavaScript-кодом, который определяет поведение. В идеале HTML и JavaScript должны быть полностью разделены; JavaScript-код следует включать из внешних файлов и назначать поведение с помощью DOM.

Если HTML и JavaScript связаны слишком сильно, при возникновении ошибки приходится сначала определять, где она произошла — в HTML-коде или в JavaScript-файле. Кроме того, сильная связанность может привносить в приложения ошибки новых типов, касающиеся доступности кода. В приведенном примере пользователь

может щелкнуть на кнопке, когда функция `doSomething()` еще не стала доступной, что приведет к ошибке. Это также затрудняет сопровождение кода, потому что для любой модификации поведения кнопки требуется изменять не только JavaScript-сценарий, но и HTML-код.

HTML и JavaScript могут быть слишком сильно связаны и в противоположной ситуации: если HTML-код содержится в JavaScript-сценарии. Обычно это имеет место при использовании свойства `innerHTML` для вставки блока HTML-кода в страницу:

```
// сильная связанность HTML и JavaScript
function insertMessage(msg){
    var container = document.getElementById("container");
    container.innerHTML = "<div class=\"msg\"><p class=\"post\">" + msg +
        "</p>" + "<p><em>Latest message above.</em></p></div>";
}
```

Вообще говоря, в JavaScript-сценариях не следует использовать много HTML-кода. Это помогает поддерживать разделение уровней приложения и упрощает поиск причин ошибок. В приведенном фрагменте при неправильном форматировании динамически создаваемого HTML-кода может возникнуть проблема с макетом страницы. Однако найти причину ошибки будет непросто, потому что при просмотре кода страницы динамически генерируемого HTML-кода вы не увидите. На сильную связанность указывает еще и то, что для модификации данных или макета потребуется внести изменения в JavaScript-код.

Генерирование HTML-кода также по возможности следует отделять от JavaScript-сценариев. Если JavaScript-код используется для вставки данных, желательно, чтобы он не добавлял еще и разметку. Как правило, разметку можно добавить и скрыть при генерировании всей страницы, а позднее просто показать ее средствами JavaScript. Другой подход — получить нужный дополнительный HTML-код с помощью Ajax-запроса; это позволяет использовать тот же уровень генерирования HTML-кода (PHP, JSP, Ruby и т. д.) для вывода разметки, а не встраивать ее в JavaScript-сценарий.

Разделение HTML и JavaScript позволяет сэкономить время на отладке, облегчая поиск причин ошибок, а также упрощает сопровождение кода, потому что изменение поведения ограничено файлами JavaScript, а изменение разметки — файлами генерирования HTML-кода.

Ослабление связанности CSS и JavaScript

Каскадные таблицы стилей (Cascading Style Sheets, CSS) определяют вид страницы. JavaScript и CSS служат для модификации HTML-страниц и часто используются вместе, а потому также иногда могут быть связаны чересчур сильно. Чаще всего это имеет место, когда JavaScript-код предназначен для изменения отдельных стилей, например:

```
// сильная связанность CSS и JavaScript
element.style.color = "red";
element.style.backgroundColor = "blue";
```

Поскольку вид страницы определяется таблицами CSS, в идеале для решения любой проблемы с видом должно быть достаточно просмотреть CSS-файлы. Однако если для изменения отдельных стилей, например цвета, используется JavaScript, это требует проверки и, возможно, изменения дополнительного фрагмента. В результате код JavaScript, который частично отвечает за отображение страницы, оказывается связан с CSS сильнее, чем хотелось бы. Если в будущем придется изменить стили такой страницы, может потребоваться изменить не только CSS-файл, но и JavaScript-код, что затрудняет сопровождение приложения. Стили и поведение следует разделять более четко.

В современных приложениях JavaScript-код часто используется для изменения стилей, так что полностью разделить его и CSS невозможно, но все же их связанность можно ослабить. Для этого следует динамически изменять классы, а не отдельные стили, например:

```
// слабая связанность CSS и JavaScript
element.className = "edit";
```

Изменяя только CSS-класс элемента, вы оставляете большинство данных о стилях строго в CSS. Даже если для изменения класса используется JavaScript-код, это не влияет на стиль элемента напрямую. Если к элементам применены правильные классы, причины любых проблем с видом будут относиться только к CSS, а не к коду JavaScript.

Сильная связанность второго типа встречается только в Internet Explorer (но не в Internet Explorer 8+ в стандартном режиме), где возможно встраивание кода JavaScript в CSS с помощью выражений, например:

```
/* сильная связанность JavaScript и CSS */
div {
    width: expression(document.body.offsetWidth - 10 + "px");
}
```

Как правило, подобные выражения не используют, так как они не обладают кроссбраузерной совместимостью, но их следует избегать еще и потому, что они сильно связывают JavaScript и CSS. Из-за таких выражений ошибки JavaScript могут проявлять себя в CSS. Разработчики, которым приходилось искать такие ошибки, знают, сколько времени можно потратить, прежде чем в голову придет мысль заглянуть в CSS-код.

Сказанное еще раз подчеркивает важность грамотного разделения уровней приложения. Единственным источником проблем с видом страницы должен быть CSS-код, а единственным источником ошибок поведения — JavaScript-код. Поддержание

слабой связанности между разными технологиями упрощает сопровождение всего приложения.

Ослабление связанности логики приложения и обработчиков событий

Веб-приложения обычно содержат множество обработчиков различных событий, но логика приложения редко отделяется от обработчиков. Рассмотрим пример:

```
function handleKeyPress(event){
    event = EventUtil.getEvent(event);
    if (event.keyCode == 13){
        var target = EventUtil.getTarget(event);
        var value = 5 * parseInt(target.value);
        if (value > 10){
            document.getElementById("error-msg").style.display = "block";
        }
    }
}
```

Эта функция не только обрабатывает событие, но и содержит логику приложения. Проблем с этим подходом две. Во-первых, логика запускается только с помощью события, что затрудняет отладку. Что, если предполагаемый результат не получен? Означает ли это, что обработчик не был вызван или логика приложения содержит дефект? Во-вторых, если последующее событие должно запускать такой же код, вам придется дублировать функционал или выносить его в отдельную функцию. В обоих случаях потребуется внести больше изменений, чем необходимо.

Вместо этого лучше отделить логику приложения от обработчиков событий, чтобы каждый блок решал конкретную задачу. Обработчик события должен получать нужные данные из объекта `event` и передавать их в некоторый метод, содержащий логику приложения. Например, предыдущий код можно переписать следующим образом:

```
function validateValue(value){
    value = 5 * parseInt(value);
    if (value > 10){
        document.getElementById("error-msg").style.display = "block";
    }
}

function handleKeyPress(event){
    event = EventUtil.getEvent(event);
    if (event.keyCode == 13){
        var target = EventUtil.getTarget(event);
        validateValue(target.value);
    }
}
```

В этом коде логика и обработчик события разделены. Функция `handleKeyPress()` проверяет, нажата ли клавиша Enter (значение `event.keyCode` равно 13), а затем получает целевой элемент события и передает свойство `value` в функцию `validateValue()`, которая содержит логику приложения. В ней нет ничего, что как-либо зависит от логики обработчика события; она просто получает значение и в зависимости от него определяет, что делать дальше.

Отделение логики приложения от обработчиков событий обеспечивает несколько преимуществ. Во-первых, это позволяет легко менять события, запускающие те или иные процессы. Например, если первоначально обработчик запускался щелчком кнопки мыши, его можно с легкостью заменить нажатием клавиши. Во-вторых, вы можете тестировать код, не подключая обработчики событий, что упрощает создание модульных тестов и автоматизацию процессов в приложении.

Вот несколько принципов ослабления связанности логики приложения и бизнес-логики:

- в методы следует передавать не сам объект `event`, а только нужные данные из него;
- приложение должно поддерживать выполнение каждого действия без вызова обработчика события;
- обработчики событий должны обрабатывать событие, а затем передавать управление логике приложения.

Соблюдение этих принципов значительно упрощает сопровождение любого кода, помогая при этом повысить эффективность тестирования и дальнейшей разработки.

Принципы программирования

Конечно, простота сопровождения JavaScript-кода зависит не только от его оформления, но и от того, что он, собственно, делает. Над корпоративными веб-приложениями обычно работают многие люди, и в таких условиях очень важно унифицировать используемую всеми среду браузера с помощью каких-то неизменных правил. Для этого разработчики должны придерживаться определенных принципов программирования.

Соблюдайте права владения объектами

Динамическая природа JavaScript позволяет в любой момент времени изменить практически любое значение. Говорят, что в JavaScript «нет ничего святого», потому что никакое значение нельзя сделать окончательным или постоянным. Хотя с добавлением в ECMAScript5 объектов, защищенных от изменений (см. главу 22), это уже не так, все равно по умолчанию вы можете изменять любые объекты. В других языках объекты и классы неизменны, если они не содержат исходного кода, но в JavaScript любой объект можно изменить когда угодно, что иногда дает

непредвиденные результаты. Поскольку язык предоставляет такие широкие возможности, важно, чтобы разработчики сами соблюдали некоторые ограничения.

Возможно, наиболее важным принципом программирования в корпоративной среде является соблюдение прав владения объектами. Это означает, что вы не должны изменять объекты, которые вам не принадлежат. Проще говоря, если вы не отвечаете за создание или обслуживание объекта, его конструктора или методов, то изменять его вы тоже не должны. Или, если точнее:

- не добавляйте свойства в экземпляры или прототипы;
- не добавляйте методы в экземпляры или прототипы;
- не переопределяйте существующие методы.

Проблема в том, что разработчики имеют определенные ожидания в отношении того, как должна работать текущая среда браузера, из-за чего изменение объектов, используемых несколькими людьми, часто приводит к ошибкам. Если кто-то исходит из того, что функция `stopEvent()` отменяет для события поведение, предлагаемое по умолчанию, а вы изменяете ее, подключая в ней другие обработчики, это наверняка повлечет за собой проблемы. Другие разработчики, не подозревающие о новых возможностях функции, будут использовать ее согласно старой спецификации, что может нарушить работу приложения.

Эти правила относятся не только к пользовательским, но и к встроенным типам и объектам, таким как `Object`, `String`, `document`, `window` и т. д. В этом случае потенциальные проблемы еще опаснее, потому что производители браузеров могут изменять эти объекты по своему усмотрению без предварительного уведомления.

В качестве примера можно привести историю с популярной библиотекой Prototype. У объекта `document` в ней был реализован метод `getElementsByClassName()`, который возвращал экземпляр `Array`, расширенный методом `each()`. Проблема возникла, когда в браузерах был реализован встроенный метод `getElementsByClassName()`, возвращающий не `Array`, а объект `NodeList`, у которого нет метода `each()`, — о событиях, приведших к проблеме, можно прочитать в блоге Джона Резига (John Resig) по адресу http://ejohn.org/blog/getelementsbyclassname-pre-prototype-16/. Разработчики, использующие библиотеку Prototype, привыкли писать подобный код:

```
document.getElementsByClassName("selected").each(Element.hide);
```

Хотя этот код нормально работал в браузерах, не имеющих встроенного метода `getElementsByClassName()`, в обновленных браузерах с этим методом он стал вызывать ошибку из-за разных возвращаемых значений. Вы не можете предугадать, как производители браузеров изменят встроенные объекты, так что любое их изменение может привести к конфликтам с вашим кодом, если он написан неаккуратно.

Таким образом, лучше никогда не изменять объекты, которые вам не принадлежат. А какие объекты принадлежат вам? Только те, которые вы создали сами, это может быть, например, пользовательский тип или литерал объекта. В то же время объекты

вроде Array, document и т. д., существовавшие еще до написания вашего кода, изменять не следует. Если нужно расширить функционал имеющихся объектов, можно сделать следующее:

- создайте новый объект с нужным функционалом и реализуйте его взаимодействие с исходным объектом;
- унаследуйте пользовательский тип от типа, который нужно изменить, и добавьте нужный функционал в новый тип.

Этот подход характерен для многих современных JavaScript-библиотек, что позволяет расширять и адаптировать их, несмотря на частые изменения браузеров.

Не используйте глобальные сущности

С соблюдением прав владения объектами тесно связан принцип отказа от глобальных переменных и функций. Он также имеет целью формирование согласованной и удобной для сопровождения среды выполнения сценариев. Суть принципа в том, что у вас должна быть только одна глобальная переменная, содержащая другие объекты и функции. Рассмотрим пример:

```
// две глобальные сущности - НЕ ДЕЛАЙТЕ ТАК!!!
var name = "Nicholas";
function sayName(){
    alert(name);
}
```

Этот код содержит две глобальные сущности: переменную name и функцию sayName(). Однако вы можете легко создать их в единственном глобальном объекте:

```
// одна глобальная сущность - предпочтительный подход
var MyApplication = {
    name: "Nicholas",
    sayName: function(){
        alert(this.name);
    }
};
```

Эта версия кода содержит единственный глобальный объект MyApplication с переменной name и методом sayName(), что устраняет проблемы прежнего кода. Во-первых, переменная name больше не перезаписывает свойство window.name, что в предыдущем примере могло мешать работе другого кода. Во-вторых, в новом коде проще понять, к чему относятся сущности. Например, вызов MyApplication.sayName() ясно показывает, что причины проблем с методом sayName() следует искать в объекте MyApplication.

Логичным развитием принципа единственного глобального объекта является концепция *пространств имен*, ставшая популярной благодаря библиотеке YUI (Yahoo! User Interface). Она включает создание объекта исключительно в качестве

контейнера для того или иного функционала. В YUI 2.x доступно несколько пространств имен, например:

- `YAHOO.util.Dom` — методы для манипулирования DOM;
- `YAHOO.util.Event` — методы для работы с событиями;
- `YAHOO.lang` — методы для использования низкоуровневых возможностей языка.

В YUI в качестве контейнера используется единственный глобальный объект `YAHOO`, в котором определены другие объекты. Когда объекты нужны для подобного группирования функционала, их называют пространствами имен. Вся библиотека YUI построена на основе этой концепции, что позволяет применять ее с любой другой JavaScript-библиотекой на одной странице.

При создании пространства имен важно выбрать подходящее имя глобального объекта, которое должно быть интуитивно понятным для всех, кто будет его использовать, но при этом достаточно уникальным, чтобы его не выбрали другие разработчики. Им вполне может быть имя компании, для которой вы разрабатываете приложение, например `YAHOO` или `Wrox`. После этого сгруппировать функционал в пространствах имен можно следующим образом:

```
// создание глобального объекта
var Wrox = {};

// создание пространства имен для данной книги
Wrox.ProJS = {};

// добавление других объектов, используемых в книге
Wrox.ProJS.EventUtil = { ... };
Wrox.ProJS.CookieUtil = { ... };
```

В этом примере пространства имен создаются в глобальном объекте `Wrox`. Если весь код для этой книги находится в пространстве имен `Wrox.ProJS`, ничто не мешает другим авторам добавить код в объект `Wrox`. Если все соблюдают этот принцип, можно не беспокоиться о том, что кто-то другой также создаст объект `EventUtil` или `CookieUtil`, потому что он будет относиться к другому пространству имен, например:

```
// создание пространства имен для книги Professional Ajax
Wrox.ProAjax = {};

// добавление других объектов, используемых в книге
Wrox.ProAjax.EventUtil = { ... };
Wrox.ProAjax.CookieUtil = { ... };

// объекты из пространства имен ProJS также доступны
Wrox.ProJS.EventUtil.addHandler( ... );

// доступ к объекту из пространства имен ProAjax
Wrox.ProAjax.EventUtil.addHandler( ... );
```

Хотя для использования пространств имен требуется немного больше кода, но это компенсируется удобством его сопровождения. Пространства имен гарантируют, что ваш код будет работать на одной странице с другим кодом без каких бы то ни было конфликтов.

Не сравнивайте значения с null

В JavaScript типы не проверяются автоматически, за это отвечают разработчики, но на деле они часто этим пренебрегают. Чаще всего при проверке типа значение сравнивают с `null`, но этот прием нередко используется неправильно, что приводит к ошибкам. Рассмотрим пример:

```
function sortArray(values){
    if (values != null){        // НЕ ДЕЛАЙТЕ ТАК!!
        values.sort(comparator);
    }
}
```

Эта функция предназначена для сортировки массива с помощью компаратора. Чтобы она работала правильно, аргумент `values` должен быть массивом, но инструкция `if` лишь сравнивает его с `null`. Проблема в том, что эту проверку пройдут также и другие значения, в том числе строка и любое число, что приведет к ошибке.

На практике сравнение значения с `null` нужно не так уж часто. Значения следует сравнивать с тем, чем они должны быть, а не наоборот. Например, в предыдущем фрагменте аргумент `values` должен быть массивом, поэтому следует проверить, действительно ли это массив, а не сравнивать его с `null`:

```
function sortArray(values){
    if (values instanceof Array){    // предпочтительный подход
        values.sort(comparator);
    }
}
```

Эта версия функции отсекает все недопустимые значения, а не только `null`.

Эта методика распознавания массива не работает, если веб-страница содержит несколько фреймов, потому что у каждого фрейма есть собственный глобальный объект и, соответственно, собственный конструктор Array. Если вы передаете массивы из одного фрейма в другой, можете вместо этого проверять существование метода sort().

Обнаружив в коде сравнение с `null`, попытайтесь заменить его, выбрав один из следующих приемов:

- если значение должно быть ссылочным, проверьте его конструктор с помощью оператора `instanceof`;
- если значение должно быть примитивным, проверьте его тип с помощью оператора `typeof`;

- если вы ожидаете объект с конкретным методом, убедитесь в наличии этого метода с помощью оператора `typeof`.

Чем меньше в коде сравнений с `null`, тем проще определить его назначение и устранить возможные ошибки.

Используйте константы

Хотя в JavaScript формально констант нет, они все же полезны. Идея в том, что данные следует изолировать от логики приложения, чтобы их можно было изменять, не рискуя внести ошибки. Рассмотрим пример:

```
function validate(value){
    if (!value){
        // недопустимое значение
        alert("Invalid value!");
        location.href = "/errors/invalid.php";
    }
}
```

Эта функция содержит два элемента данных: сообщение, отображаемое пользователю, и URL-адрес. Строки, которые отображаются в пользовательском интерфейсе, всегда следует выделять в специальные блоки, чтобы упростить локализацию приложения. С URL-адресами следует поступать так же, потому что они имеют свойство изменяться по мере развития приложения. Оба этих элемента данных могут измениться по той или иной причине, что потребует изменения кода функции. Каждый раз, когда вы меняете код логики приложения, вы можете внести в него ошибку. Чтобы предотвратить это, вы можете изолировать логику приложения от данных, заменив их константами, определенными отдельно, например:

```
var Constants = {
    INVALID_VALUE_MSG: "Invalid value!",
    INVALID_VALUE_URL: "/errors/invalid.php"
};

function validate(value){
    if (!value){
        alert(Constants.INVALID_VALUE_MSG);
        location.href = Constants.INVALID_VALUE_URL;
    }
}
```

В этой версии кода сообщение и URL-адрес определены в объекте `Constants`, а функция просто ссылается на них. Это позволяет изменять данные без изменения функции, в которой они используются. Вы даже можете определить объект `Constants` в отдельном файле и добавлять в него правильные значения на основе параметров интернационализации.

Главное здесь — отделить данные от логики, в которой они используются. Перечислим типы данных, с которыми имеет смысл так поступать:

- **Повторяющиеся значения** — любые значения, используемые более чем в одном месте (в том числе имена CSS-классов), следует определять как константы. Так вы не сможете допустить ошибку, изменив лишь одно значение, когда нужно изменить несколько.
- **Строки пользовательского интерфейса** — для упрощения интернационализации любые строки, отображаемые пользователю, следует извлекать из кода.
- **URL-адреса** — ссылки на ресурсы часто изменяются в веб-приложениях, так что рекомендуется хранить все URL-адреса в одном месте.
- **Любые значения, которые могут измениться**, — каждый раз, когда вы используете в коде литерал, спросите себя, может ли он в будущем измениться. Если да, то это значение следует выделить в константу.

Использование констант — важный аспект разработки корпоративных JavaScript-приложений, потому что это упрощает сопровождение кода и защищает его от изменений данных.

Быстродействие

Объем JavaScript-кода на типичной веб-странице за время существования языка значительно увеличился, в связи с чем его быстродействие стало вызывать нарекания. JavaScript изначально был интерпретируемым языком, так что скорость выполнения написанного на нем кода традиционно намного ниже, чем в компилируемых языках. Chrome стал первым браузером, в котором был представлен оптимизирующий модуль, компилирующий JavaScript во встроенный машинный код. Вслед за этим компиляторы JavaScript были реализованы во всех основных браузерах.

Конечно, компиляция мало чем поможет, если код написан плохо, но использование некоторых базовых шаблонов позволяет сделать код максимально быстрым.

Область видимости

В главе 4 мы обсудили области видимости и цепочку областей видимости в JavaScript. При увеличении количества областей видимости в цепочке возрастает и время доступа к переменным вне текущей области видимости. Из-за просмотра цепочки доступ к глобальной переменной всегда осуществляется медленнее, чем к локальной. Сократив время просмотра цепочки областей видимости, можно повысить общее быстродействие сценария.

Минимизируйте доступ к глобальным сущностям

Пожалуй, самое важное, что можно сделать для повышения быстродействия сценариев, — это минимизировать доступ к глобальным сущностям. Для доступа к глобальным переменным и функциям всегда требуется больше ресурсов в сравнении

с локальными операциями, потому что он подразумевает просмотр цепочки областей видимости. Рассмотрим следующую функцию:

```
function updateUI(){
    var imgs = document.getElementsByTagName("img");
    for (var i=0, len=imgs.length; i < len; i++){
        imgs[i].title = document.title + " image " + i;
    }

    var msg = document.getElementById("msg");
    msg.innerHTML = "Update complete.";
}
```

Эта функция работает правильно, но содержит три ссылки на глобальный объект `document`. Если на странице много изображений, код доступа к нему в цикле `for` может быть выполнен десятки и даже сотни раз, при этом каждый раз требуется просмотреть цепочку областей видимости. Создав локальную переменную, указывающую на объект `document`, вы можете сократить количество просмотров до одного, намного повысив быстродействие функции:

```
function updateUI(){
    var doc = document;
    var imgs = doc.getElementsByTagName("img");
    for (var i=0, len=imgs.length; i < len; i++){
        imgs[i].title = doc.title + " image " + i;
    }

    var msg = doc.getElementById("msg");
    msg.innerHTML = "Update complete.";
}
```

Здесь объект `document` сохраняется в локальной переменной `doc`, которая затем используется вместо него во всем остальном коде. Эта функция обращается к глобальному объекту лишь один раз, благодаря чему выполняется гораздо быстрее.

На практике имеет смысл сохранять в локальной переменной любой глобальный объект, используемый в функции более одного раза.

Не используйте инструкцию with

Если быстродействие действительно важно, инструкцию `with` лучше не использовать. Подобно функциям, она создает собственную область видимости, увеличивая длину цепочки для кода внутри `with`. Такой код всегда выполняется медленнее, чем код снаружи `with`, из-за накладных расходов на просмотр цепочки областей видимости.

Инструкция `with` требуется редко, потому что она служит в основном для сокращения объема вводимого кода. Как правило, того же результата можно добиться с помощью локальной переменной без создания новой области видимости:

```
function updateBody(){
    with(document.body){
        alert(tagName);
        innerHTML = "Hello world!";
    }
}
```

В этом коде инструкция `with` упрощает работу со свойством `document.body`, но локальная переменная позволяет делать то же самое:

```
function updateBody(){
    var body = document.body;
    alert(body.tagName);
    body.innerHTML = "Hello world!";
}
```

Хотя этот код немного длиннее, он более понятен, чем предыдущий фрагмент, потому что сразу ясно, к какому объекту относятся свойства `tagName` и `innerHTML`. Кроме того, сохранение значения `document.body` в локальной переменной экономит ресурсы на доступ к глобальному объекту.

Выбор оптимального подхода

Как и в других языках, в JavaScript быстродействие во многом зависит от алгоритма решения проблемы. Профессиональные разработчики по опыту знают, какие подходы более эффективны, а какие менее. Многие подходы и приемы, часто используемые в других языках программирования, можно применять и в JavaScript.

Не просматривайте свойства без необходимости

В программировании сложность алгоритмов описывается с помощью O-нотации. Простейшие и быстрейшие алгоритмы имеют постоянную сложность, или O(1), после чего алгоритмы становятся более сложными и медленными. Основные типы JavaScript-алгоритмов описаны таблице.

Нотация	Сложность	Описание
O(1)	Постоянная	
O(log *n*)	Логарифмическая	Время выполнения алгоритма зависит от количества значений, но получать все значения не требуется. Пример: двоичный поиск
O(*n*)	Линейная	Время выполнения алгоритма пропорционально количеству значений. Пример: перебор всех элементов массива
O(n^2)	Квадратичная	Для выполнения алгоритма необходимо обратиться к каждому значению не менее *n* раз. Пример: сортировка вставками

Примером операции O(1), может служить доступ к литералу или значению переменной. Нотация O(1) указывает, что время, необходимое для получения значения, остается постоянным независимо от количества значений. Получение значения — эффективная и быстрая операция, например:

```
var value = 5;
var sum = 10 + value;
alert(sum);
```

В этом коде мы обращаемся к четырем значениям: числу 5, переменной `value`, числу 10 и переменной `sum`. Общая сложность этого кода — O(1).

Доступ к элементам массива также является в JavaScript операцией со сложностью O(1) и почти столь же эффективен, как и простой просмотр переменной, например:

```
var values = [5, 10];
var sum = values[0] + values[1];
alert(sum);
```

Использование переменных и массивов более эффективно, чем доступ к свойствам объектов, который имеет сложность O(n). Просмотр свойства объекта занимает больше времени, чем доступ к переменной или массиву, потому что свойство с указанным именем нужно найти в цепочке прототипов. Проще говоря, чем больше в коде операций доступа к свойствам, тем медленнее работает такой код. Рассмотрим пример:

```
var values = { first: 5, second: 10};
var sum = values.first + values.second;
alert(sum);
```

Здесь для вычисления значения `sum` требуется прочитать два свойства. Просмотр одного или двух свойств — это, конечно, не проблема, но если таких операций сотни или тысячи, они определенно скажутся на быстродействии.

Избегайте чтения нескольких свойств для получения одного значения, например:

```
var query =
    window.location.href.substring(window.location.href.indexOf("?"));
```

Этот код просматривает шесть свойств: три при вызове метода `window.location.href.substring()` и еще три при вызове `window.location.href.indexOf()`. Вы можете легко подсчитать операции доступа к свойствам по точкам. Этот код примечателен еще и тем, что доступ к свойству `window.location.href` выполняется два раза без какой-либо оптимизации.

Если свойство объекта используется более одного раза, следует сохранить его в локальной переменной. В этом случае для первого доступа к свойству потребуется время O(n), но зато каждый последующий доступ будет выполняться за время

O(1), что более чем компенсирует начальные издержки. Например, предыдущий код можно переписать так:

```
var url = window.location.href;
var query = url.substring(url.indexOf("?"));
```

Эта версия включает только четыре просмотра свойства, что на треть меньше в сравнении с оригиналом. В большом сценарии это может сэкономить много ресурсов.

Всякий раз, когда можно снизить сложность алгоритма, не следует пренебрегать этим. Таким образом, обращения к свойствам следует во всех возможных случаях заменять локальными переменными, а если доступ к какому-либо значению возможен с помощью числового индекса или именованного свойства (например, при работе с объектом `NodeList`), лучше использовать индекс.

Оптимизируйте циклы

Циклы — одна из чаще всего используемых конструкций в программировании в целом и в JavaScript в частности. Оптимизация циклов вносит особо важный вклад в повышение быстродействия, потому что код в циклах выполняется многократно. Оптимизация циклов в других языках хорошо изучена, и многие приемы из них также применимы и в JavaScript. Рассмотрим эти основные приемы:

1. **Уменьшение итератора** — обычно в цикле используется итератор, который начинается с нуля и увеличивается до некоторого значения. Во многих случаях эффективнее инициализировать итератор максимальным значением и уменьшать его на каждой итерации цикла.
2. **Упрощение окончательного условия** — поскольку окончательное условие оценивается на каждой итерации цикла, желательно сделать его как можно более простым. Это означает, что в нем в идеале не должно быть просмотров свойств и других операций со сложностью O(n).
3. **Упрощение тела цикла** — основная доля ресурсов расходуется на выполнение тела цикла, поэтому его оптимизация заслуживает повышенного внимания. Убедитесь, что тело цикла не содержит ресурсоемких вычислений, которые можно вынести за пределы цикла.
4. **Использование циклов с постусловием** — чаще всего используются циклы `for` и `while`, в которых проверяется предусловие. В циклах с постусловием, таких как `do-while`, нет первоначальной оценки окончательного условия, поэтому они обычно выполняются быстрее.

Эти приемы лучше всего продемонстрировать на примере. Вот простой цикл `for`:

```
for (var i=0; i < values.length; i++){
    process(values[i]);
}
```

Этот код увеличивает переменную `i` от нуля до общего количества элементов в массиве `values`. Если порядок обработки элементов не важен, увеличение итератора можно заменить уменьшением:

```
for (var i=values.length-1; i >= 0; i--){
    process(values[i]);
}
```

В новой версии переменная `i` на каждой итерации уменьшается, благодаря чему в окончательном условии доступ к значению `values.length` — операция сложности $O(n)$ — сменился сравнением итератора с нулем, которое имеет сложность $O(1)$. Поскольку тело цикла включает только одну инструкцию, оптимизировать его невозможно, однако сам цикл можно преобразовать в цикл с постусловием:

```
var i=values.length-1;
if (i > -1){
    do {
        process(values[i]);
    }while(--i >= 0);
}
```

Главный фактор оптимизации здесь — это объединение окончательного условия и оператора декремента в одной инструкции. Теперь цикл оптимизирован полностью, и дальнейшее повышение быстродействия без изменения функции `process()` невозможно.

Цикл с постусловием дает прирост быстродействия лишь в том случае, если вы точно знаете, что нужно будет обработать хотя бы одно значение. Например, при пустом массиве в таком цикле все равно будет выполнена одна итерация, которая не потребовалась бы в цикле с предусловием.

Развертывайте циклы

Если количество итераций цикла конечно, часто эффективнее заменить его несколькими вызовами функции. Давайте вернемся к циклу из предыдущего раздела. Если длина массива всегда будет одинаковой, возможно, лучше просто вызвать функцию `process()` для каждого элемента:

```
// развернутый цикл
process(values[0]);
process(values[1]);
process(values[2]);
```

В этом примере предполагается, что массив `value` содержит только три элемента, поэтому функция `process()` вызывается три раза. Такое развертывание цикла позволяет не тратить ресурсы на его настройку и обработку конечного условия, что повышает быстродействие кода.

Если количество итераций цикла неизвестно, вы можете попробовать так называемый *метод Даффа* (Duff's device). Этот прием придумал Том Дафф (Tom Duff), впервые предложивший использовать его в языке программирования C. Реализацию метода Даффа на JavaScript приписывают Джеффу Гринбергу (Jeff Greenberg). Суть метода Даффа — развертывание цикла в последовательность инструкций, выполняемых в новом цикле с количеством итераций, уменьшенным в 8 раз. Рассмотрим пример:

```
// реализация метода Даффа на JS - Джефф Гринберг
// предполагается, что values.length > 0
var iterations = Math.ceil(values.length / 8);
var startAt = values.length % 8;
var i = 0;

do {
    switch(startAt){
        case 0: process(values[i++]);
        case 7: process(values[i++]);
        case 6: process(values[i++]);
        case 5: process(values[i++]);
        case 4: process(values[i++]);
        case 3: process(values[i++]);
        case 2: process(values[i++]);
        case 1: process(values[i++]);
    }
    startAt = 0;
} while (--iterations > 0);
```

Эта реализация метода Даффа начинается с вычисления количества итераций цикла, для чего общее количество элементов в массиве `values` делится на 8 и результат округляется вверх до целого числа методом `ceil()`. Переменной `startAt` присваивается количество элементов, оставшихся после деления на 8, и на первой итерации цикла она используется для обработки этих элементов. Например, если массив содержит 10 значений, переменная `startAt` будет равна 2, поэтому на первой итерации функция `process()`вызывается только два раза. В конце первой итерации переменная `startAt` обнуляется, так что на каждой последующей итерации функция `process()` вызывается 8 раз. Такое развертывание цикла ускоряет обработку больших наборов данных.

В книге *Speed Up Your Site* Эндрю Б. Кинг (Andrew B. King, New Riders, 2003) предложил еще более быстрый метод Даффа, подразумевающий разделение цикла `do-while` на два цикла:

```
// код взят из книги "Speed Up Your Site" (New Riders, 2003)
var iterations = Math.floor(values.length / 8);
var leftover = values.length % 8;
var i = 0;

if (leftover > 0){
    do {
```

```
        process(values[i++]);
    } while (--leftover > 0);
}

do {
    process(values[i++]);
    process(values[i++]);
    process(values[i++]);
    process(values[i++]);
    process(values[i++]);
    process(values[i++]);
    process(values[i++]);
    process(values[i++]);
} while (--iterations > 0);
```

В этой реализации первый цикл обрабатывает элементы, оставшиеся после деления на 8 (значение `leftover`). Когда эти дополнительные элементы обработаны, выполняется основной цикл, в котором 8 раз вызывается функция `process()`. Этот подход почти на 40 % быстрее, чем оригинальная реализация метода Даффа.

Развертывание циклов может сэкономить много ресурсов при обработке больших наборов данных, но менее эффективно, если данных немного. Небольшое повышение быстродействия за счет увеличения объема кода в этом случае обычно не стоит затрачиваемых усилий.

Избегайте двойной интерпретации

Двойная интерпретация имеет место, когда код JavaScript интерпретирует другой код JavaScript. Такая ситуация возникает при использовании функции `eval()`, конструктора `Function` и вызове функции `setTimeout()` со строковым аргументом, например:

```
// интерпретация некоторого кода – НЕ ДЕЛАЙТЕ ТАК!!
eval("alert('Hello world!')");

// создание новой функции – НЕ ДЕЛАЙТЕ ТАК!!
var sayHi = new Function("alert('Hello world!')");

// установка тайм-аута – НЕ ДЕЛАЙТЕ ТАК!!
setTimeout("alert('Hello world!')", 500);
```

В каждом из этих примеров необходимо интерпретировать строку, содержащую JavaScript-код. При первоначальном синтаксическом анализе сделать это невозможно, потому что код содержится в строке, а это означает, что для ее анализа нужно запустить новый синтаксический анализатор, что требует много ресурсов. В результате такой код работает медленнее в сравнении со встроенным.

Все эти примеры можно заменить более эффективными аналогами. Метод `eval()` по-настоящему требуется редко, и по возможности лучше его не использовать. В приведенном примере код можно просто встроить. Вызов конструктора `Function`

можно легко переписать как обычную функцию, а что касается метода `setTimeout()`, то в него можно передать функцию в качестве первого аргумента:

```
// исправлено
alert('Hello world!');

// создание новой функции - исправлено
var sayHi = function(){
    alert('Hello world!');
};

// установка тайм-аута - исправлено
setTimeout(function(){
    alert('Hello world!');
}, 500);
```

Чтобы код работал быстрее, старайтесь не использовать строки JavaScript-кода, требующие интерпретации.

Другие соображения по поводу быстродействия

При анализе быстродействия сценария следует учитывать еще несколько факторов. Их влияние менее заметно, но при частом использовании они могут повлиять на быстродействие кода.

- **Встроенные методы быстрее**. Старайтесь использовать встроенные методы, а не написанные на JavaScript. Встроенные методы реализованы на компилируемых языках, таких как C и C++, а потому работают гораздо быстрее. В частности, разработчики часто забывают о сложных математических методах, доступных объекту `Math`; эти методы работают быстрее, чем любые эквиваленты синуса, косинуса и т. д., написанные на JavaScript.
- **Инструкция switch быстрее**. Если код содержит сложную последовательность инструкций `if-else`, то преобразовав ее в одну инструкцию `switch`, можно повысить его быстродействие. Чтобы сделать инструкцию `switch` еще более быстрой, организуйте варианты от наиболее вероятных к наименее вероятным.
- **Поразрядные операторы быстрее**. Поразрядные математические операции всегда выполняются быстрее, чем логические и числовые. Выборочная замена арифметических операций поразрядными может значительно ускорить сложные вычисления. Как нельзя лучше для такой замены подходят операции вроде деления по модулю, логического ИЛИ и логического И.

Сокращение количества инструкций

Скорость выполнения операций также зависит от количества инструкций в JavaScript-коде. Единственная инструкция, охватывающая много операций, выполняется быстрее, чем много инструкций, каждая из которых выполняет

одну операцию. Грамотное объединение инструкций позволяет сократить время выполнения всего сценария, при этом особого внимания заслуживают несколько шаблонов, о которых пойдет речь в этом разделе.

Несколько объявлений переменных

При объявлении переменных разработчики часто создают избыточные инструкции. Например, нередко встречается код, в котором переменные объявляются с помощью нескольких инструкций `var`:

```
// четыре инструкции - лишние расходы
var count = 5;
var color = "blue";
var values = [1,2,3];
var now = new Date();
```

В строго типизированных языках переменные разных типов нужно объявлять с помощью отдельных инструкций, но в JavaScript все переменные можно объявить в одной инструкции `var`:

```
// одна инструкция
var count = 5,
    color = "blue",
    values = [1,2,3],
    now = new Date();
```

Здесь все объявления переменных относятся к одной инструкции `var` и разделены запятыми. Этот код работает гораздо быстрее, чем объявление переменных по отдельности, а внести такое изменение совсем не сложно.

Вставка итеративных значений

Каждый раз, когда вы используете итеративное значение (то есть значение, которое в разных местах кода увеличивается или уменьшается на единицу), объединяйте инструкции, если это возможно. Рассмотрим пример:

```
var name = values[i];
i++;
```

Каждая из двух предыдущих инструкций выполняет единственную задачу: первая получает значение из массива `values` и сохраняет его в переменной `name`, а вторая увеличивает переменную `i`. Их можно объединить в одной инструкции следующим образом:

```
var name = values[i++];
```

Эта инструкция эквивалентна двум предыдущим. Поскольку оператор инкремента является постфиксным, значение `i` увеличивается только после выполнения

остальной части инструкции. Встретив похожий фрагмент, попытайтесь вставить итеративное значение в последнюю инструкцию, где оно используется.

Использование литералов массивов и объектов

В книге мы обсудили два способа создания массивов и объектов: с помощью конструктора и с помощью литерала. В первом случае для определения свойств или вставки элементов всегда требуется несколько инструкций, тогда как во втором все делается в одной инструкции. Рассмотрим пример:

```
// четыре инструкции для создания и инициализации массива – лишние расходы
var values = new Array();
values[0] = 123;
values[1] = 456;
values[2] = 789;

// четыре инструкции для создания и инициализации объекта – лишние расходы
var person = new Object();
person.name = "Nicholas";
person.age = 29;
person.sayName = function(){
    alert(this.name);
};
```

Этот код создает и инициализирует массив и объект. В обоих случаях требуется выполнить четыре инструкции: одну для вызова конструктора и три для назначения данных. С помощью литералов можно заменить этот код следующим:

```
// одна инструкция для создания и инициализации массива
var values = [123, 456, 789];

// одна инструкция для создания и инициализации объекта
var person = {
    name : "Nicholas",
    age : 29,
    sayName : function(){
        alert(this.name);
    }
};
```

Измененный код содержит только две инструкции: одна создает и инициализирует массив, а вторая — объект. Таким образом, мы заменили восемь инструкций двумя, уменьшив их количество на 75 %. Польза от такой оптимизации особенно заметна, если база кода содержит тысячи строк.

В общем, старайтесь заменять объявления массивов и объектов их литералами для сокращения количества инструкций.

Использование литералов в Internet Explorer 6 и более ранних версий немного снижает быстродействие кода. В Internet Explorer 7 эта проблема устранена.

Оптимизация взаимодействия с DOM

Из всех JavaScript-компонентов самыми медленным, несомненно, является DOM. Взаимодействие с DOM занимает много времени, потому что часто требует частичной или полной перерисовки страницы. Даже тривиальные операции могут выполняться неожиданно долго, потому что DOM управляет огромными объемами данных. Оптимизация взаимодействия с DOM может существенно ускорить работу сценариев.

Минимизируйте динамические обновления

При доступе к той части DOM, которая отображается на странице, выполняется *динамическое обновление* (live update). Оно называется так потому, что страница обновляется немедленно. Каждое изменение, будь то вставка одного знака или удаление целого раздела, имеет свою цену, потому что для обновления браузер пересчитывает тысячи разных параметров. Чем больше операций динамического обновления вы инициируете, тем дольше выполняется код, и наоборот. Рассмотрим пример:

```
var list = document.getElementById("myList"),
           item,
           i;

for (i=0; i < 10; i++) {
    item = document.createElement("li");
    list.appendChild(item);
    item.appendChild(document.createTextNode("Item " + i));
}
```

Этот код добавляет в список 10 записей. Для каждой из них выполняются два динамических обновления: одно при добавлении элемента `<li>` и одно при добавлении к нему текстового узла. Иначе говоря, для добавления 10 записей требуется 20 динамических обновлений.

Для экономии ресурсов желательно сократить количество таких обновлений. Это можно сделать двумя способами. Первый — удалить список со страницы, выполнить его обновление, а затем вставить список обратно в том же месте. К сожалению, этот подход может вызывать мерцание при обновлениях страницы. Второй способ — создать DOM-структуру с помощью фрагмента документа, а затем добавить ее к элементу `list`. Этот подход предотвращает динамические обновления и мерцание страницы. Рассмотрим пример:

```
var list = document.getElementById("myList"),
    fragment = document.createDocumentFragment(),
    item,
    i;

for (i=0; i < 10; i++) {
    item = document.createElement("li");
```

```
    fragment.appendChild(item);
    item.appendChild(document.createTextNode("Item " + i));
}

list.appendChild(fragment);
```

В этом примере выполняется только одно динамическое обновление после создания всех записей. Созданные записи временно сохраняются во фрагменте документа, а затем все записи добавляются в список с помощью метода `appendChild()`. Помните, что когда фрагмент документа передается в метод `appendChild()`, к родительскому элементу добавляются все дочерние элементы фрагмента, но не сам фрагмент.

Если необходимо обновить DOM, иногда имеет смысл создать DOM-структуру с помощью фрагмента документа, чтобы можно было внести в документ сразу все изменения.

Используйте свойство innerHTML

Создавать DOM-узлы на странице можно двумя способами: с помощью DOM-методов, таких как `createElement()` и `appendChild()`, и с помощью свойства `innerHTML`. При небольших изменениях в DOM быстродействие в обоих случаях примерно одинаково, но при значительных изменениях второй способ гораздо быстрее.

Когда вы присваиваете значение свойству `innerHTML`, за кулисами создается синтаксический анализатор HTML, а структура DOM составляется с помощью встроенных методов DOM, а не методов, реализованных на JavaScript. Встроенные методы выполняются гораздо быстрее, потому что они уже скомпилированы, то есть их не нужно интерпретировать. Предыдущий пример можно переписать со свойством `innerHTML` следующим образом:

```
var list = document.getElementById("myList"),
    html = "",
    i;

for (i=0; i < 10; i++) {
    html += "<li>Item " + i + "</li>";
}

list.innerHTML = html;
```

Здесь мы создаем строку HTML-кода, а затем назначаем ее свойству `list.innerHTML`, формируя нужную DOM-структуру. Хотя конкатенация строк тоже требует некоторых ресурсов, эта методика все же работает быстрее, чем многократное взаимодействие с DOM.

Как и другие операции с DOM, обращения к свойству `innerHTML` нужно стараться свести к минимуму. Например, в следующем коде их слишком много:

```
var list = document.getElementById("myList"),
    i;
```

```
for (i=0; i < 10; i++) {
    list.innerHTML += "<li>Item " + i + "</li>";    // НЕ ДЕЛАЙТЕ ТАК!!!
}
```

Проблема с этим кодом в том, что свойство `innerHTML` обновляется на каждой итерации цикла, что очень неэффективно. На самом деле обновление свойства `innerHTML` является динамическим, поэтому код будет работать гораздо быстрее, если сначала полностью составить строку, а затем назначить ее свойству за один раз.

Делегируйте события

В большинстве веб-приложений для взаимодействия с пользователями используется множество обработчиков событий. Время отклика страницы на действия пользователя напрямую зависит от количества обработчиков на ней. Чтобы сократить эти накладные расходы, следует по мере возможности делегировать события.

Как отмечалось в главе 13, делегирование событий возможно благодаря тому, что некоторые из них всплывают. Любое всплывающее событие можно обработать не только у целевого элемента события, но и у любых его предков. Это позволяет обработчику, подключенному к элементу более высокого уровня, обрабатывать события с несколькими целевыми элементами. По возможности старайтесь подключать обработчики событий на уровне документа, чтобы обрабатывать события всей страницы.

Оптимизируйте доступ к объектам HTMLCollection

Из-за ряда недостатков, которые уже не раз обсуждались в книги, объекты `HTMLCollection` существенно снижают быстродействие веб-приложений. Помните, что при каждом доступе к свойству или методу `HTMLCollection` выполняется запрос документа, на что требуется довольно много ресурсов. Сведя к минимуму использование объектов `HTMLCollection`, можно заметно ускорить сценарий.

Пожалуй, важнее всего оптимизировать доступ к объектам `HTMLCollection` в циклах. Вычисление количества элементов при инициализации цикла `for` мы уже обсуждали, а теперь рассмотрим такой пример:

```
var images = document.getElementsByTagName("img"),
    i, len;

for (i=0, len=images.length; i < len; i++){
    // обработка данных
}
```

Суть примера в том, что мы сохраняем значение `length` в переменной `len`, что позволяет не обращаться каждый раз к свойству `length` объекта `HTMLCollection`. Чтобы далее оптимизировать использование объекта `HTMLCollection` в цикле, следует получать ссылки на его элементы и работать с ними, а не обращаться к `HTMLCollection` непосредственно:

```
var images = document.getElementsByTagName("img"),
    image,
    i, len;

for (i=0, len=images.length; i < len; i++){
    image = images[i];
    // обработка данных
}
```

В этом цикле текущее изображение сохраняется в переменной `image`. После этого обращаться к объекту `images` типа `HTMLCollection` в цикле больше не требуется.

Чтобы свести к минимуму количество операций доступа к объектам `HTMLCollection`, важно понимать, когда они возвращаются. Это имеет место в следующих случаях:

- вызов метода `getElementsByTagName()`;
- получение свойства `childNodes` элемента;
- получение свойства `attributes` элемента;
- доступ к специальному набору, такому как `document.forms`, `document.images` и т. д.

Грамотное использование объектов `HTMLCollection` может заметно повысить быстродействие кода.

Развертывание

Наверное, наиболее важным этапом создания веб-сайта или веб-приложения на JavaScript является его развертывание. К этому моменту вы уже спроектировали решение, написали его код, оптимизировали и, наконец, готовы вывести его в Интернет, чтобы пользователи смогли оценить его по достоинству. Однако прежде чем сделать это, нужно разобраться с несколькими вопросами.

Процесс сборки

Для подготовки JavaScript-кода к развертыванию очень важно продумать процесс его сборки. Типичный цикл разработки ПО включает этапы написания кода, его компиляции и тестирования, но поскольку JavaScript не компилируется, второй этап этого цикла выпадает, так что в браузере вы тестируете тот же код, который написали. К сожалению, этот подход не оптимален. По приведенным далее причинам написанный код не должен нетронутым попадать в браузер.

- **Права на интеллектуальную собственность**. Если вы поместите в Интернет полностью прокомментированный исходный код, другим людям будет проще понять, что вы делаете, повторно использовать фрагменты кода и, возможно, даже атаковать ваше приложение.

- **Размеры файлов**. Разработчики пишут код так, чтобы его было легко читать. Это хорошо с точки зрения его сопровождения, но плохо в плане быстродействия. Дополнительные пробелы, отступы и подробные имена функций и переменных никак не помогают браузеру, а только замедляют его работу.
- **Организация кода**. Код, удобный для сопровождения, не всегда наиболее эффективно обрабатывается браузером.

По этим причинам следует хорошо продумать процесс сборки JavaScript-файлов.

Процесс сборки начинается с определения логической структуры для хранения файлов в системе управления исходным кодом. Писать весь JavaScript-код в одном файле не рекомендуется — лучше придерживаться подхода, который обычен для объектно-ориентированных языков, и определять каждый объект и пользовательский тип в отдельном файле. Благодаря этому каждый файл будет содержать минимум кода, что поможет изменять его без внесения ошибок. Кроме того, в распределенных системах управления исходным кодом, таких как CVS или Subversion, это снижает вероятность конфликтов во время операций слияния.

Помните, что разделение кода на несколько файлов выполняется для удобства его сопровождения, а не для развертывания. Для развертывания, наоборот, желательно объединить исходные файлы в один или несколько пакетов. В веб-приложениях рекомендуется использовать как можно меньше JavaScript-файлов, потому что обработка HTTP-запросов — одно из основных «узких мест» в Сети. Не забывайте о том, что включение JavaScript-файла в код страницы с помощью тега `<script>` — блокирующая операция, на время которой прекращается загрузка всех других файлов, и старайтесь для развертывания логически группировать JavaScript-код в пакеты.

Когда вы определитесь со структурой файлов и каталогов и решите, что должно находиться в пакетах, следует создать систему сборки. Для автоматизации сборки JavaScript-кода используется *средство сборки Ant* (http://ant.apache.org), которое было создано для Java, но благодаря простоте завоевало популярность и среди разработчиков веб-приложений. Жюльен Леком (Julien Lecomte) описал автоматизацию сборки кода JavaScript и CSS с помощью Ant по адресу www.julienlecomte.net/blog/2007/09/16/.

Ant идеально подходит для сборки JavaScript-кода благодаря простоте манипулирования файлами. Например, вы можете легко получить все файлы в каталоге и объединить их в один файл:

Листинг SampleAntDir/build.xml


```
<project name="Проект JavaScript" default="js.concatenate">

    <!-- каталог для вывода -->
    <property name="build.dir" value="./js" />

    <!-- каталог с исходными файлами -->
    <property name="src.dir" value="./dev/src" />
```

```
    <!-- целевой файл для объединения всех JS-файлов -->
    <!-- Жюльен Леком, http://www.julienlecomte.net/blog/2007/09/16/ -->
    <target name="js.concatenate">
        <concat destfile="${build.dir}/output.js">
            <filelist dir="${src.dir}/js" files="a.js, b.js"/>
            <fileset dir="${src.dir}/js" includes="*.js"
                                         excludes="a.js, b.js"/>
        </concat>
    </target>

</project>
```

В этом файле `build.xml` определены два свойства: каталог сборки, в котором будет сохранен итоговый файл, и каталог с исходными JavaScript-файлами. Элемент `<concat>` определяет список файлов, которые нужно объединить, и место, где нужно сохранить итоговый файл. Элемент `<filelist>` указывает, что файлы `a.js` и `b.js` должны быть первыми в объединенном файле, а элемент `<fileset>` предписывает добавить после них все остальные файлы. Итоговый файл должен быть сохранен как `/js/output.js`.

Если средство Ant установлено, вы можете перейти в каталог с файлом `build.xml` и ввести следующую команду:

```
ant
```

Начнется процесс сборки, по завершении которого вы получите объединенный файл. Если в файле есть другие целевые элементы, выполнить только `js.concatenate` можно следующим образом:

```
ant js.concatenate
```

В зависимости от требований можно изменить процесс сборки, добавив или удалив некоторые этапы. Интеграция сборки в цикл разработки позволяет обеспечить дополнительную обработку JavaScript-файлов перед их развертыванием.

Проверка кода

Хотя уже начали появляться IDE с поддержкой JavaScript, большинство разработчиков все еще проверяют правильность кода, запуская его в браузере. Однако этот подход имеет несколько недостатков. Во-первых, такой способ проверки трудно автоматизировать или перенести в другую систему. Во-вторых, если не брать в расчет синтаксические ошибки, этот способ выявляет проблемы только при выполнении кода. Для обнаружения потенциальных проблем с JavaScript-кодом есть несколько средств, наиболее популярным из которых является утилита *JSLint* (www.jslint.com), предложенная Дугласом Крокфордом (Douglas Crockford).

JSLint ищет в коде синтаксические и некоторые другие часто допускаемые ошибки, в том числе следующие:

- использование функции `eval()`;
- использование необъявленных переменных;
- отсутствие точек с запятой;
- неправильные разрывы строк;
- неправильное использование запятых;
- отсутствие фигурных скобок;
- отсутствие ключевого слова `break` в инструкции `switch`;
- двойное объявление переменных;
- использование инструкции `with`;
- использование одинарных знаков равенства вместо двойных или тройных;
- недоступный код.

Эту утилиту можно запустить как онлайн, так и из командной строки с помощью основанного на Java интерпретатора JavaScript Rhino (www.mozilla.org/rhino/). Чтобы запустить JSLint из командной строки, нужно сначала загрузить Rhino, а затем – JSLint для Rhino (www.jslint.com/rhino/). После установки можно запустить JSLint следующим образом:

```
java -jar rhino-1.6R7.jar jslint.js [входные файлы]
```

Вот пример:

```
java -jar rhino-1.6R7.jar jslint.js a.js b.js c.js
```

При наличии проблем с синтаксисом или потенциальных ошибок в указанных файлах будет выведен отчет с ошибками и предупреждениями. Если проблем с кодом нет, никаких сообщений вы не увидите.

Кроме того, вы можете запустить JSLint во время сборки кода с помощью средства Ant:

Листинг SampleAntDir/build.xml


```
<target name="js.verify">
    <apply executable="java" parallel="false">
        <fileset dir="${build.dir}" includes="output.js"/>
        <arg line="-jar"/>
        <arg path="${rhino.jar}"/>
        <arg path="${jslint.js}" />
        <srcfile/>
    </apply>
</target>
```

В этом фрагменте предполагается, что расположение JAR-файла Rhino указано с помощью свойства `rhino.jar`, а JSLint-файла Rhino — с помощью свойства `jslint.js`. Во время сборки средство JSLint проверит файл `output.js` и выведет сведения обо всех найденных проблемах.

Добавление проверки кода в цикл разработки помогает избегать ошибок. Рекомендуется делать это также во время сборки для выявления потенциальных проблем, прежде чем они проявят себя как ошибки.

Список средств проверки JavaScript-кода приведен в приложении Г.

Сжатие

В контексте сжатия JavaScript-файлов нас интересуют две величины: *размер кода* и *объем трафика*. Размер кода — это количество байтов, которые должен обработать браузер, а объем трафика — это количество байтов, передаваемых сервером браузеру. На заре веб-разработки эти два значения почти всегда были равны, потому что сервер передавал клиенту исходные файлы без каких-либо изменений. В наше время эти значения не совпадают почти никогда.

Сжатие файлов

Поскольку JavaScript-код не компилируется в байт-код и передается по сети в исходном виде, файлы с ним часто содержат дополнительные данные и элементы форматирования, которые не требуются браузеру. Комментарии, дополнительные пустые места и длинные имена переменных или функций упрощают чтение кода, но при отправке лишь увеличивают потребление ресурсов. Чтобы уменьшить размер файла, можно сжать его.

При сжатии обычно выполняются следующие действия (все или некоторые из них):

- удаление ненужных пустот (включая разрывы строк);
- удаление всех комментариев;
- сокращение имен переменных.

Есть много средств сжатия, ориентированных на JavaScript (полный список см. в приложении Г), но лучшим из них, пожалуй, является YUI Compressor (http://developer.yahoo.com/yui/compressor/). YUI Compressor разбирает код JavaScript на лексемы с помощью синтаксического анализатора Rhino и выдает поток лексем, чтобы можно было удалить из него пробелы и комментарии. В отличие от средств сжатия, основанных на регулярных выражениях, YUI Compressor исключает внесение в код синтаксических ошибок, а потому может безопасно сокращать имена локальных переменных.

YUI Compressor распространяется как Java-файл `yuicompressor-x.y.z.jar`, где `x.y.z` — номер версии. На день написания этих строк последняя версия имеет номер 2.4.6. Запустить YUI Compressor из командной строки можно следующим образом:

```
java -jar yuicompressor-x.y.z.jar [параметры] [входной файл]
```

Параметры YUI Compressor указаны в таблице.

Параметр	Описание
-h	Вывод справочной информации
-o *выходнойФайл*	Имя выходного файла. Если этот параметр не указан, именем выходного файла является имя входного файла с суффиксом «-min». Например, для входного файла input.js создается файл input-min.js
--line-break *столбец*	Предписывает добавить разрыв строки после указанного количества символов. По умолчанию сжатый файл выводится в одной строке, что может привести к проблемам в некоторых системах управления исходным кодом
-v,--verbose	Включает подробный режим, в котором выводятся советы по сжатию и предупреждения
--charset *кодировка*	Кодировка входного файла. В выходном файле будет использована такая же кодировка
--nomunge	Отключает замену имен локальных переменных
--disable-optimizations	Отключает микрооптимизацию
--preserve-semi	Предписывает оставить ненужные точки с запятой, которые иначе удаляются

Например, сжать файл `CookieUtil.js` в файл `cookie.js` можно следующим образом:

```
java -jar yuicompressor-2.3.5.jar -o cookie.js CookieUtil.js
```

YUI Compressor можно также вызвать непосредственно из средства Ant:

Листинг SampleAntDir/build.xml


```
<!-- Жюльен Леком, http://www.julienlecomte.net/blog/2007/09/16/ -->
<target name="js.compress">
    <apply executable="java" parallel="false">
        <fileset dir="${build.dir}" includes="output.js"/>
        <arg line="-jar"/>
        <arg path="${yuicompressor.jar}"/>
        <arg line="-o ${build.dir}/output-min.js"/>
        <srcfile/>
    </apply>
</target>
```

Здесь файл `output.js`, создаваемый во время сборки, передается в YUI Compressor. Выходным файлом является файл `output-min.js` в том же каталоге. В этом коде предполагается, что свойство `yuicompressor.jar` определяет расположение JAR-файла YUI Compressor. Запустить этот код можно с помощью следующей команды:

```
ant js.compress
```

Перед развертыванием в рабочей среде все JavaScript-файлы следует сжимать с помощью YUI Compressor или аналогичного средства. Чтобы сжатие файлов JavaScript выполнялось всегда, можно добавить этап сжатия в процесс сборки.

Сжатие HTTP-трафика

Объем трафика — это фактическое количество байтов, отправляемых сервером браузеру. Оно может не совпадать с размером кода из-за его сжатия на сервере. Все пять основных веб-браузеров — Internet Explorer, Firefox, Safari, Chrome и Opera — поддерживают декомпрессию полученных ресурсов, благодаря чему на сервере можно смело сжимать JavaScript-файлы. В отправляемый ответ сервер добавляет заголовок, указывающий, что файл был сжат в конкретном формате. Получив ответ, браузер изучает этот заголовок, выясняет, что файл был сжат, и восстанавливает его в исходном виде. Благодаря этому количество байтов, передаваемых по сети, значительно меньше, чем размер оригинального кода.

Для веб-сервера Apache доступны два модуля, позволяющие легко выполнять сжатие HTTP-трафика: `mod_gzip` (для Apache 1.3.x) и `mod_deflate` (для Apache 2.0.x). В случае `mod_gzip` вы можете включить автоматическое сжатие JavaScript-файлов, добавив в файл `httpd.conf` или `.htaccess` следующую строку:

```
#Указание сжимать все файлы с расширением .js
mod_gzip_item_include file \.js$
```

Эта строка предписывает модулю `mod_gzip` сжимать все запрошенные браузером файлы с расширением `.js`. Если все JavaScript-файлы имеют такое расширение, сжиматься будут все запросы, при этом в них добавляются заголовки, указывающие, что содержимое сжато. Дополнительные сведения о модуле `mod_gzip` доступны на сайте проекта по адресу www.sourceforge.net/projects/mod-gzip/.

Если используется модуль `mod_deflate`, включить сжатие JavaScript-файлов перед отправкой можно, добавив в файл `httpd.conf` или `.htaccess` следующую строку:

```
#Указание сжимать все JavaScript-файлы
AddOutputFilterByType DEFLATE application/x-javascript
```

Для определения того, нужно ли сжимать ответ, здесь используется его MIME-тип. Как уже отмечалось, несмотря на то что в качестве атрибута `type` элемента `<script>` используется значение `text/javascript`, JavaScript-файлы часто выдаются

с MIME-типом `application/x-javascript`. Дополнительные сведения о модуле `mod_deflate` доступны по адресу http://httpd.apache.org/docs/2.0/mod/mod_deflate.html.

Оба упомянутых модуля сжимают оригинальные JavaScript-файлы примерно на 70 %. Такая высокая эффективность сжатия достигается благодаря тому, что JavaScript-файлы содержат обычный текст. Сокращение объема трафика ускоряет доставку файлов браузеру, но при этом имеет место небольшой компромисс, потому что сервер тратит некоторое время на сжатие файлов, а браузер — на их декомпрессию. Как правило, выгода от сжатия перевешивает эти «накладные расходы».

Большинство веб-серверов — как с открытым исходным кодом, так и коммерческих — поддерживают сжатие HTTP-трафика. Сведения о том, как настроить сжатие на конкретном сервере, можно найти в его документации.

Резюме

По мере накопления опыта программирования на JavaScript были придуманы наилучшие методики. То, что когда-то считалось хобби, стало полноценным направлением разработки, потребовавшим изучения вопросов удобства сопровождения кода, его быстродействия, развертывания приложений и т. д.

Удобство сопровождения JavaScript-кода частично связано с конвенциями кодирования. Упомянем некоторые нюансы их применения:

- При использовании комментариев и отступов в JavaScript можно следовать конвенциям из других языков, однако слабо типизированная природа JavaScript требует также соблюдения некоторых специальных правил.
- Поскольку JavaScript используется вместе с HTML и CSS, важно четко разграничивать области применения этих технологий: код JavaScript должен определять поведение страницы, HTML — ее контент, а CSS — вид. Смешение этих областей может приводить к неуловимым ошибкам и проблемам сопровождения кода.

С увеличением объема JavaScript-кода в веб-приложениях одной из важнейших его характеристик стало быстродействие. В связи с этим желательно помнить о некоторых аспектах:

- Время выполнения JavaScript-кода напрямую влияет на общее быстродействие веб-страницы.
- Многие рекомендации по оптимизации кода для C-подобных языков относятся и к JavaScript. Примерами могут служить приемы оптимизации циклов и использование инструкций `switch` вместо `if`.
- На взаимодействие с DOM требуется много ресурсов, поэтому количество операций с DOM желательно свести к минимуму.

Логичным итогом разработки приложения является его развертывание. В связи с этим важно отметить следующее:

- Чтобы упростить развертывание, продумайте процесс сборки, объединяющий JavaScript-файлы в меньшее количество файлов (в идеале — один).
- Наличие процесса сборки позволяет автоматически запускать для исходного кода дополнительные процессы и фильтры. Например, вы можете запустить средство проверки JavaScript-кода, чтобы убедиться, что в нем нет синтаксических и других ошибок.
- Перед развертыванием JavaScript-кода рекомендуется уменьшить его размер с помощью средства сжатия.
- Дополнив сжатие кода сжатием HTTP-трафика, можно добиться максимальной экономии ресурсов сети при минимальном влиянии на быстродействие страницы.

25 Перспективные API

- Создание плавных анимаций
- Работа с файлами
- Фоновое выполнение JavaScript-кода с помощью рабочих веб-потоков

С выходом HTML5 начало появляться множество JavaScript API, которые не входят в спецификацию HTML5, а определены в отдельных спецификациях, часто называемых в совокупности «HTML5 Related APIs» (API, связанные с HTML5). Все API, описанные в этой главе, еще разрабатываются и не совсем стабильны.

Несмотря на это разработчики браузеров уже реализуют эти API, а разработчики веб-приложений начали их использовать. Многие из этих API имеют префиксы, специфичные для браузеров, такие как `"ms"` для Microsoft или `"webkit"` для Chrome и Safari. Это позволяет экспериментировать с новыми API, пока не появились их финальные версии без префиксов, поддерживаемые всеми браузерами.

requestAnimationFrame()

Долгое время для создания анимаций в JavaScript использовали таймеры и интервалы. CSS-переходы и CSS-анимации упростили для веб-разработчиков создание анимаций, но в JavaScript в этом плане годами почти ничего не менялось. Firefox 4 стал первым браузером, в котором был реализован новый JavaScript API для анимаций, а именно метод `mozRequestAnimationFrame()`. Он указывает браузеру, когда выполняется анимация, чтобы браузер мог наилучшим образом спланировать перерисовку страницы.

Ранние способы создания анимаций

При создании анимаций в JavaScript обычно используют метод `setInterval()`, с помощью которого управляют всеми анимациями. Базовый код анимации с помощью метода `setInterval()` выглядит следующим образом:

```
(function(){
    function updateAnimations(){
        doAnimation1();
        doAnimation2();
        // и т. д.
    }

    setInterval(updateAnimations, 100);
})();
```

Если анимаций несколько, метод `updateAnimations()` выполняет их в цикле, внося в каждую нужные изменения (например, обновляя бегущую строку с новостями и панель хода выполнения операции). Если обновлять анимации не требуется, метод может просто завершить работу и, возможно, даже остановить цикл, пока не потребуется изменить какую-либо из анимаций.

Самое непростое в этом цикле — это определение нужной задержки. Интервал должен быть достаточно коротким, чтобы все анимации были плавными, но и достаточно длинным, чтобы можно было подготовить изменения для визуализации браузером. Большинство компьютерных мониторов обновляют изображение с частотой 60 Гц, то есть оно перерисовывается 60 раз в секунду. В большинстве браузеров частота перерисовки ограничена этим значением, потому что ее повышение не улучшит впечатления пользователей.

Таким образом, наилучший интервал для максимально плавного вывода анимации — 1000/60 мс, или примерно 17 мс. Этот интервал лучше всего соответствует возможностям браузера. Если анимаций много, может потребоваться отрегулировать их, чтобы они не выполнялись слишком быстро.

Хотя циклы анимаций с методом `setInterval()` более эффективны, чем несколько циклов с методом `setTimeout()`, они не лишены недостатков. Методы `setInterval()` и `setTimeout()` не являются точными. Задержка, указываемая в качестве второго аргумента, определяет момент добавления кода в очередь потока пользовательского интерфейса, поэтому при наличии других заданий в очереди код будет запущен лишь через некоторое время. В общем, если поток пользовательского интерфейса занят, эта задержка не гарантирует, что код будет выполнен в назначенное время.

Проблемы с интервалами

Чтобы сделать анимацию плавной, нужно знать, когда будет нарисован следующий кадр, и до недавних пор это было невозможно. По мере роста популярности элемента

<canvas> и появления новых игр для браузеров неточность методов setInterval() и setTimeout() начала вызывать недовольство разработчиков.

Хуже того: показания таймеров в браузерах не представляются с точностью до миллисекунды. Вот точность таймеров в некоторых браузерах:

- в Internet Explorer 8 и более ранних версий — 15,625 мс;
- в Internet Explorer 9 и более поздних версий — 4 мс;
- в Firefox и Safari — примерно 10 мс;
- в Chrome — 4 мс.

В Internet Explorer до версии 9 точность таймера составляет 15,625 мс, так что любое значение между 0 и 15 изменяется на 0 или 15. В Internet Explorer 9 точность улучшена до 4 мс, но для анимаций этого все равно мало. В Chrome точность таймера составляет 4 мс, а в Firefox и Safari — 10 мс. Осложняет дело и то, что новые версии браузеров корректируют таймеры для вкладок, которые работают в фоновом режиме или неактивны. Таким образом, даже если вы задали интервал для оптимального отображения, интервалы будут лишь примерными.

mozRequestAnimationFrame

Роберт О'Каллахан (Robert O'Callahan) из Mozilla предложил уникальное решение проблемы анимаций. Он указал, что CSS-переходы и CSS-анимации работают эффективнее, если браузер знает о выполняемой анимации, и вычислил правильный интервал обновления пользовательского интерфейса. Когда выполняется JavaScript-анимация, браузер не знает об этом. О'Каллахан создал метод mozRequestAnimationFrame(), указывающий браузеру, что некоторый JavaScript-код выполняет анимацию. Это позволяет браузеру оптимизировать обработку после выполнения некоторого кода.

Метод mozRequestAnimationFrame() принимает в качестве единственного аргумента функцию, вызываемую перед перерисовкой экрана. В этой функции вы вносите в DOM-стили изменения, которые будут отражены при следующей перерисовке. Чтобы создать цикл анимации, можно объединить несколько вызовов метода mozRequestAnimationFrame() подобно тому, как это делается для метода setTimeout(), например:

```
function updateProgress(){
    var div = document.getElementById("status");
    div.style.width = (parseInt(div.style.width, 10) + 5) + "%";

    if (div.style.left != "100%"){
        mozRequestAnimationFrame(updateProgress);
    }
}

mozRequestAnimationFrame(updateProgress);
```

Поскольку метод `mozRequestAnimationFrame()` выполняет указанную функцию только один раз, для следующего изменения пользовательского интерфейса в ходе анимации его нужно вызвать вручную. Точно так же нужно следить за временем прекращения анимации. С помощью этого метода можно сделать анимацию плавной.

Метод `mozRequestAnimationFrame()` решает проблемы отсутствия информации о том, когда выполняется анимация и каков оптимальный интервал перерисовки, а что с проблемой точного времени запуска кода? Она имеет то же решение.

Функция, которую вы передаете в метод `mozRequestAnimationFrame()`, на самом деле принимает аргумент, представляющий метку времени (в миллисекундах, прошедших с 1 января 1970 года), когда страница должна быть перерисована в следующий раз. Это очень важно: метод `mozRequestAnimationFrame()` на самом деле планирует перерисовку на какой-то известный момент в будущем и позволяет узнать, когда он наступит. Так вы можете определить, как лучше всего настроить свою анимацию.

Чтобы определить, сколько времени прошло с момента последней перерисовки, вы можете запросить значение `mozAnimationStartTime`, содержащее метку времени последней перерисовки в указанном выше формате. Вычтя это значение из времени, переданного в функцию обратного вызова, можно точно узнать, сколько времени осталось до вывода на экран следующего набора изменений. Типичный шаблон использования этих значений таков:

```
function draw(timestamp){

    // вычисление времени, прошедшего с момента последней перерисовки
    var diff = timestamp - startTime;

    // использование значения diff для определения следующего действия

    // установка значения startTime
    startTime = timestamp;

    // очередная операция рисования
    mozRequestAnimationFrame(draw);
}

var startTime = mozAnimationStartTime;
mozRequestAnimationFrame(draw);
```

Главное в этом коде — первый доступ к значению `mozAnimationStartTime` вне функции обратного вызова, передаваемой в метод `mozRequestAnimationFrame()`. Если обратиться к значению `mozAnimationStartTime` внутри функции обратного вызова, оно будет содержать время, переданное в функцию в качестве аргумента.

webkitRequestAnimationFrame и msRequestAnimationFrame

Разработчики Chrome и Internet Explorer 10+ создали собственные реализации метода `mozRequestAnimationFrame()`, которые называются `webkitRequestAnimationFrame()`

и `msRequestAnimationFrame()` соответственно. Эти версии отличаются от версии Firefox в двух отношениях. Во-первых, в функцию обратного вызова не передается метка времени, так что вы не знаете, когда случится следующая перерисовка. Во-вторых, в Chrome добавлен второй необязательный аргумент — изменяемый DOM-элемент. Если нужно изменить единственный элемент на странице, можно ограничить перерисовку только его областью.

Неудивительно, что эквивалента значения `mozAnimationStartTime` в этих браузерах нет, поскольку без времени следующей перерисовки от этой информации пользы мало. В Chrome, однако, есть метод `webkitCancelAnimationFrame()`, который отменяет ранее запланированную перерисовку.

Если не нужно придерживаться точных временных интервалов, вы можете создать цикл анимации для Firefox 4+, Internet Explorer 10+ и Chrome следующим образом:

```
(function(){

    function draw(timestamp){

        // вычисление времени, прошедшего с момента последней перерисовки
        var drawStart = (timestamp || Date.now()),
            diff = drawStart - startTime;

        // использование значения diff для определения следующего действия

        // установка значения startTime
        startTime = drawStart;

        // очередная операция рисования
        requestAnimationFrame(draw);
    }

    var requestAnimationFrame = window.requestAnimationFrame ||
                                window.mozRequestAnimationFrame ||
                                window.webkitRequestAnimationFrame ||
                                window.msRequestAnimationFrame,
        startTime = window.mozAnimationStartTime || Date.now();
    requestAnimationFrame(draw);
})();
```

Здесь мы создаем цикл анимации с учетом сведений о прошедшем времени. В Firefox для этого используется доступная метка времени, а в Chrome и Internet Explorer по умолчанию применяется менее точный объект `Date`. При использовании этого шаблона вы получаете общее представление о прошедшем интервале времени, но не можете узнать, когда произойдет следующая перерисовка в Chrome или Internet Explorer. Тем не менее некоторое представление о прошедшем интервале времени лучше, чем никакого.

Проверяя первым стандартное имя функции, а затем имена, специфичные для браузеров, мы гарантируем, что этот цикл анимации продолжит работать и в будущем.

В настоящее время API `requestAnimationFrame()` совместно разрабатывается Mozilla и Google в составе Web Performance Group. Консорциум W3C собирается сделать его новой рекомендацией.

Page Visibility API

Веб-разработчикам часто требуется узнать, когда пользователь на самом деле взаимодействует со страницей. Если страница свернута или скрыта за другой вкладкой, возможно, не имеет смысла опрашивать сервер на предмет наличия обновлений или продолжать анимацию. Page Visibility API (API видимости страниц) позволяет разработчикам узнать, видима ли страница.

Сам API очень прост и состоит из трех элементов:

- `document.hidden` — логическое значение, указывающее, скрыта ли страница (это может означать, что она открыта на фоновой вкладке или окно браузера свернуто).
- `document.visibilityState` — значение, представляющее одно из четырех состояний:
  - страница открыта на фоновой вкладке или окно браузера свернуто;
  - страница открыта на вкладке переднего плана;
  - сама страница скрыта, но отображается окно предварительного просмотра страницы (например, в Windows 7 его можно увидеть, наведя указатель мыши на ярлык страницы на панели задач);
  - выполняется предварительная визуализация страницы вне экрана.
- `visibilitychange` — это событие генерируется, когда состояние документа меняется со скрытого на видимое или наоборот.

На день написания этой главы Page Visibility API был реализован только в Internet Explorer 10 и Chrome. В Internet Explorer имена всех элементов API имеют префикс `"ms"`, а в Chrome — `"webkit"`. Таким образом, свойство `document.hidden` доступно как `document.msHidden` в Internet Explorer и как `document.webkitHidden` в Chrome. Проверить, поддерживается ли одно из них, можно следующим образом:

Листинг PageVisibilityAPIExample01.htm


```
function isHiddenSupported(){
    return typeof (document.hidden || document.msHidden ||
                   document.webkitHidden) != "undefined";
}
```

Аналогично можно проверить, скрыта ли страница:

Листинг PageVisibilityAPIExample01.htm

```
if (document.hidden || document.msHidden || document.webKitHidden){
    // страница скрыта
} else {
    // страница не скрыта
}
```

Если браузер не поддерживает данное свойство, считается, что страница не скрыта. Это сделано для обеспечения обратной совместимости.

Чтобы узнать, когда страница изменяет состояние с видимого на скрытое или наоборот, можно прослушать событие `visibilitychange`. В Internet Explorer оно называется `msvisibilitychange`, а в Chrome — `webkitvisibilitychange`. Для обработки изменения состояния страницы в обоих браузерах нужно назначить один обработчик обоим этим событиям, например:

Листинг PageVisibilityAPIExample01.htm

```
function handleVisibilityChange(){
    var output = document.getElementById("output"),
        msg;

    if (document.hidden || document.msHidden || document.webkitHidden){
        msg = "Page is now hidden." + (new Date()) + "<br>";
    } else {
        msg = "Page is now visible." + (new Date()) + "<br>";
    }

    output.innerHTML += msg;

}

// добавление обработчиков событий
EventUtil.addHandler(document, "msvisibilitychange",
    handleVisibilityChange);
EventUtil.addHandler(document, "webkitvisibilitychange",
    handleVisibilityChange);
```

Этот код работает и в Internet Explorer, и в Chrome. Данная часть API сравнительно стабильна, так что вы можете безопасно использовать этот код в реальных веб-приложениях.

Больше всего в разных реализациях API различается свойство `document.visibilityState`. В Internet Explorer 10 PR 2 оно содержит одну из четырех числовых констант:

- `document.MS_PAGE_HIDDEN` (0);
- `document.MS_PAGE_VISIBLE` (1);
- `document.MS_PAGE_PREVIEW` (2);
- `document.MS_PAGE_PRERENDER` (3).

В Chrome свойство `document.webkitVisibilityState` может содержать одно из трех строковых значений:

- `hidden`;
- `visible`;
- `prerender`.

В Chrome константы состояний не определены, хотя в окончательной реализации они, вероятно, будут доступны.

Из-за этих различий рекомендуется не использовать свойство `document.visibilityState` с префиксом производителя браузера, а вместо этого полагаться на свойство `document.hidden`.

Geolocation API

Geolocation (*API геолокации*) — один из наиболее интересных и хорошо поддерживаемых новых API, с помощью которого можно получать информацию о текущем местоположении пользователя. Конечно, это возможно, только если пользователь явно разрешит веб-странице доступ к данным. Когда страница пытается обратиться к ним, браузер выводит диалоговое окно с запросом разрешения. На рис. 25.1 показано такое диалоговое окно в Chrome.

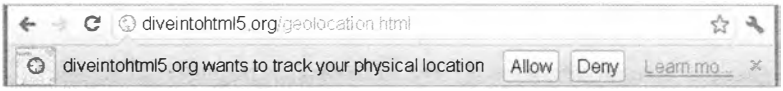

Рис. 25.1

Geolocation реализован как свойство `navigator.geolocation` с тремя методами. Метод `getCurrentPosition()` выводит диалоговое окно с запросом разрешения на доступ к данным геолокации. Он принимает три аргумента: функцию обратного вызова, которая вызывается в случае успешного запроса, необязательную функцию обратного вызова, вызываемую при неудаче, и необязательный объект с параметрами.

Первая функция обратного вызова принимает в качестве единственного аргумента объект `Position`, у которого есть свойства `coords` и `timestamp`. Объект `coords` должен содержать следующую информацию о расположении:

- `latitude` — широта в градусах;
- `longitude` — долгота в градусах;
- `accuracy` — точность координат в метрах (чем больше это значение, тем меньше точность).

Браузер также может добавлять следующие свойства:

- `altitude` — высота в метрах или значение `null`, если эти сведения недоступны;
- `altitudeAccuracy` — точность высоты в метрах (чем больше это значение, тем меньше точность);
- `heading` — направление по компасу в градусах, при этом 0 представляет север (если определить направление невозможно, возвращается `NaN`);
- `speed` — скорость в метрах в секунду или значение `null`, если определить скорость невозможно.

На практике в большинстве веб-приложений свойства `latitude` и `longitude` используются чаще остальных, например для рисования расположения пользователя на карте:

```
navigator.geolocation.getCurrentPosition(function(position){
    drawMapCenteredAt(position.coords.latitude, positions.coords.longitude);
});
```

Функция обратного вызова, вызываемая в случае неудачи, также принимает аргумент — объект со свойствами `message` и `code`. Свойству `message` назначается понятное людям сообщением с описанием причины ошибки, а свойство `code` представляет собой числовой код ошибки: пользователю отказано в разрешении (1), сведения о расположении недоступны (2) или тайм-аут (3). На практике большинство веб-приложений просто регистрируют такие ошибки, никак не изменяя пользовательский интерфейс, например:

```
navigator.geolocation.getCurrentPosition(function(position){
    drawMapCenteredAt(position.coords.latitude, positions.coords.longitude);
}, function(error){
    // Код ошибки
    console.log("Error code: " + error.code);
    // Сообщение об ошибке
    console.log("Error message: " + error.message);
});
```

Третьим аргументом метода `getCurrentPosition()` является объект `options`, с помощью которого можно задать три параметра: логическое значение `enableHighAccuracy`, запрашивающее наиболее точное расположение, `timeout`, или время ожидания сведений о расположении в миллисекундах, и `maximumAge`, или допустимое время использования последних координат до определения нового расположения (в миллисекундах), например:

```
navigator.geolocation.getCurrentPosition(function(position){
    drawMapCenteredAt(position.coords.latitude, positions.coords.longitude);
}, function(error){
    // Код ошибки
```

```
    console.log("Error code: " + error.code);
    // Сообщение об ошибке
    console.log("Error message: " + error.message);
}, {
    enableHighAccuracy: true,
    timeout: 5000,
    maximumAge: 25000
});
```

Все три параметра не обязательны и могут быть указаны по отдельности или вместе с другими. Если очень точные сведения вам не требуются, рекомендуется оставить у параметра `enableHighAccuracy` значение `false` (значение, предлагаемое по умолчанию). Изменив его на `true`, вы можете замедлить работу кода и увеличить энергопотребление мобильного устройства. Если вам не нужно активно отслеживать расположение пользователя, можно присвоить свойству `maximumAge` значение `Infinity`, чтобы всегда использовались последние координаты.

Для отслеживания расположения пользователя можно также задействовать метод `watchPosition()`, который принимает те же аргументы, что и `getCurrentPosition()`. На практике вызов `watchPosition()` эквивалентен периодическому вызову `getCurrentPosition()`. При первом вызове метод `watchPosition()` получает текущее расположение и запускает одну из двух функций обратного вызова. После этого он дожидается от системы сигнала об изменении расположения (сам он эти данные не запрашивает).

Метод `watchPosition()` возвращает числовой идентификатор, используемый для отслеживания операции наблюдения. С помощью этого значения можно отменить наблюдение, передав его в метод `clearWatch()` (это похоже на работу с методами `setTimeout()` и `clearTimeout()`). Вот пример:

```
var watchId = navigator.geolocation.watchPosition(function(position){
    drawMapCenteredAt(position.coords.latitude, positions.coords.longitude);
}, function(error){
    // Код ошибки
    console.log("Error code: " + error.code);
    // Сообщение об ошибке
    console.log("Error message: " + error.message);
});

clearWatch(watchId);
```

Здесь мы вызываем метод `watchPosition()`, сохраняя возвращенный идентификатор в переменной `watchId`. Позднее мы передаем ее в метод `clearWatch()` для отмены наблюдения.

Геолокация поддерживается в Internet Explorer 9+, Firefox 3.5+, Opera 10.6+, Safari 5+, Chrome, Safari для iOS и WebKit для Android. Прекрасный пример геолокации доступен по адресу http://html5demos.com/geo.

File API

При разработке веб-приложений много затруднений возникает из-за отсутствия средств взаимодействия с файлами на компьютере пользователя. Все, что прежде можно было сделать, это добавить в форму тег `<input type="file">`. Для безопасной работы с файлами на клиентском компьютере и предназначен File API, который поддерживается в Internet Explorer 10+, Firefox 4+, Safari 5.0.5+, Opera 11.1+ и Chrome.

В File API по-прежнему используется поле форм для ввода файлов, но при этом в нем поддерживается прямой доступ к файлам. В HTML5 к полю ввода файла в DOM добавлен набор `files`. При выборе одного или нескольких файлов в этом поле набору `files` назначаются объекты `File`, представляющие каждый из файлов. У каждого объекта `File` есть несколько свойств, доступных только для чтения, в том числе следующие:

- `name` — имя файла в локальной системе;
- `size` — размер файла в байтах;
- `type` — строка, содержащая MIME-тип файла;
- `lastModifiedDate` — строка, представляющая время последнего изменения файла (это свойство реализовано только в Chrome).

Например, получить информацию о каждом выбранном файле можно, просмотрев набор `files` в обработчике события `change`:

Листинг FileAPIExample01.htm


```
var filesList = document.getElementById("files-list");
EventUtil.addHandler(filesList, "change", function(event){
    var files = EventUtil.getTarget(event).files,
        i = 0,
        len = files.length;

    while (i < len){
        console.log(files[i].name + " (" + files[i].type + ", " +
                    files[i].size + " bytes)");
        i++;
    }
});
```

Этот код просто выводит на консоли информацию о каждом файле. Это само по себе представляет большой прогресс, но File API доступны и более мощные возможности, такие как чтение данных из файлов с помощью типа `FileReader`.

Тип FileReader

Тип `FileReader` служит для асинхронного чтения файлов. Можете думать о нем как об аналоге типа `XMLHttpRequest`, только он используется для чтения файлов из

файловой системы, а не для получения данных с сервера. Тип `FileReader` поддерживает следующие методы чтения:

- `readAsText(файл, кодировка)` — читает файл как обычный текст, сохраняя его в свойстве `result` (второй аргумент, задающий тип кодировки, не обязателен);
- `readAsDataURL(файл)` — читает файл, сохраняя URI данных в свойстве `result`;
- `readAsBinaryString(файл)` — читает файл, сохраняя в свойстве `result` строку, в которой каждый знак представляет байт;
- `readAsArrayBuffer(файл)` — читает файл, сохраняя в свойстве `result` объект `ArrayBuffer` с содержимым файла.

Поддержка нескольких способов чтения файлов обеспечивает максимальную гибкость при работе с ними. Например, вы можете прочитать изображение как URI данных, чтобы показать его пользователю, или прочитать файл как текст, чтобы можно было разобрать его на лексемы.

Поскольку чтение выполняется асинхронно, каждый объект `FileReader` поддерживает несколько свойств. Наиболее полезны из них свойства `progress`, `error` и `load`, которые соответственно уведомляют, когда доступны дополнительные данные, когда произошла ошибка и когда файл прочитан полностью.

Событие `progress` генерируется примерно каждые 50 мс и предоставляет те же сведения, что и событие `progress` объекта XHR, а именно — свойства `lengthComputable`, `loaded` и `total`. Кроме того, при обработке события `progress` можно прочитать свойство `result` объекта `FileReader`, хотя в это время оно еще может содержать не все данные.

Событие `error` генерируется, если по какой-либо причине не удается прочитать файл, при этом задается свойство `error` объекта `FileReader`. У этого объекта есть единственное свойство `code`, которое содержит код ошибки: 1 (файл не найден), 2 (ошибка безопасности), 3 (чтение было прервано), 4 (файл не может быть прочитан) или 5 (ошибка кодировки).

Событие `load` генерируется при успешной загрузке файла; если возникло событие `error`, оно не генерируется. Вот пример со всеми тремя событиями:

Листинг FileAPIExample02.htm


```
var filesList = document.getElementById("files-list");
EventUtil.addHandler(filesList, "change", function(event){
    var info = "",
        output = document.getElementById("output"),
        progress = document.getElementById("progress"),
        files = EventUtil.getTarget(event).files,
        type = "default",
        reader = new FileReader();

    if (/image/.test(files[0].type)){
```

```
        reader.readAsDataURL(files[0]);
        type = "image";
    } else {
        reader.readAsText(files[0]);
        type = "text";
    }

    reader.onerror = function(){
        // Не удалось прочитать файл
        output.innerHTML = "Could not read file, error code is " +
                           reader.error.code;
    };

    reader.onprogress = function(event){
        if (event.lengthComputable){
            progress.innerHTML = event.loaded + "/" + event.total;
        }
    };

    reader.onload = function(){

        var html = "";

        switch(type){
            case "image":
                html = "<img src=\"" + reader.result + "\">";
                break;
            case "text":
                html = reader.result;
                break;

        }
        output.innerHTML = html;
    };
});
```

Этот код читает файл, указанный в поле формы, и отображает его на странице. Если MIME-типом файла является изображение, запрашивается URI данных, который при обработке события `load` вставляется на страницу как изображение. Если файл не является изображением, он читается как строка и выводится на странице как есть. Событие `progress` используется для отслеживания и отображения читаемых данных, а событие `error` — для контроля ошибок.

Чтение можно остановить, вызвав метод `abort()`, при этом генерируется событие `abort`. После события `load`, `error` или `abort` генерируется событие `loadend`, которое указывает, что чтение было завершено по одной из трех этих причин.

Методы `readAsText()` и `readAsDataURL()` поддерживаются во всех браузерах, реализующих тип `FileReader`. В Internet Explorer 10 PR2 не реализованы методы `readAsBinaryString()` и `readAsArrayBuffer()`.

Частичное чтение

Иногда желательно прочитать не весь файл, а только его части. Для этого объект `File` предоставляет метод `slice()`, который называется `mozSlice()` в Firefox и `webkitSlice()` в Chrome; в Safari 5.1 он не реализован. Метод `slice()` принимает два аргумента: начальный байт и количество байтов, которые нужно прочитать. Возвращает он экземпляр типа `Blob`, который является родительским по отношению к `File`. Следующая функция унифицирует работу с методом `slice()` в разных браузерах:

Листинг FileAPIExample03.htm


```
function blobSlice(blob, startByte, length){
    if (blob.slice){
        return blob.slice(startByte, length);
    } else if (blob.webkitSlice){
        return blob.webkitSlice(startByte, length);
    } else if (blob.mozSlice){
        return blob.mozSlice(startByte, length);
    } else {
        return null;
    }
}
```

У типа `Blob` есть свойства `size` и `type`, а также метод `slice()` для обрезки данных. Кроме того, читать данные из него можно с помощью объекта `FileReader`. Следующий код читает из файла начальные 32 байта:

Листинг FileAPIExample03.htm


```
var filesList = document.getElementById("files-list");
EventUtil.addHandler(filesList, "change", function(event){
    var info = "",
        output = document.getElementById("output"),
        progress = document.getElementById("progress"),
        files = EventUtil.getTarget(event).files,
        reader = new FileReader(),
        blob = blobSlice(files[0], 0, 32);

    if (blob){
        reader.readAsText(blob);

        reader.onerror = function(){
        // Не удалось прочитать файл
            output.innerHTML = "Could not read file, error code is " +
                               reader.error.code;
        };

        reader.onload = function(){
            output.innerHTML = reader.result;
        };
    } else {
```

```
        // Браузер не поддерживает метод slice()
        alert("Your browser doesn't support slice().");
    }
});
```

Частичное чтение файла позволяет сэкономить время, особенно если вы просто ищете в нем конкретные данные, например заголовок.

URL-адреса объектов

URL-адреса объектов, или *URL-адреса больших двоичных объектов* (blob URLs), — это адреса, которые ссылаются на данные в объектах `File` или `Blob`. Преимущество URL-адресов объектов в том, что для работы с файлами в JavaScript вам не нужно читать их содержимое — вместо этого вы просто указываете в нужном месте URL-адрес объекта. Создать такой URL-адрес можно методом `window.URL.createObjectURL()`, передав в него объект `File` или `Blob`. В Chrome этот метод реализован как `window.webkitURL.createObjectURL()`, так что для унификации кода можно использовать следующую функцию:

Листинг FileAPIExample04.htm

```
function createObjectURL(blob){
    if (window.URL){
        return window.URL.createObjectURL(blob);
    } else if (window.webkitURL){
        return window.webkitURL.createObjectURL(blob);
    } else {
        return null;
    }
}
```

Эта функция возвращает строку, которая указывает на адрес памяти. Поскольку строка представляется URL-адресом, ее можно использовать в DOM. Например, следующий код отображает файл изображения на странице:

Листинг FileAPIExample04.htm


```
var filesList = document.getElementById("files-list");
EventUtil.addHandler(filesList, "change", function(event){
    var info = "",
        output = document.getElementById("output"),
        progress = document.getElementById("progress"),
        files = EventUtil.getTarget(event).files,
        reader = new FileReader(),
        url = createObjectURL(files[0]);

    if (url){
        if (/image/.test(files[0].type)){
            output.innerHTML = "<img src=\"" + url + "\">";
        } else {
            // Файл не является изображением
```

```
            output.innerHTML = "Not an image.";
        }
    } else {
        // Браузер не поддерживает URL-адреса объектов
        output.innerHTML = "Your browser doesn't support object URLs.";
    }
});
```

Добавляя URL-адрес объекта непосредственно в тег `<img>`, мы можем не считывать данные в JavaScript. Вместо этого они считываются на страницу непосредственно из памяти.

Когда данные больше не нужны, лучше освободить занимаемую ими память. Пока URL-адрес объекта используется, сделать это нельзя. Вы можете указать, что URL-адрес объекта больше не требуется, передав его в метод `window.URL.revokeObjectURL()` (`window.webkitURL.revokeObjectURL()` в Chrome). Чтобы охватить оба варианта, используйте такую функцию:

```
function revokeObjectURL(url){
    if (window.URL){
        window.URL.revokeObjectURL(url);
    } else if (window.webkitURL){
        window.webkitURL.revokeObjectURL(url);
    }
}
```

При выгрузке страницы память с URL-адресами объектов освобождается автоматически. Тем не менее, чтобы страница занимала как можно меньше памяти, лучше делать это для URL-адреса каждого объекта вручную, когда он становится больше не нужен.

URL-адреса объектов поддерживаются в Internet Explorer 10+, Firefox 4+ и Chrome.

Чтение файлов и перетаскивание

Объединение Drag-and-Drop API из HTML5 и File API позволяет создавать интересные интерфейсы для чтения информации о файлах. Так, создав собственный целевой элемент на странице, вы можете перетаскивать на него файлы, при этом, как и при перетаскивании изображения или ссылки, будет генерироваться событие `drop`. Отпускаемые файлы доступны в свойстве `event.dataTransfer.files`, которое представляет собой список объектов `File`, как и в поле ввода файлов.

Следующий код выводит информацию о файлах, перетаскиваемых на пользовательский целевой элемент на странице:

Листинг FileAPIExample05.htm


```
var droptarget = document.getElementById("droptarget");

function handleEvent(event){
```

```
    var info = "",
        output = document.getElementById("output"),
        files, i, len;

    EventUtil.preventDefault(event);

    if (event.type == "drop"){
        files = event.dataTransfer.files;
        i = 0;
        len = files.length;

        while (i < len){
            info += files[i].name + " (" + files[i].type + ", " +
                    files[i].size + " Б)<br>";
            i++;
        }

        output.innerHTML = info;
    }
}

EventUtil.addHandler(droptarget, "dragenter", handleEvent);
EventUtil.addHandler(droptarget, "dragover", handleEvent);
EventUtil.addHandler(droptarget, "drop", handleEvent);
```

Как и в более ранних примерах с перетаскиванием, вы должны отменить поведение, предлагаемое по умолчанию для событий `dragenter`, `dragover` и `drop`. Во время обработки события `drop` файлы доступны в свойстве `event.dataTransfer.files` и вы можете прочитать их. Этот функционал часто используется для отправки файлов с помощью объекта `XMLHttpRequest`.

Отправка файлов с помощью объекта XHR

Так как File API предоставляет доступ к содержимому файлов, вы можете использовать его для отправки файлов на сервер с помощью объекта XHR. Естественно, это можно сделать, передав данные в метод `send()` и выполнив POST-запрос, но тогда данные пришлось бы принимать на сервере и сохранять в другом файле. В идеале лучше было бы выгружать файл так, как будто он передается вместе с формой.

Это можно довольно легко сделать с помощью типа `FormData`, представленного в главе 21. Создайте объект `FormData` и передайте в его метод `append()` объект `File`. Затем передайте объект `FormData` в метод `send()` объекта XHR, и вы успешно сымитируете отправку файла с помощью формы:

Листинг FileAPIExample06.htm


```
var droptarget = document.getElementById("droptarget");

function handleEvent(event){
    var info = "",
```

```
        output = document.getElementById("output"),
        data, xhr,
        files, i, len;

    EventUtil.preventDefault(event);

    if (event.type == "drop"){
        data = new FormData();
        files = event.dataTransfer.files;
        i = 0;
        len = files.length;

        while (i < len){
            data.append("file" + i, files[i]);
            i++;
        }

        xhr = new XMLHttpRequest();
        xhr.open("post", "FileAPIExample06Upload.php", true);
        xhr.onreadystatechange = function(){
            if (xhr.readyState == 4){
                alert(xhr.responseText);
            }
        };
        xhr.send(data);
    }
}

EventUtil.addHandler(droptarget, "dragenter", handleEvent);
EventUtil.addHandler(droptarget, "dragover", handleEvent);
EventUtil.addHandler(droptarget, "drop", handleEvent);
```

Этот код создает объект `FormData` с ключами для всех файлов, таких как `file0`, `file1` и `file2`. Указывать файлы как значения формы не требуется. Использовать объект `FileReader` также не нужно — достаточно передать сам объект `File`.

Использование объекта `FormData` для отправки файлов полностью имитирует работу обычной формы на сервере. Это означает, что на сервере по-прежнему доступны полезные объекты, такие как PHP-массив `$_FILES`. Объект `FormData` и отправка файлов с его помощью поддерживаются в Firefox 4+, Safari 5+ и Chrome.

Web Timing API

Веб-разработчики всегда уделяли особое внимание быстродействию страниц. До недавнего времени для его оценки приходилось прибегать к нетривиальному использованию объекта `Date`. Web Timing API меняет ситуацию, предоставляя непосредственный доступ к внутренним метрикам браузера. В отличие от других API, описываемых в этой главе, он уже рекомендован консорциумом W3C, хотя разработчики браузеров не торопятся добавлять его.

В основе Web Timing API лежит объект `window.performance`. Все метрики, связанные со страницей, — как уже имеющиеся, так и те, что будут добавлены в будущем, — содержатся в нем. Спецификация Web Timing начинается с определения двух свойств объекта `performance`.

Свойство `performance.navigation` представляет собой объект, который содержит несколько свойств, связанных с навигацией:

- `redirectCount` — количество перенаправлений, выполненных перед загрузкой страницы.
- `type` — числовая константа, представляющая тип навигации:
  - `performance.navigation.TYPE_NAVIGATE` (0) — страница загружена впервые;
  - `performance.navigation.TYPE_RELOAD` (1) — страница перезагружена;
  - `performance.navigation.TYPE_BACK_FORWARD` (2) — страница открыта в результате щелчка на кнопке Back (Назад) или Forward (Вперед).

Свойство `performance.timing` содержит ряд меток времени возникновения различных событий (в миллисекундах, прошедших с начала эпохи):

- `navigationStart` — время начала перехода на страницу.
- `unloadEventStart` — время начала события `unload` для предыдущей страницы. Это свойство заполняется, только если предыдущая страница имеет тот же источник, что и новая, в противном случае оно равно нулю.
- `unloadEventEnd` — время завершения события `unload` для предыдущей страницы. Это свойство заполняется, только если предыдущая страница имеет тот же источник, что и новая, в противном случае оно равно нулю.
- `redirectStart` — время начала перенаправления на текущую страницу, но только если оно происходит без изменения источника, в противном случае это значение равно нулю.
- `redirectEnd` — время завершения перенаправления на текущую страницу, но только если оно происходит без изменения источника, в противном случае это значение равно нулю.
- `fetchStart` — время начала получения страницы с помощью HTTP-метода GET.
- `domainLookupStart` — время начала просмотра DNS для страницы.
- `domainLookupEnd` — время завершения просмотра DNS для страницы.
- `connectStart` — время, когда браузер попытался подключиться к серверу.
- `connectEnd` — время, когда браузеру удалось подключиться к серверу.
- `secureConnectionStart` — время, когда браузер попытался установить SSL-подключение. Если протокол SSL не используется, это значение равно нулю.
- `requestStart` — время, когда браузер начал запрашивать страницу.

- `responseStart` — время, когда браузер получил первый байт страницы.
- `responseEnd` — время, когда браузер получил всю страницу.
- `domLoading` — время изменения значения `document.readyState` на `"loading"`.
- `domInteractive` — время изменения значения `document.readyState` на `"interactive"`.
- `domContentLoadedEventStart` — время перед генерированием события `DOMContentLoaded`.
- `domContentLoadedEventEnd` — время, когда событие `DOMContentLoaded` произошло и все обработчики выполнились.
- `domComplete` — время изменения значения `document.readyState` на `"complete"`.
- `loadEventStart` — время перед генерированием события `load`.
- `loadEventEnd` — время, когда событие `load` произошло и все обработчики выполнились.

Анализируя разницу между указанными моментами времени, можно получить неплохое представление о том, как страница загружается в браузер и каковы возможные проблемы с быстродействием. Прекрасный пример использования Web Timing API доступен по адресу http://webtimingdemo.appspot.com/.

Web Timing API поддерживается в Internet Explorer 10+ и Chrome.

Рабочие веб-потоки

По мере возрастания сложности веб-приложений нужно выполнять все больше сложных вычислений. Длительные JavaScript-процессы блокируют пользовательский интерфейс браузера, что иногда воспринимается как «зависание». Рабочие веб-потоки (Web Workers) решают эту проблему, выполняя JavaScript-код в фоновом режиме. В браузерах они могут быть реализованы разными способами: с помощью потоков, фоновых процессов или процессов, выполняемых другими ядрами процессора. На самом деле детали их реализации не важны, важна возможность выполнять JavaScript-код без блокирования пользовательского интерфейса.

В настоящее время рабочие веб-потоки поддерживаются в Internet Explorer 10+, Firefox 3.5+, Safari 4+, Opera 10.6+, Chrome и Safari для iOS 5.

Использование рабочего веб-потока

Вы можете создать рабочий веб-поток, вызвав конструктор объекта `Worker` и передав в него имя файла с JavaScript-кодом, который нужно выполнить, например:

```
var worker = new Worker("stufftodo.js");
```

Эта строка кода дает браузеру команду загрузить файл `stufftodo.js`, но рабочий веб-поток не начинает работу, пока не получит сообщение. Передать ему сообщение можно методом `postMessage()` (подобно XDM):

```
worker.postMessage("start!");
```

Сообщением может быть любое сериализуемое значение, хотя, в отличие от XDM, во всех браузерах, поддерживающих рабочие веб-потоки, метод `postMessage()` может также принимать объекты (Safari 4 был последним браузером, в котором рабочие веб-потоки поддерживали только строковые сообщения). Таким образом, вы можете свободно передавать в этот метод данные в любых объектах, например:

```
worker.postMessage({
    type: "command",
    message: "start!"
});
```

Вообще говоря, любые значения, которые могут быть сериализованы в JSON-структуры, также можно передавать в метод `postMessage()`. Это означает, что как и в случае XDM, значения копируются в рабочие веб-потоки, а не передаются непосредственно.

Веб-поток взаимодействует со страницей с помощью двух событий: `message` и `error`. Событие `message` работает так же, как и в XDM, при этом данные от веб-потока поступают в виде свойства `event.data`. Они также могут быть любым сериализуемым значением:

```
worker.onmessage = function(event){
    var data = event.data;

    // какие-то действия с данными
}
```

Если рабочему веб-потоку не удалось выполнить указанную задачу, он сообщает об этом с помощью события `error`. Оно генерируется, если при выполнении JavaScript-кода в веб-потоке происходит ошибка. У объекта `event` события `error` есть три свойства: `filename` — имя файла с ошибкой, `lineno` — номер строки с ошибкой в этом файле и `message` — полное сообщение об ошибке.

```
worker.onerror = function(event){
    // Ошибка
    console.log("ERROR: " + event.filename + " (" + event.lineno + "): " +
                event.message);
};
```

При использовании рабочего веб-потока имеет смысл всегда создавать обработчик события `error`, даже если он ничего не делает, а только регистрирует ошибку. В противном случае ошибки будут оставаться незамеченными.

Вы можете в любое время остановить рабочий веб-поток, вызвав метод `terminate()`. При этом веб-поток немедленно завершается, а оставшаяся работа не выполняется (события `error` и `message` при этом не генерируются).

```
worker.terminate();        // немедленная остановка рабочего веб-потока
```

Глобальная область видимости рабочего веб-потока

Самое важное, что нужно знать о рабочем веб-потоке, это то, что его JavaScript-код выполняется в совершенно другой области видимости, чем код веб-страницы. У рабочего веб-потока другой глобальный объект и другие объекты и методы. Внутри него нет доступа к DOM, и никак нельзя повлиять на вид страницы.

Глобальным объектом внутри рабочего веб-потока является сам объект `worker`. Это означает, что при доступе к объекту `this` или `self` в глобальной области видимости осуществляется доступ к объекту `worker`. Внутри него также есть минимальная среда, позволяющая ему обрабатывать данные:

- минимальный объект `navigator` со свойствами `onLine`, `appName`, `appVersion`, `userAgent` и `platform`;
- объект `location`, доступный только для чтения;
- методы `setTimeout()`, `setInterval()`, `clearTimeout()` и `clearInterval()`;
- конструктор `XMLHttpRequest`.

Как видите, среда рабочего веб-потока довольно ограниченна в сравнении со средой страницы.

Когда страница вызывает метод `postMessage()` объекта `worker`, ему асинхронно передаются данные, что приводит к генерированию внутри него события `message`. Чтобы можно было отреагировать на отправку данных со страницы, нужно создать обработчик события `message`:

```
// внутренний код рабочего веб-потока
self.onmessage = function(event){
    var data = event.data;

    // какие-то действия с данными
};
```

Помните, что объект `self` в этом примере на самом деле ссылается на рабочий веб-поток внутри его глобальной области (это объект, отличный от экземпляра `Worker` на странице). По завершении обработки данных можно отправить их странице обратно, вызвав метод `postMessage()`. Например, в следующем примере в функцию передается массив чисел, который нужно отсортировать, а затем отсортированный массив возвращается странице:

Листинг WebWorkerExample01.js


```
// внутренний код рабочего веб-потока
self.onmessage = function(event){
    var data = event.data;

    // по умолчанию метод sort() сравнивает строки
    data.sort(function(a, b){
        return a - b;
    });
    self.postMessage(data);
};
```

Страница и рабочий веб-поток взаимодействуют друг с другом с помощью сообщений. Вызов метода `postMessage()` в рабочем веб-потоке приводит к асинхронному генерированию события `message` для экземпляра `Worker` на странице. Если бы на странице нужно было использовать этот веб-поток, это можно было бы сделать так:

Листинг WebWorkerExample01.htm

```
// код страницы
var data = [23,4,7,9,2,14,6,651,87,41,7798,24],
    worker = new Worker("WebWorkerExample01.js");

worker.onmessage = function(event){
    var data = event.data;

    // какие-то действия с итоговым массивом
};

// отправка массива рабочему веб-потоку для сортировки
worker.postMessage(data);
```

Сортировка — очень хороший пример длительной операции, которую имеет смысл делегировать рабочему веб-потоку, чтобы не блокировать пользовательский интерфейс. К другим примерам таких операций можно отнести обработку изображений, например преобразование цветного изображения в черно-белое, и криптографические вычисления.

Вы можете остановить рабочий веб-поток, вызвав метод `close()`. Он похож на метод `terminate()`, который можно вызвать со страницы, тем, что дальнейшие события после этого не генерируются:

```
// внутренний код рабочего веб-потока
self.close();
```

Подключение других сценариев

Может показаться, что без динамического создания элемента `<script>` нельзя добавлять новые сценарии в рабочий веб-поток, но, к счастью, для этого в глобальной области видимости веб-потока предусмотрен метод `importScripts()`. Он принимает

один или несколько URL-адресов, с которых нужно загрузить JavaScript-код. Код загружается синхронно, так что при вызове `importScripts()` программа приостанавливается, пока все сценарии не будут загружены и выполнены, например:

```
// внутренний код рабочего веб-потока
importScripts("file1.js", "file2.js");
```

Хотя файл `file2.js` может загрузиться до `file1.js`, они все равно будут запущены в том порядке, в котором указаны. Сценарии выполняются в глобальной области видимости рабочего веб-потока, поэтому если они содержат JavaScript-код, специфичный для страницы, они могут не запуститься в веб-потоке. Как правило, код рабочего веб-потока очень специализирован и не имеет ничего общего с кодом страницы.

Будущее рабочих веб-потоков

Работа над спецификацией Web Workers далека от завершения. Веб-потоки, описанные в этом разделе, называются *специализированными рабочими веб-потоками* (dedicated workers), то есть они выделяются конкретной странице и не могут быть общими. Однако в спецификации также представлена концепция *общих рабочих веб-потоков* (shared workers), которые могут быть общими для одной страницы, открытой в нескольких вкладках браузера. Хотя Safari 5, Chrome и Opera 10.6 поддерживают общие рабочие веб-потоки, спецификация еще может претерпеть изменения.

Также продолжаются споры о том, что должно быть доступно внутри рабочих веб-потоков. Есть мнение, что им должны быть доступны все хранилища данных, которые доступны странице, то есть не только объект XHR, но и объекты `localStorage` и `sessionStorage`, индексированная база данных (Indexed DB), веб-сокеты, серверные события и т. д. Судя по всему, большинству разработчиков нравится эта идея, поэтому можно ожидать, что возможности для манипуляций в глобальной области видимости рабочих веб-потоков будут расширены.

Резюме

Некоторые JavaScript API не входят в спецификацию HTML5, но часто упоминаются и используются в связи с ней. Многие из этих API все еще разрабатываются, но уже реализованы в браузерах, а потому заслуживают внимания.

- С помощью метода `requestAnimationFrame()` вы можете оптимизировать анимации на основе JavaScript, указав, когда анимация активна. Это позволяет браузеру эффективнее планировать перерисовку экрана.
- Page Visibility позволяет узнать, когда пользователь просматривает страницу и когда она скрыта.

- Geolocation позволяет определить местоположение пользователя (нужно его разрешение). Эта возможность очень популярна в веб-приложениях для мобильных устройств.
- File позволяет читать данные из файлов для их отображения, обработки или отправки. Объединив эти средства с функционалом перетаскивания из HTML5, можно легко отправлять файлы серверу.
- Веб-таймеры предоставляют ценные сведения о скорости загрузки и визуализации страниц.
- Рабочие веб-потоки позволяют асинхронно выполнять JavaScript-код без блокирования пользовательского интерфейса. Это очень полезно при обработке больших объемов данных и сложных вычислениях, которые в противном случае мешали бы пользователю работать со страницей, занимая много времени.

Приложение A. ECMAScript Harmony

С возобновлением интереса к веб-разработке в 2004 году производители браузеров и другие заинтересованные стороны начали обсуждать, каковы предпочтительные направления развития JavaScript. При работе над четвертой редакцией ECMA-262 за основу были взяты два предложения: JavaScript 2.0 от NetScape и JScript.NET от Microsoft. Вместо того чтобы конкурировать на рынке браузеров, стороны решили объединиться под эгидой ECMA с целью разработки нового языка на основе JavaScript. Предполагалось, что таким языком станет ECMAScript 4, но когда позднее была представлена спецификация ECMAScript 3.1, она поставила будущее JavaScript под вопрос. После многих споров и обсуждений было принято решение одобрить ECMAScript 3.1 в качестве очередного этапа эволюции JavaScript и согласовать некоторые аспекты ECMAScript 4 и ECMAScript 3.1 в проекте с кодовым названием Harmony.

В итоге спецификация ECMAScript 3.1 была переименована в ECMAScript 5 и довольно скоро стандартизирована. Детали ECMAScript 5 мы рассмотрели в этой книге. Сразу после утверждения окончательной версии ECMAScript 5 началась работа над Harmony. Цель этого проекта — модернизировать ECMAScript 5 без радикальных изменений, чтобы сохранить дух языка. Хотя в 2011 году многие аспекты Harmony, или ECMAScript 6, все еще разрабатываются, некоторые части спецификации завершены. В этом приложении описаны те части Harmony, которые наверняка войдут в окончательную спецификацию, хотя итоговые реализации языка в браузерах могут от них отличаться.

Общие изменения

В проект Harmony вошли несколько базовых изменений языка ECMAScript, которые устраняют ряд пробелов в функциональности языка.

Константы

Одним из серьезных недостатков JavaScript является отсутствие формальных констант. Чтобы исправить ситуацию, разработчики Harmony добавили в язык ключевое слово `const`. Используемое подобно `var`, оно позволяет определить переменную, значение которой невозможно изменить после инициализации:

```
const MAX_SIZE = 25;
```

Константу можно определить везде, где может быть определена переменная. Имена констант не могут совпадать с именами переменных или функций, объявленных в той же области видимости, так что следующий код приведет к ошибке:

```
const FLAG = true;
var FLAG = false;    // ошибка!
```

Если не считать, что значения констант неизменны, их можно использовать так же, как переменные. Попытки изменить значение константы просто игнорируются, например:

```
const FLAG = true;
FLAG = false;
alert(FLAG);    // true
```

Константы поддерживаются в Firefox, Safari 3+, Opera 9+ и Chrome. В Safari и Opera ключевое слово `const` работает так же, как `var`, то есть значения переменных, объявленных с его помощью, можно изменять.

Блочная и другие области видимости

Как не раз упоминалось, в JavaScript нет блочных областей видимости. Это означает, что переменные, определенные внутри блоков, работают так же, как если бы они были определены в функции-контейнере. В Harmony для создания блочных областей видимости введено ключевое слово `let`.

Подобно ключевым словам `const` и `var`, его можно использовать для определения и инициализации любой переменной. Отличие такой переменной в том, что она становится недоступна, как только завершается блок, в котором она определена. Например, довольно часто используется следующая конструкция:

```
for (var i=0; i < 10; i++) {
    // какие-то действия
}

alert(i);    // 10
```

Переменная `i` в этом коде объявлена как локальная в текущей функции, а это означает, что после завершения цикла `for` она по-прежнему доступна. Если же вместо

var использовать ключевое слово let, сделать что-либо с переменной i после цикла не получится:

```
for (let i=0; i < 10; i++) {
    // какие-то действия
}

alert(i);    // Ошибка! Переменная i не определена
```

Если попытаться выполнить этот код, в последней строке возникнет ошибка, потому что определение i недействительно вне цикла for.

Есть и другие способы применения ключевого слова let. Например, с его помощью вы можете явно определить переменные, используемые только в последующем блоке кода:

```
var num = 5;

let (num=10, multiplier=2){
    alert(num * multiplier);    // 20
}

alert(num);        // 5
```

В этом коде инструкция let определяет область, в которой переменная num равна 10, а переменная multiplier — 2. Эта инструкция переопределяет значение num, объявленное ранее с помощью ключевого слова var, так что внутри блока let результат умножения num на multiplier равен 20, тогда как вне блока переменная num остается равной 5. Поскольку для каждой инструкции let создается отдельная область видимости, значения переменных внутри нее не влияют на значения снаружи.

Используя похожий синтаксис, можно задать значения переменных только для одного выражения let, например:

```
var result = let(num=10, multiplier=2) num * multiplier;
alert(result);    // 20
```

Здесь выражение let используется для умножения переменных num и multiplier, которые становятся недоступны после сохранения результата в переменной result.

Блочные области видимости позволяют лучше контролировать доступность переменных в отдельных фрагментах JavaScript-кода.

Функции

Почти весь код содержится в функциях, поэтому разработчики Harmony приложили немало усилий, чтобы сделать функции эффективнее и упростить их использование.

Прочие и распределенные аргументы

В Harmony больше нет объекта `arguments`, и вы никак не можете получить доступ к необъявленным аргументам. Тем не менее с помощью *прочих аргументов* (rest arguments) вы можете указать, что количество аргументов может быть переменным. Прочие аргументы обозначаются тремя точками, за которыми следует идентификатор. Это позволяет определить аргументы, которые наверняка будут переданы, и собрать прочие аргументы в массив, например:

```
function sum(num1, num2, ...nums){
    var result = num1 + num2;
    for (let i=0, len=nums.length; i < len; i++){
        result += nums[i];
    }
    return result;
}

var result = sum(1, 2, 3, 4, 5, 6);
```

Эта функция `sum()` принимает как минимум два аргумента, но может принимать и дополнительные аргументы в массиве `nums`. В отличие от объекта `arguments`, прочие аргументы передаются в экземпляре `Array`, так что вам доступны все методы массивов. Объект с прочими аргументами всегда является экземпляром типа `Array`, даже если они в функцию не передаются.

С концепцией прочих аргументов тесно связаны *распределенные аргументы* (spread arguments). С их помощью вы можете передать в функцию массив и спроецировать каждый его элемент на конкретный аргумент в функции. Синтаксис у распределенных аргументов такой же, что и у прочих: три точки перед значением. Единственное их отличие в том, что распределенные аргументы используются при вызове функции, а прочие — при ее определении. Например, вместо того чтобы передавать в метод `sum()` отдельные числа, используйте синтаксис распределенных аргументов:

```
var result = sum(...[1, 2, 3, 4, 5, 6]);
```

Здесь массив аргументов передается в функцию `sum()` в качестве распределенных аргументов. Этот код функционально эквивалентен следующему:

```
var result = sum.apply(this, [1, 2, 3, 4, 5, 6]);
```

Предлагаемые по умолчанию значения аргументов

Все аргументы ECMAScript-функций считаются не обязательными, потому что количество переданных аргументов не проверяется, но вместо того чтобы вручную проверять, какие из них на самом деле переданы, вы можете указать их значения, предлагаемые по умолчанию. Если аргументы не будут переданы в функцию, используются эти значения.

Чтобы задать для аргумента значение, предлагаемое по умолчанию, просто укажите его после определения аргумента и знака равенства, например:

```
function sum(num1, num2=0){
    return num1 + num2;
}

var result1 = sum(5);
var result2 = sum(5, 5);
```

Функция `sum()` принимает два аргумента, но второй из них не обязателен и имеет по умолчанию значение 0. Необязательные аргументы удобны тем, что при их использовании можно не проверять, переданы ли в функцию соответствующие значения.

Генераторы

Генератор (generator) — это объект, который генерирует последовательность значений, по одному за раз. В Harmony вы можете создать генератор, определив функцию, которая возвращает специфическое значение с помощью оператора `yield`. При вызове такой функции создается и возвращается экземпляр типа `Generator`, после чего можно вызвать метод `next()` для получения первого значения генератора. При этом вызывается оригинальная функция, выполнение которой прекращается по достижении оператора `yield`, возвращающего указанное значение. Таким образом, оператор `yield` работает подобно `return`. Если снова вызвать метод `next()`, выполнение кода продолжится с инструкции, следующей за `yield`, до следующего оператора `yield`, возвращающего новое значение. Вот пример:

```
function myNumbers(){
    for (var i=0; i < 10; i++){
        yield i * 2;
    }
}

var generator = myNumbers();

try {
    while(true){
        document.write(generator.next() + "<br />");
    }
} catch(ex){
    // пустой блок
} finally {
    generator.close();
}
```

При вызове функции `myNumbers()` возвращается генератор. Сама функция `myNumbers()` очень проста и содержит только цикл `for`, который генерирует значение на каждой итерации. Каждый вызов `next()` инициирует очередную итерацию цикла `for`, которая

возвращает следующее значение. Первым возвращается значение 0, вторым — 2, третьим — 4, и т. д. Когда метод `myNumbers()` выполняется без вызова `yield` (после заключительной итерации цикла), вызов `next()` генерирует ошибку `StopIteration`. Для предотвращения ошибки цикл `while`, выводящий все числа генератора, заключен в блок `try-catch`.

Если генератор больше не требуется, рекомендуется вызвать метод `close()`. Это гарантирует, что будет выполнен остальной код оригинальной функции, в том числе любые блоки `finally`, связанные с инструкциями `try-catch`.

Генераторы полезны, если нужно получить последовательность значений, в которой каждое последующее значение как-то связано с предыдущим.

Массивы и другие структуры

Другая область повышенного внимания в Harmony — массивы. Это одна из самых часто используемых структур данных в JavaScript, поэтому разработка интуитивных и более эффективных способов работы с массивами была и остается важной задачей.

Итераторы

Итератор (iterator) — это объект, который перебирает последовательность значений и возвращает их по одному за раз. Для этого обычно используют циклы `for` и `for-in`, но итераторы позволяют делать то же самое без циклов. В Harmony итераторы поддерживаются объектами всех типов.

Чтобы создать итератор, вызовите конструктор `Iterator`, передав ему объект, значения которого нужно перебрать. Получить следующее значение в последовательности можно методом `next()`. По умолчанию он возвращает массив, первым элементом которого является индекс значения (в случае массивов) или имя свойства (в случае объектов), а вторым — значение. Если доступных значений больше нет, метод `next()` генерирует ошибку `StopIteration`. Вот пример:

```
var person = {
    name: "Nicholas",
    age: 29
};
var iterator = new Iterator(person);

try {
    while(true){
        let value = iterator.next();
        document.write(value.join(":") + "<br>");
    }
} catch(ex){
    // пустой блок
}
```

Этот код создает итератор для объекта `person`. При первом вызове метод `next()` возвращает массив `["name", "Nicholas"]`, а при втором — `["age", 29]`. Весь фрагмент выводит следующее:

```
name:Nicholas
age:29
```

Если итератор создается для объекта, отличного от массива, свойства возвращаются в том же порядке, что и в цикле `for-in`. Помимо прочего, это означает, что возвращаются только свойства экземпляра и порядок возврата свойств зависит от реализации. Итераторы, созданные для массивов, работают подобным образом, перебирая все их элементы:

```
var colors = ["red", "green", "blue"];
var iterator = new Iterator(colors);

try {
    while(true){
        let value = iterator.next();
        document.write(value.join(":") + "<br>");
    }
} catch(ex){

}
```

Этот код выводит следующее:

```
0:red
1:green
2:blue
```

Если передать в конструктор `Iterator` второй аргумент `true`, метод `next()` вернет только имя свойства или индекс:

```
var iterator = new Iterator(colors, true);
```

В этом случае каждый вызов `next()` будет возвращать только индекс значения, а не массив, содержащий индекс и значение.

Вы можете создавать итераторы для собственных пользовательских типов, определив специальный метод __iterator__(), который должен возвращать объект, содержащий метод next(). Этот метод вызывается при передаче экземпляра пользовательского типа в конструктор Iterator.

Абстракции массивов

Абстракция массива (array comprehension) — это способ инициализации массива конкретными значениями, соответствующими определенным критериям. Этот

элемент, представленный в Harmony, является популярной конструкцией в Python. Синтаксис абстракции массива в JavaScript таков:

```
массив = [ значение for each (переменная in значения) условие ];
```

Здесь *значение* — это фактическое значение, включаемое в итоговый массив и основанное на элементах массива *значений*. Конструкция `for each` перебирает все элементы массива *значений* и сохраняет каждое значение в *переменной*. Если необязательное *условие* выполняется, значение добавляется в итоговый массив, например:

```
// исходный массив
var numbers = [0,1,2,3,4,5,6,7,8,9,10];

// копирование всех элементов в новый массив
var duplicate = [i for each (i in numbers)];

// получение только четных чисел
var evens = [i for each (i in numbers) if (i % 2 == 0)];

// умножение каждого значения на 2
var doubled = [i*2 for each (i in numbers)];

// умножение каждого нечетного числа на 3
var tripledOdds = [i*3 for each (i in numbers) if (i % 2 > 0)];
```

Во всех абстракциях массивов в этом коде для перебора значений массива `numbers` используется переменная `i`, при этом некоторые из абстракций содержат условия, которые фильтруют результаты массива. Если условие эквивалентно `true`, значение добавляется в массив. Этот синтаксис не слишком отличается от традиционного JavaScript-синтаксиса, но более лаконичен, чем использование цикла `for` для решения аналогичной задачи. Абстракции массивов реализованы только в Firefox (версии 2 и более поздних) и для их применения требуется, чтобы атрибут `type` элемента `<script>` имел значение `"application/javascript;version=1.7"`.

Значение в абстракции массива может быть генератором или итератором.

Присваивание с деструктуризацией

Довольно часто требуется извлечь одно или несколько значений из группы и присвоить их переменным. Возьмем для примера массив, возвращаемый методом `next()` итератора и содержащий имя и значение свойства. Чтобы сохранить каждый элемент в отдельной переменной, требуется две инструкции, например:

```
var nextValue = ["color", "red"];
var name = nextValue[0];
var value = nextValue[1];
```

Присваивание с деструктуризацией (destructuring assignment) позволяет назначить два элемента массива переменным с помощью одной инструкции:

```
var [name, value] = ["color", "red"];
alert(name);     // "color"
alert(value);    // "red"
```

В традиционном JavaScript-синтаксисе литерал массива не может находиться с левой стороны знака равенства, однако теперь это возможно. Присваивание с деструктуризацией указывает, что переменным в массиве слева от знака равенства необходимо присвоить значения из массива, указанного справа от знака равенства. В результате переменная `name` получает значение `"color"`, а переменная `value` — значение `"red"`.

Если вам требуются не все значения, можете предоставить переменные только для тех из них, которые нужны, например:

```
var [, value] = ["color", "red"];
alert(value);    // "red"
```

Здесь значение (`"red"`) присваивается только переменной `value`.

Присваивание с деструктуризацией предоставляет некоторые интересные возможности. Например, с его помощью можно поменять местами значения двух переменных. В ECMAScript 3 это обычно выполняют следующим образом:

```
var value1 = 5;
var value2 = 10;

var temp = value1;
value1 = value2;
value2 = temp;
```

Теперь это можно сделать без временной переменной:

```
var value1 = 5;
var value2 = 10;

[value2, value1] = [value1, value2];
```

Объекты также поддерживают присваивание с деструктуризацией, например:

```
var person = {
    name: "Nicholas",
    age: 29
};

var { name: personName, age: personAge } = person;

alert(personName);    // "Nicholas"
alert(personAge);     // 29
```

Если литерал объекта используется слева от знака равенства, это считается присваиванием с деструктуризацией. На самом деле указанная инструкция определяет две переменные, `personName` и `personAge`, которые заполняются соответствующими данными из объекта `person`. Как и при работе с массивами, вы можете отобрать нужные значения:

```
var { age: personAge } = person;
alert(personAge);    // 29
```

Этот код извлекает из объекта `person` только свойство `age`.

Новые типы объектов

Harmony содержит несколько новых типов объектов, реализующих возможности, которые ранее были доступны только интерпретатору JavaScript.

Прокси-объекты

Harmony вводит в JavaScript концепцию прокси. *Прокси* (proxy) — это объект, который предоставляет интерфейс для работы с другими объектами. Например, присвоение значения свойству прокси может на самом деле приводить к вызову функции другого объекта. Прокси обеспечивают полезную абстракцию для доступа к подмножеству той или иной информации через API при сохранении полного контроля над источником данных.

Прокси можно создать с помощью метода `Proxy.create()`, передав ему объект `handler` и необязательный объект `prototype`:

```
var proxy = Proxy.create(handler);

// создание прокси с прототипом myObject
var proxy = Proxy.create(handler, myObject);
```

Объект `handler` содержит свойства, которые определяют *ловушки* (traps). Ловушки — это функции, которые обрабатывают (перехватывают) встроенные операции, чтобы они могли выполняться иначе. Далее описано семь *фундаментальных ловушек* (fundamental traps), которые должны быть реализованы во всех прокси-объектах, чтобы они могли работать предсказуемо и без ошибок.

- `getOwnPropertyDescriptor` — функция, вызываемая, когда для прокси вызывается метод `Object.getOwnPropertyDescriptor()`. Она принимает в качестве аргумента имя свойства и должна возвращать дескриптор свойства или значение `null`, если свойство не существует.
- `getPropertyDescriptor` — функция, вызываемая, когда для прокси вызывается метод `Object.getPropertyDescriptor()` (это новый метод в Harmony). Она принимает

в качестве аргумента имя свойства и должна возвращать дескриптор свойства или значение `null`, если свойство не существует.

- `getOwnPropertyNames` — функция, вызываемая, когда для прокси вызывается метод `Object.getOwnPropertyNames()`. Она принимает в качестве аргумента имя свойства и должна возвращать массив строк.
- `getPropertyNames` — функция, вызываемая, когда для прокси вызывается метод `Object.getPropertyNames()` (это новый метод в Harmony). Она принимает в качестве аргумента имя свойства и должна возвращать массив строк.
- `defineProperty` — функция, вызываемая, когда для прокси вызывается метод `Object.defineProperty()`. Она принимает в качестве аргументов имя свойства и дескриптор свойства.
- `delete` — функция, которая вызывается, когда к свойству объекта применяется оператор `delete`. Функция принимает в качестве аргумента имя свойства и возвращает значение `true`, если свойство было успешно удалено, и `false` в противном случае.
- `fix` — функция, которая вызывается при вызове метода `Object.freeze()`, `Object.seal()` или `Object.preventExtensions()`. Возвращает значение `undefined` для генерирования ошибки, когда для прокси вызывается один из этих методов.

В дополнение к этим фундаментальным ловушкам есть также шесть *производных ловушек* (derived traps). В отличие от фундаментальных ловушек, отсутствие одной или нескольких производных ловушек не приводит к ошибкам. Каждая производная ловушка переопределяет JavaScript-операцию, выполняемую по умолчанию.

- `has` — функция, которая вызывается, когда с объектом используется оператор `in`, например `"name" in object`. Функция принимает в качестве аргумента имя свойства и возвращает `true`, если свойство содержится в объекте, и `false` в противном случае.
- `hasOwn` — функция, которая выполняется, когда для прокси вызывается метод `hasOwnProperty()`. Функция принимает в качестве аргумента имя свойства и возвращает `true`, если свойство содержится в объекте, и `false` в противном случае.
- `get` — функция, которая вызывается при чтении свойства. Функция принимает два аргумента: ссылку на объект, свойство которого читается, и имя свойства. Ссылкой на объект может быть сам прокси или объект, производный от прокси.
- `set` — функция, которая вызывается при записи свойства. Функция принимает три аргумента: ссылку на объект, свойство которого записывается, имя свойства и значение свойства. Как и в предыдущем случае, ссылкой на объект может быть сам прокси или объект, производный от прокси.
- `enumerate` — функция, которая вызывается, когда прокси помещается в цикл `for-in`. Функция должна возвращать массив строк, содержащий имена свойств для использования в цикле `for-in`.

- `keys` — функция, которая выполняется, когда для прокси вызывается метод `Object.keys()`. Как и функция `enumerate`, эта функция должна возвращать массив строк.

Обычно прокси используются для доступа к данным через API, если непосредственный доступ к данным недопустим. Предположим, например, что вам нужно реализовать традиционный стек. Конечно, с массивом можно работать как со стеком, но вы хотите использовать только методы `push()` и `pop()` и свойство `length`. В этом случае можно создать прокси, работающий с массивом, но предоставляющий доступ только к трем его членам:

```
/*
 * Еще один эксперимент с прокси из ES6. Этот код создает стек
 * на основе массива. Прокси отфильтровывает из интерфейса стека
 * все, кроме операций "push", "pop" и "length", что делает его подлинным
 * стеком, содержимым которого невозможно манипулировать.
 */

var Stack = (function(){

    var stack = [],
        allowed = [ "push", "pop", "length" ];

    return Proxy.create({
        get: function(receiver, name){;
            if (allowed.indexOf(name) > -1){
                if(typeof stack[name] == "function"){
                    return stack[name].bind(stack);
                } else {
                    return stack[name];
                }
            } else {
                return undefined;
            }
        }

    });

});

var mystack = new Stack();

mystack.push("hi");
mystack.push("goodbye");

console.log(mystack.length);    // 1

console.log(mystack[0]);        // undefined
console.log(mystack.pop());     // "goodbye"
```

Этот код создает конструктор `Stack`. Вместо того чтобы работать с объектом `this`, конструктор `Stack` для работы с массивом возвращает объект-прокси. Из ловушек

мы определяем только ловушку `get`, которая просто проверяет массив допустимых свойств перед возвращением значения. Все запрещенные свойства возвращают значение `undefined`, тогда как методы `push()`, `pop()` и свойство `length` функционируют надлежащим образом. Ключевым фрагментом этого кода является объявление ловушки `get`, которая фильтрует получение членов объекта. Если членом является функция, ловушка возвращает связанную версию функции, которая работает с нижележащим массивом, а не с самим прокси.

Функции-прокси

Кроме объектов-прокси, в Harmony можно также создавать *прокси-функции* (proxy functions). Прокси-функция — это то же самое, что и объект-прокси, только она может выполняться. Прокси-функцию можно создать методом `Proxy.createFunction()`, передав ему объект `handler`, функцию-ловушку вызова и необязательную функцию-ловушку конструктора, например:

```
var proxy = Proxy.createFunction(handler, function(){}, function(){});
```

У объекта `handler` доступны те же ловушки, что и у объектов-прокси. Функция-ловушка вызова — это код, выполняемый при вызове функции-прокси, например `proxy()`. Ловушка конструктора — это код, выполняемый, когда прокси-функция вызывается с помощью оператора `new`, например `new proxy()`. Если ловушка конструктора не определена, вместо нее используется ловушка вызова.

Типы Map и Set

Тип `Map` — ассоциативный массив, или *простая проекция* (simple map), предназначен для хранения пар ключей и значений. Разработчики обычно используют для этого те или иные обобщенные объекты, но при этом ключи можно легко спутать со встроенными свойствами. В ассоциативных массивах ключи и значения хранятся отдельно от свойств объекта, что отчасти защищает данные. Вот пример:

```
var map = new Map();

map.set("name", "Nicholas");
map.set("book", "Professional JavaScript");

console.log(map.has("name"));    // true
console.log(map.get("name"));    // "Nicholas"

map.delete("name");
```

Базовый API ассоциативных массивов состоит из методов `get()`, `set()`, `has()` и `delete()`, назначение которых очевидно, а ключами могут быть примитивные или ссылочные значения.

Тип `Set` (множество) — это коллекция элементов без дубликатов. В отличие от ассоциативного массива, множество содержит только ключи без значений. Базовый API множества содержит метод `add()` для добавления элементов, метод `has()` для проверки существования элементов и метод `delete()` для удаления элементов. Вот примеры их использования:

```
var set = new Set();
set.add("name");

console.log(set.has("name"));    // true
set.delete("name");

console.log(set.has("name"));    // false
```

На октябрь 2011 года спецификация типов `Map` и `Set` не завершена, так что описанные детали могут измениться, прежде чем эти типы будут реализованы в интерпретаторах JavaScript.

Тип WeakMap

Тип `WeakMap` интересен тем, что это первый ECMAScript-тип, позволяющий узнать, что на объект нет ссылок. Он похож на ассоциативный массив, но отличается от него тем, что ключ должен быть объектом, а когда объект перестает существовать, пара из этого объекта и значения удаляется из `WeakMap`, например:

```
var key = {},
    map = new WeakMap();

map.set(key, "Hi!");

// разрыв связи ключа со значением; удаление значения
key = null;
```

Сценарии использования типа `WeakMap` пока не ясны, но в Java он доступен как `WeakHashMap`, так что и в JavaScript применение ему найдется.

Тип StructType

Одним из общепризнанных недостатков JavaScript является представление всех чисел единственным типом данных. В WebGL для смягчения этой проблемы были введены типизированные массивы, а в ECMAScript 6 для интеграции дополнительных числовых типов добавлены типизированные структуры. *Структурный тип* (struct type) аналогичен структурам в языке C, которые позволяют объединять несколько свойств в одной записи. Теперь в JavaScript можно создавать похожие структуры данных, указывая свойства и типы данных, которые они содержат. В первоначальной реализации определены следующие блочные типы:

- `uint8` — 8-разрядное целое число без знака;
- `int8` — 8-разрядное целое число со знаком;
- `uint16` — 16-разрядное целое число без знака;
- `int16` — 16-разрядное целое число со знаком;
- `uint32` — 32-разрядное целое число без знака;
- `int32` — 32-разрядное целое число со знаком;
- `float32` — 32-разрядное число с плавающей точкой;
- `float64` — 64-разрядное число с плавающей точкой.

Каждый из этих блочных типов содержит одно значение, и ожидается, что в будущем к ним будут добавлены дополнительные типы.

Чтобы создать структурный тип, нужно вызвать конструктор `StructType`, передав ему литерал объекта с определениями свойств, например:

```
var Size = new StructType({ width: uint32, height: uint32 });
```

Этот код создает новый структурный тип с именем `Size` и двумя свойствами: `width` и `height`, каждое из которых содержит 32-разрядное целое число без знака. Переменная `Size` на самом деле является конструктором, который можно использовать так же, как и конструктор объекта. Чтобы инициализировать экземпляр структурного типа, нужно передать в его конструктор литерал объекта со значениями свойств:

```
var boxSize = new Size({ width: 80, height: 60 });
console.log(boxSize.width);        // 80
console.log(boxSize.height);       // 60
```

Этот код создает экземпляр типа `Size` со значениями `width` и `height`, равными 80 и 60 соответственно. Эти свойства можно записывать и читать, но они всегда должны содержать 32-разрядные целые числа без знака.

Структуры могут быть свойствами более сложных структурных типов, например:

```
var Location = new StructType({ x: int32, y: int32 });
var Box = new StructType({ size: Size, location: Location });

var boxInfo = new Box({ size: { width:80, height:60 },
                        location: { x: 0, y: 0 }});
console.log(boxInfo.size.width);    // 80
```

Здесь мы создаем простой структурный тип с именем `Location` и сложный структурный тип `Box`, свойства которого сами являются экземплярами структурных типов. Конструктор `Box` принимает литерал объекта, определяющий значения всех свойств, и проверяет при этом их типы.

Тип ArrayType

Со структурными типами тесно связаны типы-массивы. Они позволяют создавать массивы, значениями которых могут быть только данные конкретных типов, что очень похоже на типизированные массивы в WebGL. Чтобы создать новый тип-массив, вызовите конструктор `ArrayType`, передав ему тип данных, которые он должен содержать, и количество элементов в массиве, например:

```
var SizeArray = new ArrayType(Size, 2);
var boxes = new SizeArray([ { width: 80, height: 60 },
                            { width: 50, height: 50 } ]);
```

Этот код создает тип-массив с именем `SizeArray`, содержащий два экземпляра типа `Size`. Чтобы инициализировать тип-массив, нужно передать ему массив с преобразуемыми данными, при этом поддерживаются литералы, которые приводятся к правильному типу (как и в случае структурных типов).

Классы

Разработчики давно мечтали о простом способе определения в JavaScript *классов*, таких как Java-классы, и теперь это, наконец, возможно. Классы в ECMAScript 6 представляют собой синтаксические ухищрения, маскирующие текущий подход, основанный на конструкторах и прототипах. Рассмотрим следующее определение типа:

```
function Person(name, age){
    this.name = name;
    this.age = age;
}

Person.prototype.sayName = function(){
    alert(this.name);
};

Person.prototype.getOlder = function(years){
    this.age += years;
};
```

Вот эквивалент этого определения типа, определенный с использованием нового синтаксиса классов:

```
class Person {

    constructor(name, age){
        public name = name;
        public age = age;
    }
```

```
    sayName(){
        alert(this.name);
    }

    getOlder(years){
        this.age += years;
    }
}
```

Определение класса начинается с ключевого слова `class`, за которым следует имя типа. Свойства и методы класса определяются в фигурных скобках. Чтобы определить метод, достаточно указать его имя и скобки, а ключевое слово `function` не требуется. Если метод называется `constructor`, он работает как конструктор класса (аналогично функции `Person` в предыдущем фрагменте). Все остальные методы и свойства, определенные в фигурных скобках класса, применяются к прототипу, так что в данном случае методы `sayName()` и `getOlder()` добавляются к свойству `Person.prototype`.

В конструкторе переменные, которым предшествует ключевое слово `public` или `private`, создаются как свойства экземпляра. В приведенном примере свойства `name` и `age` определены как открытые.

Закрытые члены

И экземпляры и прототипы классов по умолчанию поддерживают закрытые члены. Ключевое слово `private` указывает, что член является закрытым и недоступен извне методов класса. Для доступа к закрытому члену нужно сначала вызвать функцию `private()` с аргументом `this`. Например, следующий код добавляет в класс `Person` закрытое свойство `age`:

```
class Person {

    constructor(name, age){
        public name = name;
        private age = age;
    }

    sayName(){
        alert(this.name);
    }

    getOlder(years){
        private(this).age += years;
    }

}
```

Синтаксис доступа к закрытым свойствам все еще обсуждается и может быть изменен в будущем.

Методы чтения и записи свойств

Классы позволяют определять методы чтения и записи непосредственно для свойств, без вызова метода `Object.defineProperty()`. Синтаксис при этом такой же, что и в случае обычных методов, только содержит дополнительное ключевое слово `get` или `set`, например:

```
class Person {

    constructor(name, age){
        public name = name;
        public age = age;
        private innerTitle = "";

        get title(){
            return innerTitle;
        }

        set title(value){
            innerTitle = value;
        }
    }

    sayName(){
        alert(this.name);
    }

    getOlder(years){
        this.age += years;
    }

}
```

В этой версии класса `Person` определены методы чтения и записи свойства `title`. Каждый из них работает с переменной `innerTitle`, которая определена в конструкторе. Методы чтения и записи свойств прототипа тоже можно использовать, только они должны быть определены вне конструктора.

Наследование

Главное преимущество классов над более традиционным JavaScript-синтаксисом — простота реализации наследования. Вместо кражи конструктора и цепочек прототипов можно использовать простой синтаксис, общий для многих языков: ключевое слово `extends`. Вот пример:

```
class Employee extends Person {
    constructor(name, age){
        super(name,age);
    }
}
```

Здесь мы наследуем от класса `Person` класс `Employee`. Цепочка прототипов при этом используется неявно, а кража конструктора теперь формально поддерживается в виде функции `super()`. Предыдущий фрагмент является логическим эквивалентом следующего:

```
function Employee(name, age){
    Person.call(this, name, age);
}

Employee.prototype = new Person();
```

Кроме того, при определении класса можно указать объект, который нужно сделать прототипом. Для этого следует использовать ключевое слово `prototype` вместо `extends`:

```
var basePerson = {
    sayName: function(){
        alert(this.name);
    },

    getOlder: function(years){
        this.age += years;
    }
};

class Employee prototype basePerson {
    constructor(name, age){
        public name = name;
        public age = age;
    }
}
```

В этом примере мы делаем объект `basePerson` прототипом класса `Employee`, что позволяет реализовать такое же наследование, которое в настоящее время осуществляется с помощью метода `Object.create()`.

Модули

Модули (modules), называемые также *пространствами имен* (namespaces), или *пакетами* (packages), — популярное средство организации JavaScript-приложений. Каждый модуль поддерживает специфичную уникальную функциональность, является самодостаточным и отделен от других модулей. Хотя в JavaScript со временем возникло несколько форматов модулей, разработчики пытаются формализовать создание модулей и управление ими.

Модули работают в собственном верхнеуровневом контексте выполнения и потому не могут засорить глобальный контекст выполнения, в который импортируются. По умолчанию все переменные, функции, классы и другие сущности, объявленные

в модуле, закрыты в нем. Чтобы член был доступен извне, нужно указать перед ним ключевое слово `export`, например:

```
module MyModule {
    // экспорт членов
    export let myobject = {};
    export function hello(){ alert("hi"); };

    // сокрытие членов
    function goodbye(){
        // ...
    }
}
```

Этот модуль экспортирует объект `myobject` и функцию `hello()`. Его можно использовать в любых других местах — на странице или в другом модуле, — но импортировать из него можно только эти члены. Для импорта служит команда `import`:

```
// импорт только объекта myobject
import myobject from MyModule;
console.log(myobject);

// импорт всех сущностей
import * from MyModule;
console.log(myobject);
console.log(hello);

// явное указание импортируемых сущностей
import {myobject, hello} from MyModule;
console.log(myobject);
console.log(hello);

// импорта нет - модуль используется напрямую
console.log(MyModule.myobject);
console.log(MyModule.hello);
```

Если модуль доступен в контексте выполнения, экспортируемые им члены можно получить напрямую. Импортируемые члены просто вносятся в текущий контекст выполнения, чтобы с ними можно было работать, не ссылаясь на сам модуль.

Внешние модули

Модули можно подключать динамически по URL-адресу. Для этого просто укажите URL-адрес после объявления модуля:

```
module MyModule from "mymodule.js";
import myobject from MyModule;
```

Этот код предписывает интерпретатору JavaScript загрузить файл `mymodule.js` и извлечь из него модуль `MyModule`. Имейте в виду, что это блокирующий вызов:

интерпретатор JavaScript не продолжит обработку кода, пока указанный файл не будет загружен из сети.

Чтобы включить в код какие-то экспортируемые модулем сущности, не добавляя сам модуль, используется директива `import`:

```
import myobject from "mymodule.js";
```

В общем, модули группируют связанный функционал и защищают глобальную область видимости от засорения.

Приложение Б. Строгий режим

Строгий режим (strict mode) был представлен в ECMAScript 5. Он позволяет реализовать более строгую проверку ошибок глобально или локально (в одной функции). Преимущество строгого режима в том, что он позволяет раньше получать уведомления об ошибках, и вы можете оперативно устранять их.

Важно понимать правила строгого режима, так как в следующей версии ECMAScript он будет использоваться по умолчанию. Строгий режим поддерживается в Internet Explorer 10+, Firefox 4+, Safari 5.1+ и Chrome.

Включение строгого режима

Включить строгий режим можно с помощью следующей *директивы* (pragma) — строки, которая не назначается никакой переменной:

```
"use strict";
```

Этот синтаксис допустим даже в ECMAScript 3. Если интерпретатор JavaScript не поддерживает строгий режим, эта директива просто игнорируется как строковый литерал, не назначенный никакой переменной.

Если эта директива применяется глобально, то есть вне функции, строгий режим включается для всего сценария. Это означает, что добавление директивы в сценарий, который объединяется с другими сценариями в одном файле, включает строгий режим для всего кода в файле.

Строгий режим можно также включить только внутри отдельной функции, например:

```
function doSomething(){
    "use strict";

    // другие операции
}
```

Если у вас нет полного контроля над всеми сценариями на странице, рекомендуется включать строгий режим только внутри конкретных функций, которые были протестированы в нем.

Переменные

Способ и время создания переменных в строгом и обычном режимах различаются. Так, в строгом режиме нельзя создать глобальную переменную случайно. Например, следующий код в нестрогом режиме создает глобальную переменную:

```
// Переменная не объявлена
// Нестрогий режим: создается глобальная переменная
// Строгий режим: генерируется ошибка ReferenceError

message = "Hello world!";
```

Хотя переменной `message` не предшествует ключевое слово `var` и она не определена как свойство глобального объекта, она все же автоматически создается как глобальная. В строгом режиме присваивание значения необъявленной переменной приводит при запуске кода к ошибке `ReferenceError`.

Кроме того, в строгом режиме нельзя вызвать для переменной оператор `delete`. В нестрогом режиме это возможно, при этом интерпретатор просто возвращает `false`. В строгом режиме попытка удалить переменную приводит к ошибке:

```
// Удаление переменной
// Нестрогий режим: ошибка игнорируется без каких-либо действий
// Строгий режим: генерируется ошибка ReferenceError

var color = "red";
delete color;
```

Строгий режим также налагает ограничения на имена переменных. В нем запрещено использовать переменные с именами `implements`, `interface`, `let`, `package`, `private`, `protected`, `public`, `static` и `yield`. Они теперь являются зарезервированными словами, которые, возможно, будут задействованы в будущих редакциях ECMAScript. Попытка использовать их в качестве имен переменных в строгом режиме приведет к синтаксической ошибке.

Объекты

В строгом режиме операции с объектами чаще возвращают ошибки, потому что в нестрогом режиме многие ошибки просто игнорируются. Благодаря этому строгий режим способствует раннему устранению ошибок.

Вот некоторые ситуации, когда при доступе к свойству объекта возникает ошибка:

- присваивание значения свойству, доступному только для чтения, приводит к ошибке `TypeError`;
- применение оператора `delete` к неконфигурируемому свойству приводит к ошибке `TypeError`;
- попытка добавить свойство к нерасширяемому объекту приводит к ошибке `TypeError`.

Другое ограничение объектов имеет место, когда вы объявляете их с помощью литералов. Если используется литерал объекта, имена свойств должны быть уникальными, например:

```
// Два свойства с одним именем
// Нестрогий режим: ошибки нет, приоритет отдается второму свойству
// Строгий режим: генерируется синтаксическая ошибка

var person = {
                name: "Nicholas",
                name: "Greg"
            };
```

Этот литерал объекта `person` содержит два свойства с именем `name`. В нестрогом режиме к объекту будет добавлено второе свойство, а в строгом возникнет синтаксическая ошибка.

Функции

Прежде всего, строгий режим требует, чтобы именованные аргументы функций были уникальными, например:

```
// Повторяющиеся именованные аргументы
// Нестрогий режим: ошибки нет, действителен только второй аргумент
// Строгий режим: генерируется ошибка SyntaxError

function sum (num, num){
    // какие-то действия
}
```

В нестрогом режиме это объявление функции не приводит к ошибке. Вы можете получить доступ ко второму аргументу `num` по имени, тогда как первый доступен только через объект `arguments`.

В строгом режиме также слегка меняется поведение объекта `arguments`. В нестрогом режиме изменения именованного аргумента отражаются в этом объекте, а в строгом — нет, например:

```
// Изменение значения именованного аргумента
// Нестрогий режим: изменение отражается на объекте arguments
// Строгий режим: изменение не отражается на объекте arguments

function showValue(value){
    value = "Foo";
    alert(value);            // "Foo"
    alert(arguments[0]);     // Нестрогий режим: "Foo"
                             // Строгий режим: "Hi"
}

showValue("Hi");
```

Эта функция `showValue()` принимает единственный именованный аргумент `value`. Мы вызываем ее с аргументом `"Hi"`, который назначается переменной `value`, но внутри функции значение `value` изменяется на `"Foo"`. В нестрогом режиме при этом также изменяется значение `arguments[0]`, но в строгом режиме это разные сущности.

В строгом режиме также недоступны свойства `arguments.callee` и `arguments.caller`. В нестрогом режиме они представляют текущую функцию и вызвавшую ее функцию соответственно, а в строгом попытка доступа к любому из этих свойств приводит к ошибке `TypeError`, например:

```
// Попытка доступа к свойству arguments.callee
// Нестрогий режим: код выполняется обычным образом
// Строгий режим: генерируется ошибка TypeError

function factorial(num){
    if (num <= 1) {
        return 1;
    } else {
        return num * arguments.callee(num-1)
    }
}

var result = factorial(5);
```

При попытке прочитать или записать свойство `caller` или `arguments` функции также генерируется ошибка `TypeError`. В приведенном примере эта ошибка возникла бы при доступе к свойствам `factorial.caller` и `factorial.callee`.

Как и в случае переменных, в строгом режиме нельзя назначать функциям имена `implements`, `interface`, `let`, `package`, `private`, `protected`, `public`, `static` и `yield`.

Наконец, в строгом режиме можно объявлять функции только на верхнем уровне сценария или функции. Это означает, например, что объявление функции в инструкции `if` является синтаксической ошибкой:

```
// Объявление функции в инструкции if
// Нестрогий режим: функция поднимается за пределы инструкции if
// Строгий режим: генерируется синтаксическая ошибка

if (true){
    function doSomething(){
```

```
        // ...
    }
}
```

Этот синтаксис допустим во всех браузерах в нестрогом режиме, но приводит к синтаксической ошибке в строгом.

Функция eval()

Печально известная функция `eval()` в строгом режиме стала работать иначе. Основное изменение состоит в том, что она больше не создает переменные или функции в контексте-контейнере, например:

```
// Использование функции eval() для создания переменной
// Нестрогий режим: в оповещении выводится число 10
// Строгий режим: при вызове alert(x) генерируется ошибка ReferenceError

function doSomething(){
    eval("var x=10");
    alert(x);
}
```

В нестрогом режиме этот код создает локальную переменную `x` в функции `doSomething()`, а затем выводит ее значение в оповещении. В строгом режиме вызов `eval()` не создает переменную `x` внутри функции `doSomething()`, и при вызове `alert()` возникает ошибка `ReferenceError`, потому что переменная `x` не объявлена.

Переменные и функции можно объявлять в функции `eval()`, но они остаются ограничены специальной областью видимости, которая доступна в течение выполнения кода, а по его завершении уничтожается. Например, следующий код работает без ошибок:

```
"use strict";
var result = eval("var x=10, y=11; x+y");
alert(result);    // 21
```

Здесь мы объявляем переменные `x` и `y` внутри функции `eval()`, а затем вычисляем их сумму. Переменная `result` содержит результат сложения, хотя сами переменные `x` и `y` в момент вызова `alert()` больше не существуют.

Идентификаторы eval и arguments

В строгом режиме теперь явно запрещено использовать имена `eval` и `arguments` в качестве идентификаторов и изменять значения этих переменных:

```
// Переопределение eval и arguments
// Нестрогий режим: все в порядке
```

```
// Строгий режим: синтаксическая ошибка

var eval = 10;
var arguments = "Hello world!";
```

В нестрогом режиме вы можете перезаписать значения `eval` и `arguments`, но в строгом это приводит к синтаксической ошибке. Ошибка возникнет во всех перечисленных далее случаях использования этих имен:

- объявление переменной с помощью ключевого слова `var`;
- присваивание переменной другого значения;
- попытка изменить значение переменной, например с помощью оператора `++`;
- использование в качестве имени функции;
- использование в качестве именованного аргумента функции;
- использование в качестве имени исключения в инструкции `try-catch`.

Преобразование значения this

Одной из серьезнейших проблем с безопасностью и одним из самых непонятных аспектов применения JavaScript-кода является преобразование значения `this` в ряде ситуаций. При вызове метода `apply()` или `call()` для функции значение `null` или `undefined` приводится в нестрогом режиме к глобальному объекту. В строгом режиме значение `this` функции всегда используется так, как указано, например:

```
// Доступ к свойству
// Нестрогий режим: доступ к глобальному свойству
// Строгий режим: ошибка, потому что this имеет значение null

var color = "red";

function displayColor(){
    alert(this.color);
}

displayColor.call(null);
```

Этот код передает в метод `displayColor.call()` значение `null`, так что в нестрогом режиме значением `this` функции является глобальный объект, в результате выводится оповещение со строкой `"red"`. В строгом режиме значение `this` функции равно `null`, и при доступе к свойству объекта `null` генерируется ошибка.

Другие изменения

Следует также упомянуть несколько других отличий строгого режима. Так, в нем недоступна инструкция `with`. Она изменяет способ разрешения идентификаторов

и была удалена из строгого режима ради упрощения языка. Попытка использовать `with` в строгом режиме приведет к синтаксической ошибке:

```
// Использование инструкции with
// Нестрогий режим: все в порядке
// Строгий режим: синтаксическая ошибка

with(location){
    alert(href);
}
```

Также в строгом режиме недоступны *восьмеричные литералы*. Они начинаются с нуля и традиционно были причиной многих ошибок. Теперь использование восьмеричного литерала в строгом режиме считается синтаксической ошибкой:

```
// Использование восьмеричного литерала
// Нестрогий режим: value имеет значение 8
// Строгий режим: синтаксическая ошибка

var value = 010;
```

Как уже говорилось, в ECMAScript 5 метод `parseInt()` был изменен; в нестрогом режиме он обрабатывает восьмеричные литералы так же, как десятичные литералы с начальным нулем, например:

```
// Использование восьмеричного литерала в функции parseInt()
// Нестрогий режим: value имеет значение 8
// Строгий режим: value имеет значение 10

var value = parseInt("010");
```

Приложение B. JavaScript-библиотеки

Библиотеки (libraries) в JavaScript помогают компенсировать различия браузеров и упрощают доступ к их сложным функциям. Различают *библиотеки общего назначения* (general libraries) и *специальные библиотеки* (specialty libraries). JavaScript-библиотеки общего назначения предоставляют доступ к наиболее востребованному функционалу браузеров и могут использоваться для реализации базовых возможностей веб-сайтов и веб-приложений. Специальные библиотеки предназначены для решения специфичных задач и используются только в отдельных частях веб-сайтов и веб-приложений. В этом приложении представлен обзор нескольких библиотек и даны ссылки на дополнительные ресурсы.

Библиотеки общего назначения

JavaScript-библиотеки общего назначения предоставляют функционал, охватывающий несколько областей. Все библиотеки общего назначения уравнивают различия в интерфейсе и реализациях браузеров, заключая общий функционал в новые API. Некоторые из этих API напоминают встроенный функционал, тогда как другие выглядят совершенно иначе. Библиотеки общего назначения обычно поддерживают взаимодействие с DOM, технологию Ajax и вспомогательные методы, помогающие решать типичные задачи.

YUI

YUI (Yahoo! User Interface — пользовательский интерфейс Yahoo!) — это JavaScript- и CSS-библиотека с открытым исходным кодом. Она поддерживает большое количество конфигураций с разными наборами файлов, из которых вы можете загрузить только нужные. YUI охватывает все аспекты применения JavaScript, от базовых служебных средств и вспомогательных функций до полнофункциональных виджетов. Над YUI работает отдельная команда программистов из

Yahoo!, которые ведут информативную документацию и оказывают поддержку пользователям библиотеки.

Лицензия: BSD License.

Веб-сайт: www.yuilibrary.com.

Prototype

Prototype — это библиотека с открытым исходным кодом, предоставляющая простые API для решения типичных задач веб-программирования. Первоначально разработанная для Ruby on Rails, она содержит определения полезных классов и обеспечивает возможность наследования в JavaScript. Для этого в Prototype реализованы классы, инкапсулирующие часто используемый и сложный функционал в простые API-вызовы. Библиотека Prototype содержится в одном файле, который можно легко подключить к любой странице. Библиотеку написал и сопровождает Сэм Стивенсон (Sam Stephenson).

Лицензия: MIT License и Creative Commons Attribution-Share Alike 3.0 Unported.

Веб-сайт: www.prototypejs.org/.

Dojo Toolkit

В Dojo Toolkit, библиотеке с открытым исходным кодом, группы функциональных возможностей организованы в пакеты, которые можно загружать по требованию. Dojo поддерживает множество параметров и конфигураций, охватывая почти все задачи, которые приходится решать средствами JavaScript. Библиотеку Dojo Toolkit создал Алекс Расселл (Alex Russell), а сопровождают ее сотрудники и добровольцы из Dojo Foundation.

Лицензия: «новая» BSD License или Academic Free License 2.1.

Веб-сайт: www.dojotoolkit.org/.

MooTools

MooTools — это компактная оптимизированная библиотека с открытым исходным кодом, которая добавляет методы к встроенным JavaScript-объектам для расширения их функционала и предоставляет ряд новых объектов. К ее достоинствам можно отнести небольшой размер и простой API.

Лицензия: MIT License.

Веб-сайт: www.mootools.net/.

jQuery

jQuery — это библиотека с открытым исходным кодом, которая предоставляет функциональный интерфейс программирования для JavaScript. В основе jQuery лежит использование CSS-селекторов для работы с DOM-элементами. Благодаря цепочке вызовов jQuery-код больше похож не на JavaScript-сценарий, а на описание того, что должно произойти. Такой стиль кодирования особо популярен среди дизайнеров и разработчиков прототипов. jQuery написал и сопровождает Джон Резиг (John Resig).

Лицензия: MIT License или General Public License (GPL).

Веб-сайт: http://jquery.com/.

MochiKit

MochiKit — это библиотека с открытым исходным кодом, состоящая из нескольких меньших служебных программ. Она содержит информативную документацию, хорошо протестирована, предоставляет множество API, примеров их применения и тестов для гарантирования качества. Написал и поддерживает MochiKit Боб Ипполито (Bob Ippolito).

Лицензия: MIT License или Academic Free License 2.1.

Веб-сайт: www.mochikit.com/.

Underscore.js

Хотя *Underscore.js* можно назвать библиотекой общего назначения лишь с натяжкой, она предоставляет некоторые возможности для функционального программирования на JavaScript. В документации Underscore.js преподносится как дополнение к jQuery, предоставляющее низкоуровневые средства для работы с объектами, массивами, функциями и другими JavaScript-типами данных. Underscore.js сопровождает Джереми Ашкеназ (Jeremy Ashkenas) из DocumentCloud.

Лицензия: MIT License.

Веб-сайт: http://documentcloud.github.com/underscore/.

Библиотеки для интернет-приложений

Библиотеки для интернет-приложений (Internet application libraries) упрощают разработку веб-приложений в целом. Вместо того чтобы упрощать программирование отдельных небольших фрагментов, они предоставляют целые концептуальные каркасы для приложений. Хотя эти библиотеки могут содержать низкоуровневый функционал, в первую очередь они предназначены для быстрой разработки веб-приложений.

Backbone.js

Backbone.js — это компактная библиотека, реализующая концепцию «модель-представление-контроллер» (Model-View-Controller, MVC) на базе Underscore.js. Она оптимизирована для создания одностраничных приложений и позволяет легко обновлять части страницы при изменении состояния приложения. Backbone.js сопровождает Джереми Ашкеназ (Jeremy Ashkenas) из DocumentCloud.

Лицензия: MIT License.

Веб-сайт: http://documentcloud.github.com/backbone/.

Rico

Rico — это библиотека с открытым исходным кодом, созданная для того, чтобы упростить разработку многофункциональных интернет-приложений. Она содержит средства для работы с Ajax, анимациями, стилями и виджетами. Библиотеку сопровождает небольшая группа добровольцев, но в 2008 году ее развитие заметно замедлилось.

Лицензия: Apache License 2.0.

Веб-сайт: http://openrico.org/.

qooxdoo

Библиотека *qooxdoo* с открытым исходным кодом помогает на всех этапах разработки веб-приложений. Она содержит собственные версии классов и интерфейсов, реализуя модель программирования, похожую на традиционные объектно-ориентированные языки. Библиотека включает полный набор инструментов для разработки GUI и компиляторы для упрощения сборки клиентского кода. Первоначально она разрабатывалась как внутренняя библиотека в компании веб-хостинга 1&1 (www.1and1.com), но позднее была выпущена по лицензии с открытым исходным кодом. Сопровождают и разрабатывают эту библиотеку несколько специалистов из компании 1&1.

Лицензия: GNU Lesser General Public License (LGPL) или Eclipse Public License (EPL).

Веб-сайт: www.qooxdoo.org/.

Библиотеки для анимации и эффектов

Анимация и другие визуальные эффекты стали важными составляющими веб-контента. Реализовать плавные анимации на веб-страницах непросто, но, к счастью, некоторые разработчики уже позаботились об этом, создав специальные библиотеки.

Многие из упомянутых ранее JavaScript-библиотек общего назначения также поддерживают анимации.

script.aculo.us

Библиотека *script.aculo.us* дополняет Prototype, обеспечивая простой доступ к эффектным анимациям средствами CSS и DOM. Чтобы ее можно было использовать, необходимо предварительно загрузить Prototype. Библиотека script.aculo.us — одна из самых популярных библиотек эффектов, используемая на крупных веб-сайтах и в веб-приложениях по всему миру. Создал и сопровождает ее Томас Фукс (Thomas Fuchs).

Лицензия: MIT License.

Веб-сайт: http://script.aculo.us/.

moo.fx

Библиотека *moo.fx* с открытым исходным кодом работает поверх Prototype или MooTools. Она очень компактна (последняя версия занимает всего 3 Кбайт) и предназначена для создания анимаций при минимуме кодирования. По умолчанию moo.fx входит в MooTools, но ее можно также загрузить отдельно для использования с Prototype.

Лицензия: MIT License.

Веб-сайт: http://moofx.mad4milk.net/.

Lightbox

Библиотека *Lightbox* предназначена для создания простых графических наложений на страницах. Для работы ей требуются библиотеки Prototype и script.aculo.us. Идея в том, чтобы пользователи могли просматривать изображения или последовательности изображений, не покидая текущую страницу. Для Lightbox-наложений можно настраивать вид и переходы. Lightbox разрабатывает и сопровождает Локеш Дакар (Lokesh Dhakar).

Лицензия: Creative Commons Attribution 2.5 License.

Веб-сайт: www.huddletogether.com/projects/lightbox2/.

Библиотеки для криптографии

По мере роста популярности Ajax-приложений возросла и потребность в защите передаваемых данных с помощью криптографии. Популярные алгоритмы

безопасности реализованы в нескольких JavaScript-библиотеках. Большинство из них не поддерживаются авторами официально, но все же широко используются.

JavaScript MD5

JavaScript MD5 — это библиотека с открытым исходным кодом, в которой реализованы хэш-функции MD4, MD5 и SHA-1. Пол Джонсон (Paul Johnston) с несколькими соавторами реализовали каждый алгоритм в отдельном файле. На главной странице проекта можно найти обзор хэш-функций и краткое описание уязвимостей и сценариев использования библиотеки.

Лицензия: BSD License.

Веб-сайт: http://pajhome.org.uk/crypt/md5/.

JavaScrypt

В библиотеке *JavaScrypt* реализованы алгоритмы MD5 и AES (256-разрядный). На веб-сайте JavaScrypt доступно много информации об истории криптографии и о том, как она используется в компьютерных средах. Нормальной документации об интеграции библиотеки в веб-приложения нет, а ее исходный код полон сложных математических манипуляций и вычислений.

Лицензия: всеобщее достояние.

Веб-сайт: www.fourmilab.ch/javascrypt/.

Приложение Г. JavaScript-инструменты

Написание JavaScript-кода во многом напоминает программирование на любом другом языке, при этом, как и в других языках, вы можете использовать вспомогательные инструменты. Количество таких инструментов для JavaScript постоянно растет, благодаря чему искать проблемы, оптимизировать и развертывать приложения становится проще. Некоторые из этих инструментов запускаются из JavaScript-кода, другие можно запускать извне браузера. В этом приложении представлен обзор нескольких полезных инструментов и ссылки на ресурсы с дополнительными сведениями.

Средства проверки кода

Проблемы с отладкой JavaScript-сценариев отчасти связаны с тем, что многие среды IDE не выделяют синтаксические ошибки при вводе кода. Большинство разработчиков пишут некий код и запускают его в браузере, чтобы проверить, правилен ли он. Вы можете значительно ускорить устранение ошибок, проверив JavaScript-код перед запуском. *Средства проверки кода* проверяют его синтаксис и выводят предупреждения, если он не соответствует нормам стиля.

JSLint

JSLint — это средство проверки JavaScript-кода от Дугласа Крокфорда (Douglas Crockford). Данная утилита ищет основные синтаксические ошибки, следуя при этом наиболее строгим правилам, чтобы гарантировать, что код будет работать везде. Вы также можете включить режим вывода предупреждения о стиле кодирования, в том числе о форматировании кода, использовании необъявленных глобальных переменных и не только. Утилита JSLint написана на JavaScript, но ее можно запустить из командной строки с помощью интерпретатора Rhino на основе Java, а также с помощью WScript и других JavaScript-интерпретаторов.

На веб-сайте проекта можно найти версию JSLint для каждого интерпретатора командной строки.

Цена: бесплатно.

Веб-сайт: www.jslint.com/.

JSHint

JSHint — это ветвь JSLint с расширенными возможностями настройки применяемых правил. Как и JSLint, эта утилита проверяет код на предмет синтаксических ошибок и ищет в нем нежелательные кодовые последовательности. Подобно JSLint, JSHint можно запускать из командной строки с помощью Rhino.

Цена: бесплатно.

Веб-сайт: www.jshint.com/.

JavaScript Lint

JavaScript Lint — это средство проверки JavaScript-кода с открытым исходным кодом, которое на языке C написал Маттиас Миллер (Matthias Miller). Эта утилита не имеет отношения к JSLint и ищет синтаксические ошибки с помощью SpiderMonkey — JavaScript-интерпретатора, который использовался в Firefox. Она поддерживает довольно много параметров, позволяющих запросить дополнительные предупреждения о стиле кодирования, необъявленных переменных и недоступном коде. Есть версии этой утилиты для Windows и Macintosh.

Цена: бесплатно.

Веб-сайт: www.javascriptlint.com/.

Средства сокращения объема кода

При сборке JavaScript-проекта важно удалить из кода лишние символы, чтобы сократить объем трафика и ускорить обработку кода ради улучшения впечатлений пользователей. Доступно несколько *средств сокращения объема кода*, обеспечивающих разную степень сжатия.

JSMin

JSMin — это средство сокращения объема кода от Дугласа Крокфорда, написанное на языке C. Оно предназначено в основном для удаления пустот и комментариев, но так, чтобы итоговый код мог быть выполнен без каких-либо проблем. Утилита JSMin доступна как исполняемый Windows-файл с исходным кодом на языке C и многих других языках.

Цена: бесплатно.

Веб-сайт: www.crockford.com/javascript/jsmin.html.

Dojo ShrinkSafe

Создатели Dojo Toolkit разработали также средство *ShrinkSafe*, которое с помощью интерпретатора Rhino сначала выполняет синтаксический анализ JavaScript-кода, преобразуя его в поток лексем, а затем сокращает его. Как и JSMin, эта утилита удаляет пробелы (но не разрывы строк) и комментарии, но в дополнение заменяет имена локальных переменных двухсимвольными именами. Итоговый код получается меньше, чем при применении JSMin, при этом внесение синтаксических ошибок также исключено.

Цена: бесплатно.

Веб-сайт: http://shrinksafe.dojotoolkit.org/.

YUI Compressor

Группа YUI разработала средство сжатия, которое называется *YUI Compressor*. Подобно ShrinkSafe, эта утилита выполняет синтаксический анализ JavaScript-кода, преобразуя его в поток лексем с помощью Rhino, а затем удаляет комментарии и пробелы и заменяет имена переменных. В отличие от ShrinkSafe, YUI Compressor также удаляет разрывы строк и вносит несколько других небольших изменений. Как правило, файлы, обработанные с помощью YUI Compressor, имеют меньшие размеры, чем итоговые файлы, полученные с помощью JSMin или ShrinkSafe.

Цена: бесплатно.

Веб-сайт: http://yuilibrary.com/projects/yuicompressor.

Средства модульного тестирования

Разработка через тестирование (Test-Driven Development, TDD) — это процесс разработки ПО, основанный на модульном тестировании. До недавних пор средств модульного тестирования для JavaScript почти не было, но теперь они доступны в большинстве библиотек и отдельно.

JsUnit

JsUnit — это оригинальная библиотека для модульного тестирования JavaScript-кода, не связанная с каким-либо продуктом для JavaScript и основанная на популярной библиотеке JUnit для Java. Она поддерживает автоматизацию

тестирования и отправку результатов серверу, а веб-сайт проекта предлагает примеры и базовую документацию.

Цена: бесплатно.

Веб-сайт: www.jsunit.net/.

YUI Test

YUI Test — это часть библиотеки YUI, с помощью которой можно тестировать не только YUI-код, но и любой другой JavaScript-код. YUI Test включает простые и сложные утверждения, а также средства имитации простых событий мыши и клавиатуры. Эта среда полностью документирована в Yahoo! Developer Network и включает примеры, документацию по API и не только. Тесты выполняются в браузере, а результаты выводятся на странице. YUI Test используется в YUI для тестирования всей библиотеки.

Цена: бесплатно.

Веб-сайт: http://yuilibrary.com/projects/yuitest/.

Dojo Object Harness (DOH)

Средство *Dojo Object Harness* (DOH) первоначально использовалось для модульного тестирования в Dojo, а затем было выпущено для всех. Как и в других средах тестирования, модульные тесты выполняются в браузере.

Цена: бесплатно.

Веб-сайт: www.dojotoolkit.org/.

qUnit

qUnit — это среда модульного тестирования, разработанная для использования с jQuery. В действительности она служит для тестирования в самой библиотеке jQuery. Несмотря на это qUnit не зависит от jQuery и может применяться для тестирования любого JavaScript-кода. Она считается очень простой и позволяет легко приступить к работе.

Цена: бесплатно.

Веб-сайт: https://github.com/jquery/qunit.

Генераторы документации

Большинство сред IDE включают генераторы документации для основного языка. Поскольку у JavaScript нет официальной среды IDE, документацию традиционно

составляли вручную или с помощью перепрофилированных генераторов других языков. К счастью, теперь доступны генераторы документации, специально предназначенные для JavaScript.

JsDoc Toolkit

JsDoc Toolkit — один из первых генераторов документации для JavaScript. Он требует добавления в исходный код комментариев в стиле Javadoc, которые затем обрабатываются и выводятся как HTML-файлы. Для настройки формата HTML-кода можно использовать один из готовых JsDoc-шаблонов или создать собственный. JsDoc Toolkit доступен как Java-пакет.

Цена: бесплатно.

Веб-сайт: http://code.google.com/p/jsdoc-toolkit/.

YUI Doc

YUI Doc — это генератор документации из YUI. Он написан на языке Python, так что для работы с ним нужно установить на компьютере исполняющую среду Python. YUI Doc выдает в качестве результата HTML-файлы с интегрированными средствами поиска свойств и методов, основанными на виджете Autocomplete из YUI. Как и JsDoc, YUI Doc требует, чтобы разработчики добавляли в исходный код комментарии в стиле Javadoc. HTML-код по умолчанию можно настроить, изменив файла шаблона и связанную таблицу стилей.

Цена : бесплатно.

Веб-сайт: www.yuilibrary.com/projects/yuidoc/.

AjaxDoc

Назначение *AjaxDoc* немного отличается от предыдущих генераторов. Вместо создания HTML-файлов документации для JavaScript это средство создает XML-файлы в формате, используемом .NET-языками, такими как C# и Visual Basic .NET. Такой подход позволяет стандартным системам подготовки документации в .NET создавать документацию в виде HTML-файлов. В AjaxDoc комментарии в документации должны оформляться так же, как в .NET-языках. Этот генератор был создан для Ajax-решений ASP.NET, но может использоваться и в отдельных проектах.

Цена: бесплатно.

Веб-сайт: www.codeplex.com/ajaxdoc/.

Среды безопасного выполнения кода

По мере роста популярности гибридных веб-приложений разработчики стали использовать на своих страницах гораздо больше JavaScript-кода сторонних компаний. Это привело к некоторым проблемам с безопасностью, связанным с доступом к ограниченному функционалу. Инструменты, описанные в этом разделе, предназначены для создания сред, в которых JavaScript-сценарии из нескольких источников могут работать, не влияя друг на друга.

ADsafe

ADsafe — это среда от Дугласа Крокфорда, определяющая подмножество JavaScript-средств, доступ к которым можно безопасно предоставлять сторонним сценариям. Чтобы код можно было запустить в ADsafe, страница должна включать JavaScript-библиотеку ADsafe и быть размечена в формате ADsafe-виджетов. Такой код можно безопасно выполнять на любой странице.

Цена: бесплатно.

Веб-сайт: www.adsafe.org/.

Caja

В *Caja* реализован уникальный подход к безопасному выполнению JavaScript-кода. Подобно ADsafe, Caja определяет подмножество JavaScript-средств, которое можно использовать без каких-либо опасений. Caja позволяет санировать выполняемый JavaScript-код и убедиться, что он делает только то, что должен. В рамках проекта доступен язык *Cajita*, который определяет еще меньшее подмножество JavaScript-функционала. Работа над средой Caja началась сравнительно недавно, но она уже привлекла внимание как перспективное средство безопасного выполнения многих сценариев на одной странице.

Цена: бесплатно.

Веб-сайт: http://code.google.com/p/google-caja/.

Н. Закас

JavaScript для профессиональных веб-разработчиков

Перевел с английского А. Лютич

Заведующий редакцией	*П. Щеголев*
Ведущий редактор	*Ю. Сергиенко*
Литературный редактор	*А. Жданов*
Корректор	*С. Беляева*
Верстка	*Л. Соловьева*

ООО «Питер Пресс», 192102, Санкт-Петербург, ул. Андреевская (д. Волкова), д. 3, литер А, пом. 7Н.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12.000 — Книги печатные профессиональные, технические и научные.

Подписано в печать 22.01.15. Формат 70×100/16. Усл. п. л. 77,400. Тираж 1500. Заказ 135.

Отпечатано в полном соответствии с предоставленными материалами в типографии ООО «Чеховский печатник».
142300, Московская область, г. Чехов, ул. Полиграфистов, д.1.
тел.: +7 915 222 15 42, +7 926 063 81 80.

ИЗДАТЕЛЬСКИЙ ДОМ
ПИТЕР®
WWW.PITER.COM

**ПРЕДСТАВИТЕЛЬСТВА ИЗДАТЕЛЬСКОГО ДОМА «ПИТЕР»
предлагают профессиональную и популярную литературу по различным направлениям: история и публицистика, экономика и финансы, менеджмент и маркетинг, компьютерные технологии, медицина и психология.**

РОССИЯ

Санкт-Петербург: м. «Выборгская», Б. Сампсониевский пр., д. 29а
тел./факс: (812) 703-73-73, 703-73-72; e-mail: sales@piter.com

Москва: м. «Электрозаводская», Семеновская наб., д. 2/1, стр. 1
тел./факс: (495) 234-38-15; e-mail: sales@msk.piter.com

Воронеж: тел.: 8 951 861-72-70; e-mail: voronej@piter.com

Екатеринбург: ул. Бебеля, д. 11а
тел./факс: (343) 378-98-41, 378-98-42; e-mail: office@ekat.piter.com

Нижний Новгород: тел.: 8 960 187-85-50; e-mail: nnovgorod@piter.com

Новосибирск: Комбинатский пер., д. 3
тел./факс: (383) 279-73-92; e-mail: sib@nsk.piter.com

Ростов-на-Дону: ул. Ульяновская, д. 26
тел./факс: (863) 269-91-22, 269-91-30; e-mail: piter-ug@rostov.piter.com

Самара: ул. Молодогвардейская, д. 33а, офис 223
тел./факс: (846) 277-89-79, 229-68-09; e-mail: samara@piter.com

УКРАИНА

Киев: Московский пр., д. 6, корп. 1, офис 33
тел./факс: (044) 490-35-69, 490-35-68; e-mail: office@kiev.piter.com

Харьков: ул. Суздальские ряды, д. 12, офис 10
тел./факс: (057) 7584145, +38 067 545-55-64; e-mail: piter@kharkov.piter.com

БЕЛАРУСЬ

Минск: ул. Розы Люксембург, д. 163
тел./факс: (517) 208-80-01, 208-81-25; e-mail: minsk@piter.com

Издательский дом «Питер» приглашает к сотрудничеству зарубежных торговых партнеров или посредников, имеющих выход на зарубежный рынок
тел./факс: (812) 703-73-73; e-mail: spb@piter.com

Издательский дом «Питер» приглашает к сотрудничеству авторов
тел./факс издательства: (812) 703-73-72, (495) 974-34-50

Заказ книг для вузов и библиотек
тел./факс: (812) 703-73-73, доб. 6250; e-mail: uchebnik@piter.com

Заказ книг по почте: на сайте www.piter.com; по тел.: (812) 703-73-74, доб. 6225