Создание активных веб-страниц

5-е издание
Включает Ajax и DOM

JavaScript

Подробное руководство

Символ®

O'REILLY®

Дэвид Флэнаган

JavaScript

The Definitive Guide

Fifth Edition

David Flanagan

O'REILLY®

JavaScript

Подробное руководство

Пятое издание

Дэвид Флэнаган

Санкт-Петербург — Москва
2008

Дэвид Флэнаган

JavaScript. Подробное руководство, 5-е издание

Перевод А. Киселева

Главный редактор	*А. Галунов*
Зав. редакцией	*Н. Макарова*
Научный редактор	*О. Цилюрик*
Редактор	*А. Жданов*
Корректор	*С. Минин*
Верстка	*Д. Орлова*

Флэнаган Д.

JavaScript. Подробное руководство. – Пер. с англ. – СПб: Символ-Плюс, 2008. – 992 с., ил.

ISBN-10: 5-93286-103-7
ISBN-13: 978-5-93286-103-5

Пятое издание бестселлера «JavaScript. Подробное руководство» полностью обновлено. Рассматриваются взаимодействие с протоколом HTTP и применение технологии Ajax, обработка XML-документов, создание графики на стороне клиента с помощью тега <canvas>, пространства имен в JavaScript, необходимые для разработки сложных программ, классы, замыкания, Flash и встраивание сценариев JavaScript в Java-приложения.

Часть I знакомит с основами JavaScript. В части II описывается среда разработки сценариев, предоставляемая веб-броузерами. Многочисленные примеры демонстрируют, как генерировать оглавление HTML-документа, отображать анимированные изображения DHTML, автоматизировать проверку правильности заполнения форм, создавать всплывающие подсказки с использованием Ajax, как применять XPath и XSLT для обработки XML-документов, загруженных с помощью Ajax. Часть III – обширный справочник по базовому JavaScript (классы, объекты, конструкторы, методы, функции, свойства и константы, определенные в JavaScript 1.5 и ECMAScript v3). Часть IV – справочник по клиентскому JavaScript (API веб-броузеров, стандарт DOM API Level 2 и недавно появившиеся стандарты: объект XMLHttpRequest и тег <canvas>).

ISBN-10: 5-93286-103-7
ISBN-13: 978-5-93286-103-5
ISBN 0-596-10199-6 (англ)

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

Подписано в печать 14.02.2008. Формат 70×100 1/16 . Печать офсетная.
Объем 62 печ. л. Тираж 2000 экз. Заказ N
Отпечатано с готовых диапозитивов в ГУП «Типография «Наука»
199034, Санкт-Петербург, 9 линия, 12.

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

Оглавление

Предисловие

После выхода из печати четвертого издания книги «JavaScript. Подробное руководство» объектная модель документов (Document Object Model, DOM), представляющая собой основу прикладного программного интерфейса (Application Programming Interface, API) для сценариев на языке JavaScript™, исполняющихся на стороне клиента, была реализована достаточно полно, если не полностью, в веб-броузерах. Это означает, что разработчики веб-приложений получили в свое распоряжение универсальный прикладной программный интерфейс для работы с содержимым веб-страниц на стороне клиента и зрелый язык (JavaScript 1.5), остававшийся стабильным на протяжении последующих лет.

Сейчас интерес к JavaScript опять начинает расти. Теперь разработчики используют JavaScript для создания сценариев, работающих по протоколу HTTP, управляющих XML-данными и даже динамически создающих графические изображения в веб-броузере. Многие программисты с помощью JavaScript создают большие программы и применяют достаточно сложные технологии программирования, такие как замыкания и пространства имен. Пятое издание полностью пересмотрено с позиций вновь появившихся технологий Ajax и Web 2.0.

Что нового в пятом издании

В первой части книги, «Основы JavaScript», была расширена глава 8, описывающая функции; в нее включен материал, охватывающий замыкания и вложенные функции. Информация о порядке создания собственных классов была дополнена и выделена в отдельную главу 9. Глава 10 – это еще одна новая глава, которая содержит сведения о пространствах имен, являющихся основой для разработки модульного программного кода многократного использования. Наконец, глава 12 демонстрирует, как применять JavaScript при разработке сценариев на языке Java. Здесь показано, как встраивать интерпретатор JavaScript в приложения на Java 6, как использовать JavaScript для создания Java-объектов и как вызывать методы этих объектов.

Во второй части книги, «Клиентский язык JavaScript», описываются прежняя (уровня 0) объектная модель документа и стандарт DOM консорциума W3C. Поскольку в настоящее время модель DOM имеет универсальные реализации, отпала необходимость в двух отдельных главах, где в предыдущем издании описывались приемы работы с документами. Вторая часть книги подверглась самым существенным изменениям; в нее включен следующий новый материал:

- Глава 19 «Cookies и механизм сохранения данных на стороне клиента» дополнена новой информацией о cookies и сведениями о методиках программирования, применяемых на стороне клиента.

- Глава 20 «Работа с протоколом HTTP» описывает, как выполнять HTTP-запросы с помощью такого мощного инструмента, как объект `XMLHttpRequest`, который делает возможным создание Ajax-подобных веб-приложений.
- Глава 21 «JavaScript и XML» демонстрирует, как средствами JavaScript организовать создание, загрузку, синтаксический разбор, преобразование, выборку, сериализацию и извлечение данных из XML-документов. Кроме того, рассматривается расширение языка JavaScript, получившее название E4X.
- Глава 22 «Работа с графикой на стороне клиента» описывает графические возможности языка JavaScript. Здесь рассматриваются как простейшие способы создания анимированных изображений, так и достаточно сложные приемы работы с графикой с использованием ультрасовременного тега `<canvas>`. Кроме того, здесь говорится о создании графики на стороне клиента средствами подключаемых SVG-, VML-, Flash- и Java-модулей.
- Глава 23 «Сценарии с Java-апплетами и Flash-роликами» рассказывает о подключаемых Flash- и Java-модулях. В этой главе объясняется, как создавать Flash-ролики и Java-апплеты.

Часть III книги представляет собой справочник по прикладному интерфейсу базового языка JavaScript. Изменения в этой части по сравнению с предыдущим изданием весьма незначительные, что обусловлено стабильностью API. Если вы читали 4-е издание, вы найдете эту часть книги удивительно знакомой.

Существенные изменения коснулись организации справочного материала, описывающего прикладной интерфейс объектной модели документа (DOM API), который ранее был выделен в самостоятельную часть отдельно от описания клиентского языка JavaScript. Теперь же оставлена единственная часть со справочной информацией, относящейся к клиентскому языку JavaScript. Благодаря этому отпала необходимость читать описание объекта `Document` в одной части, а затем искать описание объекта `HTMLDocument` в другой. Справочный материал об интерфейсах модели DOM, которые так и не были достаточно полно реализованы в броузерах, попросту убран. Так, интерфейс `NodeIterator` не поддерживается в броузерах, поэтому его описание из этой книги исключено. Кроме того, акцент смещен от сложных формальных определений DOM-интерфейсов к JavaScript-объектам, которые являются фактической реализацией этих интерфейсов. Например, метод `getComputedStyle()` теперь описывается не как метод интерфейса `AbstractView`, а как метод объекта `Window`, что логичнее. Для JavaScript-программистов, создающих клиентские сценарии, нет серьезных оснований вникать в особенности интерфейса `AbstractView`, поэтому его описание было убрано из справочника. Все эти изменения сделали справочную часть книги, посвященную клиентскому языку JavaScript, более простой и удобной.

Порядок работы с книгой

Глава 1 представляет собой введение в язык JavaScript. Остальная часть книги делится на четыре части. Первая часть, которая непосредственно следует за главой 1, описывает основы языка JavaScript. Главы со 2 по 6 содержат достаточно скучный материал, тем не менее прочитать его совершенно необходимо, т. к. он охватывает самые основы, без знания которых невозможно начать изучение нового языка программирования:

- Глава 2 «Лексическая структура» описывает основные языковые конструкции.
- Глава 3 «Типы данных и значения» рассказывает о типах данных, поддерживаемых языком JavaScript.
- Глава 4 «Переменные» охватывает темы переменных, областей видимости переменных и всего, что с этим связано.
- Глава 5 «Выражения и операторы» описывает выражения языка JavaScript и документирует каждый оператор, поддерживаемый этим языком программирования. Поскольку синтаксис JavaScript основан на синтаксисе языка Java, который, в свою очередь, очень многое заимствовал из языков C и C++, программисты, имеющие опыт работы с этими языками, могут лишь вкратце ознакомиться с содержимым этой главы.
- Глава 6 «Инструкции» описывает синтаксис и порядок использования каждой JavaScript-инструкции. Программисты, имеющие опыт работы с языками C, C++ и Java, могут пропустить не все, но некоторые разделы этой главы.

Последующие шесть глав первой части содержат куда более интересные сведения. Они также описывают основы языка JavaScript, но охватывают те его части, которые едва ли вам знакомы, даже если вам приходилось писать на языке C или Java. Если вам требуется настоящее понимание JavaScript, к изучению материала этих глав следует подходить с особой тщательностью.

- Глава 7 «Объекты и массивы» описывает объекты и массивы языка JavaScript.
- Глава 8 «Функции» рассказывает о том, как определяются функции, как они вызываются, каковы их отличительные особенности в языке JavaScript.
- Глава 9 «Классы, конструкторы и прототипы» касается вопросов объектно-ориентированного программирования на языке JavaScript. Рассказывается о том, как определяются функции-конструкторы для новых классов объектов и как работает механизм наследования на основе прототипов. Кроме того, продемонстрирована возможность эмулирования традиционных идиом объектно-ориентированного программирования на языке JavaScript.
- Глава 10 «Модули и пространства имен» показывает, как определяются пространства имен в JavaScript-объектах, и описывает некоторые практические приемы, позволяющие избежать конфликтов имен в модулях.
- Глава 11 «Шаблоны и регулярные выражения» рассказывает о том, как использовать регулярные выражения в языке JavaScript для выполнения операций поиска и замены по шаблону.
- Глава 12 «Разработка сценариев для Java-приложений» демонстрирует возможность встраивания интерпретатора JavaScript в Java-приложения и рассказывает, как JavaScript-программы, работающие внутри Java-приложений, могут обращаться к Java-объектам. Эта глава представляет интерес только для тех, кто программирует на языке Java.

Часть II книги описывает реализацию JavaScript в веб-броузерах. Первые шесть глав рассказывают об основных характеристиках клиентского JavaScript:

- Глава 13 «JavaScript в веб-броузерах» рассказывает об интеграции JavaScript в веб-броузеры. Здесь броузеры рассматриваются как среда программирования и описываются различные варианты встраивания программного JavaScript-кода в веб-страницы для исполнения его на стороне клиента.

- Глава 14 «Работа с окнами броузера» описывает центральный элемент клиентского языка JavaScript – объект `Window` и рассказывает, как использовать этот объект для управления окнами броузера.
- Глава 15 «Работа с документами» описывает объект `Document` и рассказывает, как из JavaScript управлять содержимым, отображаемым в окне броузера. Эта глава является наиболее важной во второй части.
- Глава 16 «CSS и DHTML» рассказывает о порядке взаимодействия между JavaScript-кодом и таблицами CSS-стилей. Здесь показано, как средствами JavaScript изменять стили, вид и положение элементов HTML-документа, создавая визуальные эффекты, известные как DHTML.
- Глава 17 «События и обработка событий» описывает события и порядок их обработки, что является немаловажным для программ, ориентированных на взаимодействие с пользователем.
- Глава 18 «Формы и элементы форм» посвящена тому, как работать с HTML-формами и отдельными элементами форм. Данная глава является логическим продолжением главы 15, но обсуждаемая тема настолько важна, что была выделена в самостоятельную главу.

Вслед за этими шестью главами следуют пять глав, содержащих более узкоспециализированный материал:

- Глава 19 «Cookies и механизм сохранения данных на стороне клиента» охватывает вопросы хранения данных на стороне клиента для последующего использования. В этой главе показано, как средствами HTTP манипулировать cookies и как сохранять их с помощью соответствующих инструментов Internet Explorer и подключаемого Flash-модуля.
- Глава 20 «Работа с протоколом HTTP» демонстрирует, как управлять протоколом HTTP из JavaScript-сценариев, как с помощью объекта `XMLHttpRequest` отправлять запросы веб-серверам и получать от них ответы. Данная возможность является краеугольным камнем архитектуры веб-приложений, известной под названием Ajax.
- Глава 21 «JavaScript и XML» описывает, как средствами JavaScript создавать, загружать, анализировать, преобразовывать и сериализовать XML-документы, а также как извлекать из них данные.
- Глава 22 «Работа с графикой на стороне клиента» рассказывает о средствах JavaScript, ориентированных на работу с графикой. Здесь рассматриваются как простейшие способы создания анимированных изображений, так и достаточно сложные приемы работы с графикой с использованием форматов SVG (Scalable Vector Graphics – масштабируемая векторная графика) и VML (Vector Markup Language – векторный язык разметки), тега `<canvas>` и подключаемых Flash- и Java-модулей.
- Глава 23 «Сценарии с Java-апплетами и Flash-роликами» показывает, как организовать взаимодействие JavaScript-кода с Java-апплетами и Flash-роликами. Кроме того, в ней рассказывается, как обращаться к JavaScript-коду из Java-апплетов и Flash-роликов.

Третья и четвертая части содержат справочный материал соответственно по базовому и клиентскому языкам JavaScript. Здесь приводятся описания объектов, методов и свойств в алфавитном порядке.

Типографские соглашения

В этой книге приняты следующие соглашения:

Курсив

Обозначает первое появление термина. Курсив также применяется для выделения адресов электронной почты, веб-сайтов, FTP-сайтов, имен файлов и каталогов, групп новостей.

Моноширинный шрифт

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

Моноширинный жирный

Используется для выделения текста командной строки, который должен быть введен пользователем.

Моноширинный курсив

Обозначает аргументы функций и элементы, которые в программе необходимо заменить реальными значениями.

Клавиши и элементы пользовательского интерфейса, такие как кнопка Назад или меню Сервис, выделены шрифтом OfficinaSansC.

Использование программного кода примеров

Данная книга призвана оказать помощь в решении ваших задач. Вы можете свободно использовать примеры программного кода из этой книги в своих приложениях и в документации. Вам не нужно обращаться в издательство за разрешением, если вы не собираетесь воспроизводить существенные части программного кода. Например, если вы разрабатываете программу и задействуете в ней несколько отрывков программного кода из книги, вам не нужно обращаться за разрешением. Однако в случае продажи или распространения компакт-дисков с примерами из этой книги вам необходимо получить разрешение от издательства O'Reilly. При цитировании данной книги или примеров из нее и при ответе на вопросы получение разрешения не требуется. При включении существенных объемов программного кода примеров из этой книги в вашу документацию вам *необходимо* получить разрешение издательства.

Мы приветствуем, но не требуем добавлять ссылку на первоисточник при цитировании. Под ссылкой на первоисточник мы подразумеваем указание авторов, издательства и ISBN. Например: «JavaScript: The Definitive Guide, by David Flanagan. Copyright 2006 O'Reilly Media, Inc., 978-0-596-10199-2».

За получением разрешения на использование значительных объемов программного кода примеров из этой книги обращайтесь по адресу *permissions@oreilly.com*.

Отзывы и предложения

С вопросами и предложениями, касающимися этой книги, обращайтесь в издательство:

O'Reilly Media
1005 Gravenstein Highway North
Sebastopol, CA 95472
(800) 998-9938 (в Соединенных Штатах Америки или в Канаде)
(707) 829-0515 (международный)
(707) 829-0104 (факс)

Список опечаток, файлы с примерами и другую дополнительную информацию вы найдете на сайте книги по адресу:

http://www.oreilly.com/catalog/jscript5

Кроме того, все примеры, описываемые в книге, можно загрузить с сайта авторов:

http://www.davidflanagan.com/javascript5

Свои пожелания и вопросы технического характера отправляйте по адресу:

bookquestions@oreilly.com

Дополнительную информацию о книгах, обсуждения, центр ресурсов издательства O'Reilly вы найдете на сайте:

http://www.oreilly.com

Safari® Enabled

Если на обложке технической книги есть пиктограмма «Safari® Enabled», это означает, что книга доступна в Сети через O'Reilly Network Safari Bookshelf.

Safari предлагает намного лучшее решение, чем электронные книги. Это виртуальная библиотека, позволяющая без труда находить тысячи лучших технических книг, вырезать и вставлять примеры кода, загружать главы и находить быстрые ответы, когда требуется наиболее верная и свежая информация. Она свободно доступна по адресу *http://safari.oreilly.com*.

Благодарности

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

Это руководство получило благословение первоклассных технических рецензентов, комментарии которых весьма способствовали улучшению этой книги. Аристотель Пагальцис (Aristotle Pagatzis) (*http://plasmasturm.org*) рецензировал новый материал о функциях, а также главы, в которых в этом издании приводится описание классов и пространств имен. Он с особым тщанием просмотрел программный код примеров и дал весьма ценные комментарии. Дуглас Крокфорд (Douglas Crockford) (*http://www.crockford.com*) рецензировал новый материал о функциях и классах. Норрис Бойд (Norris Boyd), создатель интерпретатора Rhino для JavaScript, рецензировал главу, в которой описывается механизм

встраивания JavaScript-кода в Java-приложения. Питер-Пауль Кох (Peter-Paul Koch) (*http://www.quirksmode.org*), Кристиан Хейльманн (Christian Heilmann) (*http://www.wait-till-i.com*) и Кен Купер (Ken Cooper) рецензировали главы этой книги, связанные с технологией Ajax. Тодд Дихендорф (Todd Ditchendorf) (*http://www.ditchnet.org*) и Джефф Штернс (Geoff Stearns) (*http://blog.deconcept.com*) рецензировали главу, описывающую приемы работы с графикой на стороне клиента. Тодд был настолько любезен, что занялся поиском и исправлением ошибок в программном коде примеров, а Джефф помог разобраться в технологиях Flash и ActionScript. Наконец, Сандерс Клейнфельд (Sanders Kleinfeld) рецензировал всю книгу, уделяя особое внимание деталям. Его предложения и исправления сделали эту книгу более точной и понятной. Приношу слова искренней признательности каждому, кто занимался рецензированием книги. Любые ошибки, которые вам встретятся в книге, лежат на моей совести.

Я очень признателен рецензентам четвертого издания книги. Валдемар Хорват (Waldermar Horwat) из Netscape рецензировал новый материал по JavaScript 1.5. Материал по W3C DOM проверен Филиппом Ле Хегаре (Philippe Le Hegaret) из W3C, Питером-Паулем Кохом (Peter-Paul Koch), Диланом Шиманом (Dylan Schiemann) и Джеффом Ятсом (Jeff Yates). Джозеф Кесселман (Joseph Kesselman) из IBM Research ничего не рецензировал, но очень помог мне, отвечая на вопросы по W3C DOM.

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

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

Эта книга стала лучше еще и благодаря многочисленным редакторам, которые над ней работали. Деб Камерон (Deb Cameron) – редактор этого издания, он особое внимание уделял обновлению и удалению устаревшего материала. Паула Фергюсон (Paula Ferguson) – редактор третьего и четвертого изданий. Френк Уиллисон (Frank Willison) редактировал второе издание, а Эндрю Шульман – первое.

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

Дэвид Флэнаган
http://www.davidflanagan.com
апрель 2006

1

Введение в JavaScript

JavaScript – это интерпретируемый язык программирования с объектно-ориентированными возможностями. С точки зрения синтаксиса базовый язык JavaScript напоминает C, C++ и Java такими программными конструкциями, как инструкция `if`, цикл `while` и оператор `&&`. Однако это подобие ограничивается синтаксической схожестью. JavaScript – это нетипизированный язык, т. е. в нем не требуется определять типы переменных. Объекты в JavaScript отображают имена свойств на произвольные значения. Этим они больше напоминают ассоциативные массивы Perl, чем структуры C или объекты C++ или Java. Механизм объектно-ориентированного наследования JavaScript скорее похож на механизм прототипов в таких малоизвестных языках, как Self, и сильно отличается от механизма наследования в C++ и Java. Как и Perl, JavaScript – это интерпретируемый язык, и некоторые его инструменты, например регулярные выражения и средства работы с массивами, реализованы по образу и подобию языка Perl.

Ядро языка JavaScript поддерживает работу с такими простыми типами данных, как числа, строки и булевы значения. Помимо этого он обладает встроенной поддержкой массивов, дат и объектов регулярных выражений.

Обычно JavaScript применяется в веб-броузерах, а расширение его возможностей за счет введения объектов позволяет организовать взаимодействие с пользователем, управлять веб-броузером и изменять содержимое документа, отображаемое в пределах окна веб-броузера. Эта встроенная версия JavaScript запускает сценарии, внедренные в HTML-код веб-страниц. Как правило, эта версия называется *клиентским* языком JavaScript, чтобы подчеркнуть, что сценарий исполняется на клиентском компьютере, а не на веб-сервере.

В основе языка JavaScript и поддерживаемых им типов данных лежат международные стандарты, благодаря чему обеспечивается прекрасная совместимость между реализациями. Некоторые части клиентского JavaScript формально стандартизированы, другие части стали стандартом де-факто, но есть части, которые являются специфическими расширениями конкретной версии броузера. Совместимость реализаций JavaScript в разных броузерах зачастую приносит немало беспокойств программистам, использующим клиентский язык JavaScript.

В этой главе приводится краткий обзор JavaScript и дается вводная информация, перед тем как перейти к фактическому изучению возможностей языка. Кроме того, в данной главе на нескольких фрагментах кода на клиентском языке JavaScript демонстрируется практическое веб-программирование.

1.1. Что такое JavaScript

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

1.1.1. JavaScript – это не Java

Одно из наиболее распространенных заблуждений о JavaScript состоит в том, что этот язык представляет собой упрощенную версию Java, языка программирования, разработанного в компании Sun Microsystems. Кроме некоторой синтаксической схожести и способности предоставлять исполняемое содержимое для веб-броузеров, эти два языка между собой ничто не связывает. Схожесть имен – не более чем уловка маркетологов (первоначальное название языка – LiveScript – было изменено на JavaScript в последнюю минуту). Однако JavaScript и Java могут взаимодействовать друг с другом (подробнее об этом см. в главах 12 и 23).

1.1.2. JavaScript не простой язык

Поскольку JavaScript является интерпретируемым языком, очень часто он позиционируется как язык сценариев, а не как язык программирования, при этом подразумевается, что языки сценариев проще и в большей степени ориентированы не на программистов, а на обычных пользователей. В самом деле, при отсутствии контроля типов JavaScript прощает многие ошибки, которые допускают неопытные программисты. Благодаря этому многие веб-дизайнеры могут использовать JavaScript для решения ограниченного круга задач, выполняемых по точным рецептам.

Однако за внешней простотой JavaScript скрывается полноценный язык программирования, столь же сложный, как любой другой, и даже более сложный, чем некоторые. Программисты, пытающиеся решать с помощью JavaScript нетривиальные задачи, часто разочаровываются в процессе разработки из-за того, что недостаточно понимают возможности этого языка. Данная книга содержит всеобъемлющее описание JavaScript, позволяющее вам стать искушенным знатоком. Если прежде вы пользовались справочниками по JavaScript, содержащими готовые рецепты, вас наверняка удивит глубина и подробность изложения материала в последующих главах.

1.2. Версии JavaScript

Подобно любой другой новой технологии программирования, развитие JavaScript в самом начале шло быстрыми темпами. В предыдущих изданиях книги рассказывалось о развитии языка версия за версией и попутно говорилось, в какой версии какие новшества были введены. Однако к настоящему моменту язык стабили-

зировался и был стандартизован ассоциацией европейских производителей компьютеров (European Computer Manufacturer's Association, ECMA).[1] Реализации этого стандарта охватывают интерпретатор JavaScript 1.5 компаний Netscape и Mozilla Foundation, а также интерпретатор Jscript 5.5 корпорации Microsoft. Любые веб-броузеры, выпущенные после Netscape 4.5 или Internet Explorer 4, поддерживают последнюю версию языка. На практике вам едва ли придется столкнуться с интерпретаторами, не совместимыми с этими реализациями.

Обратите внимание, что в соответствии со стандартом ECMA-262 язык официально называется ECMAScript. Но это несколько неудобное название используется только в случае, если необходимо явно сослаться на стандарт. Чисто технически название «JavaScript» относится только к реализации, выполненной Netscape и Mozilla Foundation. Однако на практике все предпочитают использовать это название для обозначения любой реализации JavaScript.

После длительного периода стабильного существования JavaScript появились некоторые признаки изменений. Веб-броузер Firefox 1.5, выпущенный Mozilla Foundation, включает в себя обновленный интерпретатор JavaScript версии 1.6. Данная версия включает новые (нестандартные) методы работы с массивами, которые описываются в разделе 7.7.10, а также обладает поддержкой расширения E4X, которое описывается ниже.

В дополнение к спецификациям ECMA-262, которые стандартизуют ядро языка JavaScript, ассоциация ECMA разработала еще один стандарт, имеющий отношение к JavaScript, – ECMA-357. В этой спецификации было стандартизовано расширение JavaScript, известное под названием E4X, или ECMAScript for XML. С помощью этого расширения в язык была добавлена поддержка нового типа данных – XML – вместе с операторами и инструкциями, позволяющими манипулировать XML-документами. К моменту написания этих строк расширение E4X было реализовано только в JavaScript 1.6 и Firefox 1.5. В данной книге нет формального описания E4X, однако в главе 21 дается расширенное введение в форме практических примеров.

Несколько лет тому назад были внесены предложения к четвертой редакции стандарта ECMA-262, где предполагалось стандартизировать JavaScript 2.0. Эти предложения предусматривают полную перестройку языка, включая введение строгого контроля типов и механизма истинного наследования на основе классов. До настоящего времени наблюдалось некоторое движение по направлению к стандартизации JavaScript 2.0. Однако реализации, выполненные на основе этих предложений, должны включать в себя поддержку языка Microsoft JScript.NET, а также языков ActionScript 2.0 и ActionScript 3.0, используемых в проигрывателе Adobe (ранее Macromedia) Flash. На текущий момент наблюдаются некоторые признаки, свидетельствующие о возобновлении движения к JavaScript 2.0, например выпуск JavaScript 1.6 можно расценивать как один из шагов в этом направлении. Предполагается, что любая новая версия языка будет обратно совместима с версией, описываемой в этой книге. Но даже когда язык JavaScript 2.0 будет стандартизирован, потребуется несколько лет, чтобы его реализации появились во всех веб-броузерах.

---

[1] Стандарт ECMA-262, версия 3 (доступен по адресу *http://www.ecma-international.org/publications/files/ecma-st/ECMA-262.pdf*).

1.3. Клиентский JavaScript

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

Клиентский JavaScript включает в себя интерпретатор JavaScript и объектную модель документа (Document Object Model, DOM), определяемую веб-броузером. Документы могут содержать JavaScript-сценарии, которые в свою очередь могут использовать модель DOM для модификации документа или управления способом его отображения. Другими словами, можно сказать, что клиентский JavaScript позволяет определить поведение статического содержимого веб-страниц. Клиентский JavaScript является основой таких технологий разработки веб-приложений, как DHTML (глава 16), и таких архитектур, как Ajax (глава 20). Введение к главе 13 включает обзор большинства возможностей клиентского JavaScript.

Спецификация ECMA-262 определила стандартную версию базового языка JavaScript, а организация World Wide Web Consortium (W3C) опубликовала спецификацию DOM, стандартизующую возможности, которые броузер должен поддерживать в своей объектной модели. (В главах 15, 16 и 17 содержится более подробное обсуждение этого стандарта.) Основные положения стандарта W3C DOM достаточно полно поддерживаются наиболее распространенными броузерами за одним важным исключением – Microsoft Internet Explorer; в этом броузере отсутствует поддержка механизма обработки событий.

1.3.1. Примеры использования клиентского JavaScript

Веб-броузер, оснащенный интерпретатором JavaScript, позволяет распространять через Интернет исполняемое содержимое в виде JavaScript-сценариев. В примере 1.1 показана простая программа на языке JavaScript, которая и представляет собой сценарий, встроенный в веб-страницу.

Пример 1.1. Простая программа на языке JavaScript

```
<html>
<head><title>Факториалы</title></head>
<body>
<h2>Таблица факториалов</h2>
<script>
var fact = 1;
for(i = 1; i < 10; i++) {
    fact = fact*i;
    document.write(i + "! = " + fact + "<br>");
}
</script>
</body>
</html>
```

После загрузки в броузер, поддерживающий JavaScript, этот сценарий выдаст результат, показанный на рис. 1.1.

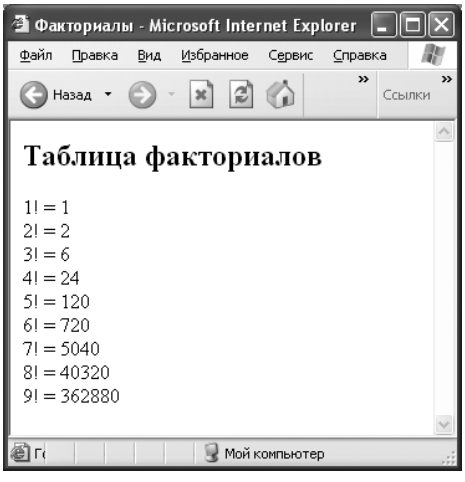

Рис. 1.1. *Веб-страница, сгенерированная с помощью JavaScript*

Как видно из этого примера, для встраивания JavaScript-кода в HTML-файл были использованы теги `<script>` и `</script>`. О теге `<script>` подробнее рассказывается в главе 13. Главное, что демонстрируется в данном примере, – это использование метода `document.write()`.[1] Этот метод позволяет динамически выводить HTML-текст внутри HTML-документа по мере его загрузки веб-броузером.

JavaScript обеспечивает возможность управления не только содержимым HTML-документов, но и их поведением. Другими словами, JavaScript-программа может реагировать на действия пользователя: ввод значения в текстовое поле или щелчок мышью в области изображения в документе. Это достигается путем определения *обработчиков событий* для документа – фрагментов JavaScript-кода, исполняемых при возникновении определенного события, например щелчка на кнопке. В примере 1.2 показан простой фрагмент HTML-кода, который включает в себя обработчик события, вызываемый в ответ на такой щелчок.

Пример 1.2. HTML-кнопка с обработчиком события на языке JavaScript

```
<button onclick="alert('Был зафиксирован щелчок на кнопке');">
Щелкни здесь
</button>
```

На рис. 1.2 показан результат щелчка на кнопке.

Атрибут `onclick` из примера 1.2 – это строка JavaScript-кода, исполняемого, когда пользователь щелкает на кнопке. В данном случае обработчик события `onclick` вызывает функцию `alert()`. Как видно из рис. 1.2, функция `alert()` выводит диалоговое окно с указанным сообщением.

Примеры 1.1 и 1.2 демонстрируют лишь простейшие возможности клиентского JavaScript. Реальная его мощь состоит в том, что сценарии имеют доступ к со-

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

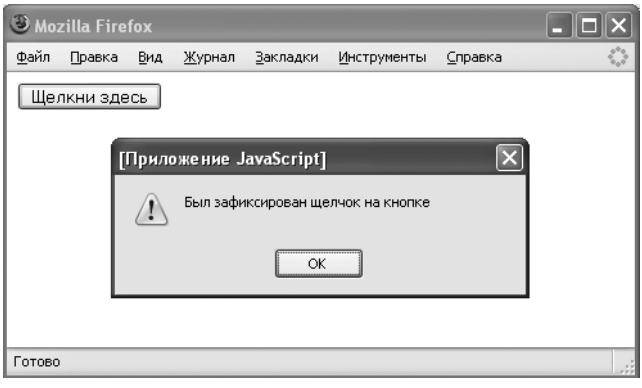

***Рис. 1.2.** Отклик JavaScript на событие*

держимому HTML-документов. В примере 1.3 приводится листинг полноценной нетривиальной JavaScript-программы. Программа вычисляет месячный платеж по закладной на дом или другой ссуде исходя из размера ссуды, процентной ставки и периода выплаты. Программа считывает данные, введенные пользователем в поля HTML-формы, выполняет расчет на основании введенных данных, после чего отображает полученные результаты.

На рис. 1.3 показана HTML-форма в окне веб-броузера. Как видно из рисунка, HTML-документ содержит саму форму и некоторый дополнительный текст. Однако рисунок – это лишь статический снимок окна программы. Благодаря JavaScript-коду она становится динамической: как только пользователь изменяет сумму ссуды, процентную ставку или количество платежей, JavaScript-код заново вычисляет месячный платеж, общую сумму платежей и общий процент, уплаченный за все время ссуды.

Первая половина примера – это HTML-форма, аккуратно отформатированная с помощью HTML-таблицы. Обратите внимание, что обработчики событий

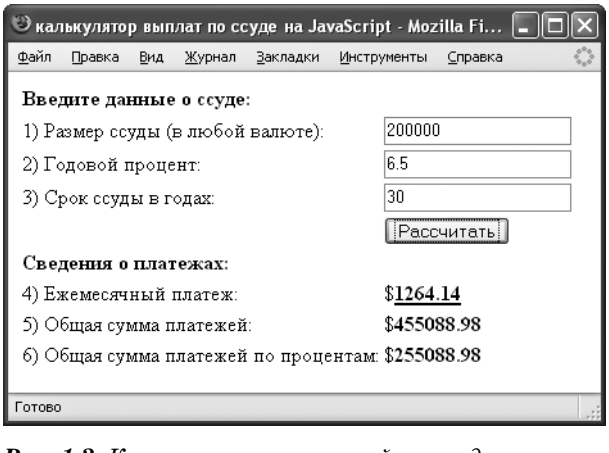

***Рис. 1.3.** Калькулятор платежей по ссуде на языке JavaScript*

onchange и onclick определены лишь для нескольких элементов формы. Веб-броузер запускает эти обработчики в тот момент, когда пользователь изменяет входные данные или щелкает на кнопке Рассчитать, отображаемой на форме. Во всех этих случаях значением атрибута обработчика события является строка JavaScript-кода calculate(). Вызываемый обработчик события исполняет этот код, приводящий к вызову функции calculate().

Функция calculate() определена во второй половине примера внутри тега <script>. Функция читает введенные пользователем данные из формы, выполняет математические действия, требуемые для вычисления платежей по ссуде, и отображает результаты этих действий внутри тегов , каждый из которых имеет уникальный идентификатор, определяемый атрибутом id.

Пример 1.3 прост, но на его внимательное рассмотрение стоит потратить время. Вам не обязательно сейчас понимать весь JavaScript-код, однако комментарии в HTML-, CSS- и JavaScript-коде, а также внимательное изучение этого примера должны дать вам хорошее представление о том, как выглядят программы на клиентском языке JavaScript.[1]

Пример 1.3. Вычисление платежей по ссуде с помощью JavaScript

```
<html>
<head>
<title>калькулятор выплат по ссуде на JavaScript</title>
<style>
/* Это каскадная таблица стилей: с ее помощью определяется внешний вид документа */
.result { font-weight: bold; }      /*стиль отображения элементов с class="result"*/
#payment { text-decoration: underline; }    /* для элемента с id="payment" */
</style>
</head>
<body>
<!-
    Это HTML-форма, дающая пользователю возможность вводить
    данные и с помощью JavaScript показывать ему результат вычислений.
    Элементы формы для улучшения их внешнего вида помещены в таблицу.
    Сама форма имеет имя "loandata", а поля в форме - такие имена,
    как "interest" и "years". Эти имена полей используются
    в JavaScript-коде, следующем за кодом формы.
    Обратите внимание: для некоторых элементов формы
    определены обработчики событий "onchange" и "onclick".
    В них заданы строки JavaScript-кода, выполняемого
    при вводе данных или щелчке на кнопке.
-->
<form name="loandata">
  <table>
    <tr><td><b>Введите данные о ссуде:</b></td></tr>
    <tr>
```

[1] Если интуиция подсказывает вам, что смешивать HTML-, CSS- и JavaScript-код, как в данном примере, не очень хорошо, знайте, что вы не одиноки. В настоящее время в кругах, связанных с веб-дизайном, наблюдается тенденция выделения содержимого, представления и поведения в отдельные файлы. Как это сделать, рассказывается в разделе 13.1.5 главы 13.

```
      <td>1) Размер ссуды (в любой валюте):</td>
      <td><input type="text" name="principal" onchange="calculate();"></td>
    </tr>
    <tr>
      <td>2) Годовой процент:</td>
      <td><input type="text" name="interest" onchange="calculate( );"></td>
    </tr>
    <tr>
      <td>3) Срок ссуды в годах:</td>
      <td><input type="text" name="years" onchange="calculate( );"></td>
    </tr>
    <tr><td></td>
      <td><input type="button" value="Рассчитать"
           onclick="calculate( );"></td>
    </tr>
    <tr><td><b>Сведения о платежах:</b></td></tr>
    <tr>
      <td>4) Ежемесячный платеж:</td>
      <td>$<span class="result" id="payment"></span></td>
    </tr>
    <tr>
      <td>5) Общая сумма платежей:</td>
      <td>$<span class="result" id="total"></span></td>
    </tr>
    <tr>
      <td>6) Общая сумма платежей по процентам:</td>
      <td>$<span class="result" id="totalinterest"></span></td>
    </tr>
  </table>
</form>
<script language="JavaScript">
/*
 * Это JavaScript-функция, которая заставляет пример работать.
 * Обратите внимание: в этом сценарии определяется функция calculate(),
 * вызываемая обработчиками событий в форме. Функция извлекает значения
 * из полей <input> формы, используя имена, определенные в коде, который
 * приведен ранее. Результаты выводятся в именованные элементы <span>
 */
function calculate( ) {
  // Получаем пользовательские данные из формы. Предполагаем, что данные
  // являются корректными. Преобразуем процентную ставку из процентов
  // в десятичное значение. Преобразуем период платежа
  // в годах в количество месячных платежей.
  var principal = document.loandata.principal.value;
  var interest = document.loandata.interest.value / 100 / 12;
  var payments = document.loandata.years.value * 12;

  // Теперь вычисляется сумма ежемесячного платежа.
  var x = Math.pow(1 + interest, payments);
  var monthly = (principal*x*interest)/(x-1);

  // Получить ссылки на именованные элементы <span> формы.
  var payment = document.getElementById("payment");
  var total = document.getElementById("total");
```

```
    var totalinterest = document.getElementById("totalinterest");

    // Убедиться, что результат является конечным числом. Если это так -
    // отобразить результаты, определив содержимое каждого элемента <span>.
    if (isFinite(monthly)) {
      payment.innerHTML = monthly.toFixed(2);
      total.innerHTML = (monthly * payments).toFixed(2);
      totalinterest.innerHTML = ((monthly*payments)-principal).toFixed(2);
    }
    // В противном случае данные, введенные пользователем, по-видимому
    // были некорректны, поэтому ничего не выводится.
    else {
      payment.innerHTML = "";
      total.innerHTML = "";
      totalinterest.innerHTML = "";
    }
  }
  </script>
  </body>
  </html>
```

1.4. Другие области использования JavaScript

JavaScript – это язык программирования общего назначения, и его использование не ограничено веб-броузерами. Изначально JavaScript разрабатывался с прицелом на встраивание в любые приложения и предоставление возможности исполнять сценарии. С самых первых дней веб-серверы компании Netscape включали в себя интерпретатор JavaScript, что позволяло исполнять JavaScript-сценарии на стороне сервера. Аналогичным образом в дополнение к Internet Explorer корпорация Microsoft использует интерпретатор JScript в своем веб-сервере IIS и в продукте Windows Scripting Host. Компания Adobe задействует производный от JavaScript язык для управления своим проигрывателем Flash-файлов. Компания Sun также встроила интерпретатор JavaScript в дистрибутив Java 6.0, что существенно облегчает возможность встраивания сценариев в любое Java-приложение (о том, как это делается, рассказывается в главе 12).

И Netscape, и Microsoft сделали доступными свои реализации интерпретаторов JavaScript для компаний и программистов, желающих включить их в свои приложения. Интерпретатор, созданный в компании Netscape, был выпущен как свободно распространяемое ПО с открытыми исходными текстами и ныне доступен через организацию Mozilla (*http://www.mozilla.org/js/*). Mozilla фактически распространяет две разные версии интерпретатора JavaScript 1.5: один написан на языке C и называется SpiderMonkey, другой написан на языке Java и, что весьма лестно для автора книги, называется Rhino (носорог).

Если вам придется писать сценарии для приложений, включающих интерпретатор JavaScript, первая половина книги, где описываются основы этого языка, будет для вас особенно полезна. Однако информация из глав, в которых описываются особенности конкретных веб-броузеров, скорее всего, будет неприменима для ваших сценариев.

1.5. Изучение JavaScript

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

Наиболее очевидный подход к изучению JavaScript – это написание простых сценариев. Одно из достоинств клиентского JavaScript состоит в том, что любой, кто имеет веб-броузер и простейший текстовый редактор, имеет и полноценную среду разработки. Для того чтобы начать писать программы на JavaScript, нет необходимости в покупке или загрузке специального ПО.

Например, чтобы вместо факториалов вывести последовательность чисел Фибоначчи, пример 1.1 можно переписать следующим образом:

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

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

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

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

Для еще большего упрощения экспериментов с JavaScript можно использовать URL-адрес со спецификатором псевдопротокола `javascript:` для вычисления значения JavaScript-выражения и получения результата. Такой URL-адрес состоит из спецификатора псевдопротокола (`javascript:`), за которым указывается произвольный JavaScript-код (инструкции отделяются одна от другой точками с запятой). Загружая URL-адрес с псевдопротоколом, броузер просто исполняет JavaScript-код. Значение последнего выражения в таком URL-адресе преобразуется в строку, и эта строка выводится веб-броузером в качестве нового документа. Например, для того чтобы проверить свое понимание некоторых операторов и инструкций языка JavaScript, можно набрать следующие URL-адреса в адресном поле веб-броузера:

```
javascript:5%2
javascript:x = 3; (x < 5)? "значение x меньше": "значение x больше"
javascript:d = new Date(); typeof d;
javascript:for(i=0,j=1,k=0,fib=1; i<5; i++,fib=j+k,k=j,j=fib) alert(fib);
javascript:s=""; for(i in navigator) s+=i+":"+navigator[i]+"\n"; alert(s);
```

В веб-броузере Firefox однострочные сценарии вводятся в JavaScript-консоли, доступ к которой можно получить из меню Инструменты. Просто введите выражение или инструкцию, которую требуется проверить. При использовании JavaScript-консоли спецификатор псевдопротокола (`javascript:`) можно опустить.

Не любой код, написанный вами при изучении JavaScript, будет работать так, как ожидается, и вам захочется его отладить. Базовая методика отладки JavaScript-кода совпадает с методикой для многих других языков: вставка в код инструкций, которые будут выводить значения нужных переменных так, чтобы можно было понять, что же на самом деле происходит. Как мы уже видели, иногда для этих целей можно использовать метод `document.write()` или `alert()`. (Более сложный способ отладки, основанный на выводе отладочных сообщений в файл, приводится в примере 15.9.)

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

Если вам постоянно приходится сталкиваться с ошибками в JavaScript-сценариях, вероятно, вас заинтересует настоящий отладчик JavaScript. В Internet Explorer можно воспользоваться отладчиком Microsoft Script Debugger, в Firefox – модулем расширения, известным под названием Venkman. Описание этих инструментов выходит далеко за рамки темы этой книги, но вы без труда найдете его в Интернете, воспользовавшись какой-нибудь поисковой системой. Еще один инструмент, который, строго говоря, не является отладчиком, – это *jslint*; он способен отыскивать распространенные ошибки в JavaScript-коде программ (*http://jslint.com*).

I

Основы JavaScript

Данная часть книги включает главы со 2 по 12 и описывает базовый язык JavaScript. Этот материал задуман как справочный, и прочитав главы этой части один раз, вы, возможно, будете неоднократно возвращаться к ним, чтобы освежить в памяти некоторые особенности языка.

- Глава 2 «Лексическая структура»
- Глава 3 «Типы данных и значения»
- Глава 4 «Переменные»
- Глава 5 «Выражения и операторы»
- Глава 6 «Инструкции»
- Глава 7 «Объекты и массивы»
- Глава 8 «Функции»
- Глава 9 «Классы, конструкторы и прототипы»
- Глава 10 «Модули и пространства имен»
- Глава 11 «Шаблоны и регулярные выражения»
- Глава 12 «Разработка сценариев для Java-приложений»

2

Лексическая структура

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

2.1. Набор символов

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

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

Стандарт ECMAScript v3 допускает наличие Unicode-символов в любом месте JavaScript-программы. Однако версии 1 и 2 стандарта допускают использование Unicode-символов только в комментариях и строковых литералах, заключенных в кавычки, все остальные составляющие программы ограничены набором

ASCII-символов.[1] Версии JavaScript, предшествующие стандарту ECMAScript, обычно вообще не поддерживают Unicode.

2.2. Чувствительность к регистру

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

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

2.3. Символы-разделители и переводы строк

JavaScript игнорирует пробелы, табуляции и переводы строк, присутствующие между лексемами в программе. Поэтому символы пробела, табуляции и перевода строки могут без ограничений использоваться в исходных текстах программ для форматирования и придания им удобочитаемого внешнего вида. Однако имеется небольшое ограничение, которое касается символов перевода строк и о котором рассказывается в следующем разделе.

2.4. Необязательные точки с запятой

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

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

[1] Для русскоязычных программистов это означает, что а) русскоязычный текст может появляться только в комментариях и в строковых литералах, предназначенных непосредственно для вывода; б) такие тексты представляются в кодировке UTF-16 (Unicode – это единая система связывания символов любого языка с однозначным численным кодом, а для кодирования этого численного кода могут применяться различные кодировки, например UTF-8, UTF-16 и др.); в) все остальные лексемы программы – операторы, имена переменных и т. д. – должны состоять из латинских литер; это достаточно обычная и привычная практика и для других языков программирования. – *Примеч. науч. ред.*

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

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

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

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

```
return
true;
```

Синтаксический анализатор JavaScript предполагает, что программист имеет в виду следующее:

```
return;
true;
```

Хотя на самом деле программист, видимо, хотел написать

```
return true;
```

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

```
break
outerloop;
```

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

2.5. Комментарии

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

```
// Это однострочный комментарий.
/* Это тоже комментарий */ // а это другой комментарий.
/*
```

```
 * Это еще один комментарий.
 * Он располагается в нескольких строках.
 */
```

2.6. Литералы

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

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

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

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

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

2.7. Идентификаторы

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

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

В ECMAScript v3 идентификаторы могут содержать буквы и цифры из полного набора символов Unicode. До этой версии стандарта JavaScript-идентификаторы были ограничены набором ASCII. ECMAScript v3 также допускает наличие

[1] Знак $ недопустим в идентификаторах для более ранних версий, чем JavaScript 1.1. Этот знак предназначен только для средств генерации кода, поэтому следует избегать его использования в идентификаторах.

в идентификаторах escape-последовательностей Unicode – символов \u, за которыми расположены 4 шестнадцатеричные цифры, обозначающие 16-разрядный код символа. Например, идентификатор π можно записать как \u03c0. Этот синтаксис неудобен, но обеспечивает возможность транслитерации JavaScript-программ с Unicode-символами в форму, допускающую работу с ними в текстовых редакторах и других средствах, не поддерживающих полный набор Unicode.

Наконец, идентификаторы не могут совпадать ни с одним из ключевых слов, предназначенных в JavaScript для других целей. В следующем разделе перечислены ключевые слова, зарезервированные для специальных нужд JavaScript.

2.8. Зарезервированные слова

В JavaScript имеется несколько зарезервированных слов. Они не могут быть идентификаторами (именами переменных, функций и меток циклов) в JavaScript-программах. В табл. 2.1 перечислены ключевые слова, стандартизованные в ECMAScript v3. Для интерпретатора JavaScript они имеют специальное значение, т. к. являются частью синтаксиса языка.

Таблица 2.1. Зарезервированные ключевые слова JavaScript

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

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

Таблица 2.2. Слова, зарезервированные для расширений ECMA

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

Помимо нескольких только что перечисленных формально зарезервированных слов текущие проекты стандарта ECMAScript v4 рассматривают применение ключевых слов as, is, namespace и use. Хотя текущие интерпретаторы JavaScript не запрещают использование этих четырех слов в качестве идентификаторов, однако все равно следует этого избегать.

Кроме того, следует избегать использования идентификаторов глобальных переменных и функций, предопределенных в языке JavaScript. Если попытаться создать переменную или функцию с таким идентификатором, то это будет приводить либо к ошибке (если свойство определено как доступное только для чтения), либо к переопределению глобальной переменной или функции, чего точно не стоит делать, если вы не стремитесь к этому преднамеренно. В табл. 2.3 перечислены имена глобальных переменных и функций, определяемых стандартом ECMAScript v 3. Конкретные реализации могут содержать свои предопределенные элементы с глобальной областью видимости, кроме того, каждая конкретная платформа JavaScript (клиентская, серверная и прочие) может еще больше расширять этот список.[1]

Таблица 2.3. Другие идентификаторы, которых стоит избегать

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

[1] При описании объекта Window в четвертой части книги приведен список глобальных переменных и функций, определенных в клиентском JavaScript.

3

Типы данных и значения

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

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

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

Помимо функций и массивов в базовом языке JavaScript определено еще несколько специальных видов объектов. Эти объекты представляют собой не новые типы данных, а лишь новые *классы* (*classes*) объектов. Класс `Date` определяет объекты, представляющие даты, класс `RegExp` – объекты, представляющие регуляр-

ные выражения (мощное средство поиска по шаблону, описываемое в главе 11), и класс `Error` – объекты, представляющие синтаксические ошибки и ошибки времени выполнения, которые могут возникать в JavaScript-программе.

В оставшейся части этой главы подробно описан каждый из элементарных типов данных. В ней также приведены начальные сведения об объектах, массивах и функциях, которые более подробно рассмотрены в главах 7 и 8. И наконец, в ней приведен обзор классов `Date`, `RegExp` и `Error`, подробно документируемых в III части книги. Глава содержит некоторые узкоспециализированные подробности, которые можно пропустить при первом прочтении.

3.1. Числа

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

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

3.1.1. Целые литералы

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

```
0
3
10000000
```

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

3.1.2. Шестнадцатеричные и восьмеричные литералы

Помимо десятичных целых литералов JavaScript распознает шестнадцатеричные значения (по основанию 16). Шестнадцатеричные литералы начинаются с последовательности символов «0x» или «0X», за которой следует строка шест-

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

надцатеричных цифр. Шестнадцатеричная цифра – это одна из цифр от 0 до 9 или букв от a (или A) до f (или F), представляющих значения от 10 до 15. Ниже приводятся примеры шестнадцатеричных целых литералов:

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

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

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

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

3.1.3. Литералы вещественных чисел

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

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

Более лаконичное определение синтаксиса таково:

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

Например:

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

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

3.1.4. Работа с числами

Для работы с числами в JavaScript-программах используются поддерживаемые языком арифметические операторы, к которым относятся операторы сложения

(+), вычитания (-), умножения (*) и деления (/). Подробное описание этих и других арифметических операторов имеется в главе 5.

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

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

А так вычисляется квадратный корень числового выражения:

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

Подробные сведения обо всех математических функциях, поддерживаемых JavaScript, приведены в описании объекта `Math` и соответствующих листингах третьей части книги.

3.1.5. Преобразования чисел

В языке JavaScript имеется возможность представлять числа в виде строк и преобразовывать строки в числа. Порядок этих преобразований описывается в разделе 3.2.

3.1.6. Специальные числовые значения

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

Еще одно специальное числовое значение возвращается JavaScript, когда математическая операция (например, деление нуля на ноль) приводит к неопределенному результату или ошибке. В этом случае результатом является специальное значение «нечисло», обозначаемое как `NaN`. «Нечисло» (Not-a-Number) ведет себя необычно: оно не равно ни одному другому числу, в том числе и самому себе! По данной причине для проверки на это значение имеется специальная функция `isNaN()`. Похожая функция, `isFinite()`, позволяет проверить число на неравенство `NaN` или положительной/отрицательной бесконечности.

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

Таблица 3.1. Специальные числовые константы

Константа	Значение
`Infinity`	Специальное значение, обозначающее бесконечность
`NaN`	Специальное значение – «нечисло»
`Number.MAX_VALUE`	Максимальное представимое значение

Константа	Значение
Number.MIN_VALUE	Наименьшее (ближайшее к нулю) представимое значение
Number.NaN	Специальное значение – «нечисло»
Number.POSITIVE_INFINITY	Специальное значение, обозначающее плюс бесконечность
Number.NEGATIVE_INFINITY	Специальное значение, обозначающее минус бесконечность

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

3.2. Строки

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

3.2.1. Строковые литералы

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

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

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

Обратите внимание: ограничивая строку одинарными кавычками, необходимо проявлять осторожность в обращении с апострофами, употребляемыми в анг-

лийском языке для обозначения притяжательного падежа и в сокращениях, как, например, в словах «can't» и «O'Reilly's». Поскольку апостроф и одиночная кавычка – это одно и то же, необходимо при помощи символа обратного слэша (\) экранировать апострофы, расположенные внутри одиночных кавычек (подробнее об этом – в следующем разделе).

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

```
<a href="" onclick="alert('Спасибо')">Щелкни на мне</a>
```

3.2.2. Управляющие последовательности в строковых литералах

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

Другой пример, упомянутый в предыдущем разделе, – это последовательность `\'`, обозначающая символ одинарной кавычки. Эта управляющая последовательность необходима для включения символа одинарной кавычки в строковый литерал, заключенный в одинарные кавычки. Теперь становится понятно, почему мы называем эти последовательности управляющими – здесь символ обратного слэша позволяет управлять интерпретацией символа одинарной кавычки. Вместо того чтобы отмечать ею конец строки, мы используем ее как апостроф:

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

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

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

но такие управляющие последовательности не поддерживаются в стандарте ECMAScript v3 и не должны использоваться.

Таблица 3.2. Управляющие последовательности JavaScript

Константа	Значение
\0	Символ NUL (\u0000)
\b	«Забой» (\u0008)
\t	Горизонтальная табуляция (\u0009)
\n	Перевод строки (\u000A)
\v	Вертикальная табуляция (\u000B)
\f	Перевод страницы (\u000C)
\r	Возврат каретки (\u000D)
\"	Двойная кавычка (\u0022)
\'	Одинарная кавычка (\u0027)
\\	Обратный слэш (\u005C)
\x*XX*	Символ Latin-1, заданный двумя шестнадцатеричными цифрами *XX*
\u*XXXX*	Unicode-символ, заданный четырьмя шестнадцатеричными цифрами *XXXX*
XXX	Символ из набора Latin-1, заданный тремя восьмеричными цифрами *XXX*, с кодом в диапазоне от 1 до 377. Не поддерживается ECMAScript v3; такой способ записи не должен использоваться

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

3.2.3. Работа со строками

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

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

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

```
s.length
```

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

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

Чтобы извлечь второй, третий и четвертый символы из строки `s`, применяется инструкция:

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

Определить позицию первого символа «a» в строке `s` можно следующим образом:

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

Есть и еще ряд методов, которые можно использовать при работе со строками. Полностью эти методы документированы в описании объекта `String` и в листингах третьей части книги.

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

В некоторых реализациях JavaScript отдельные символы могут извлекаться из строк (но не записываться в строки) при обращении к строкам как к массивам, в результате приведенный ранее вызов метода `charAt()` может быть записан следующим образом:

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

Однако этот синтаксис не стандартизован в ECMAScript v3, не является переносимым и его следует избегать.

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

3.2.4. Преобразование чисел в строки

Преобразование чисел в строки производится автоматически, по мере необходимости. Например, если число используется в операции конкатенации строк, оно будет преобразовано в строку:

```
var n = 100;
var s = n + " бутылок пива на стене.";
```

Такая способность JavaScript к автоматическому преобразованию числа в строку реализует идиому программирования, которую часто можно встретить на практике: чтобы преобразовать число в строку, достаточно просто сложить его с пустой строкой:

```
var n_as_string = n + "";
```

Для явного преобразования числа в строку используется функция `String()`:

```
var string_value = String(number);
```

Еще один способ преобразования числа в строку заключается в вызове метода `toString()`:

```
string_value = number.toString( );
```

Метод `toString()` объекта `Number` (примитивы чисел автоматически преобразуются в объекты типа `Number`, благодаря чему можно воспользоваться этим методом) может принимать один необязательный аргумент, который определяет базу, или основание, системы счисления для преобразования. Если основание системы счисления не указывается, по умолчанию она предполагается равной 10. Однако существует возможность выполнять преобразования и в других системах счисления (от 2 до 36)[1], например:

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

Одним из недостатков реализаций JavaScript, существовавших до версии JavaScript 1.5, было отсутствие встроенной возможности определить число десятичных знаков, которые должны получиться в результате, или задать результат в экспоненциальном представлении. В связи с этим могут возникать определенные сложности с представлением чисел в традиционных форматах, таких как денежные суммы.

В стандарте ECMAScript v3 и JavaScript 1.5 эта проблема была решена за счет добавления нового метода преобразования числа в строку в классе `Number`. Метод `toFixed()` преобразует число в строку и отображает определенное число знаков после десятичной точки. Однако данный метод не выполняет преобразование числа в экспоненциальную форму. Эту задачу решает метод `toExponential()`, который преобразует число в экспоненциальное представление с одним знаком перед точкой и с заданным числом десятичных знаков после точки. Для отображения определенного числа значащих разрядов числа используется метод `toPrecision()`. Он возвращает строку с экспоненциальным представлением числа, если заданного количества значащих разрядов недостаточно для точного отображения целой части числа. Обратите внимание: все три метода корректно выполняют округление результата. Ниже приводятся примеры обращения к этим методам:

```
var n = 123456.789;
n.toFixed(0);        // "123457"
n.toFixed(2);        // "123456.79"
n.toExponential(1); // "1.2e+5"
n.toExponential(3); // "1.235e+5"
n.toPrecision(4);    // "1.235e+5"
n.toPrecision(7);    // "123456.8"
```

[1] Спецификациями ECMAScript предусматривается возможность определения основания системы счисления в методе `toString()`, но при этом допускается возвращать из метода строку в представлении, зависящем от конкретной реализации, если основание не равно 10. Таким образом, согласно стандарту метод может просто игнорировать значение аргумента и всегда возвращать число в десятичном представлении. Однако на практике большинство реализаций возвращают корректный результат с учетом заданного основания системы счисления.

3.2.5. Преобразование строк в числа

Когда строка используется в числовом контексте, она автоматически преобразуется в число. Например, следующее выражение является вполне допустимым:

```
var product = "21" * "2";    // в результате получится число 42.
```

Это обстоятельство можно взять на вооружение при необходимости преобразовать строку в число; для этого достаточно просто вычесть из строки значение 0:

```
var number = string_value - 0;
```

(Будьте внимательны: операция сложения в данной ситуации будет интерпретирована как операция конкатенации строк.)

Менее изощренный и более прямолинейный способ преобразования строки в число заключается в обращении к конструктору `Number()` как к обычной функции:

```
var number = Number(string_value);
```

Недостаток такого способа преобразования строки в число заключается в его чрезмерной строгости. Этот способ может использоваться только для преобразования десятичных чисел, и хотя он допускает наличие ведущих и оконечных символов пробела, появление других нецифровых символов после числа в строке недопустимо.

Более гибкий способ преобразования обеспечивается функциями `parseInt()` и `parseFloat()`. Эти функции преобразуют и возвращают произвольные числа, стоящие в начале строки, игнорируя любые нецифровые символы, расположенные вслед за числом. Функция `parseInt()` выполняет только целочисленное преобразование, тогда как `parseFloat()` может преобразовывать как целые, так и вещественные числа. Если строка начинается с символов «0x» или «0X», функция `parseInt()` интерпретирует строку как шестнадцатеричное число.[1] Например:

```
parseInt("3 слепых мышки");  // Вернет 3
parseFloat("3.14 метров");   // Вернет 3.14
parseInt("12.34");           // Вернет 12
parseInt("0xFF");            // Вернет 255
```

В качестве второго аргумента функция `parseInt()` может принимать основание системы счисления. Корректными значениями являются числа в диапазоне от 2 до 36, например:

```
parseInt("11", 2);    // Вернет 3 (1*2 + 1)
parseInt("ff", 16);   // Вернет 255 (15*16 + 15)
parseInt("zz", 36);   // Вернет 1295 (35*36 + 35)
parseInt("077", 8);   // Вернет 63 (7*8 + 7)
parseInt("077", 10);  // Вернет 77 (7*10 + 7)
```

[1] Стандарт ECMAScript утверждает, что если строка начинается с символа «0» (но не «0x» или «0X»), функция `parseInt()` может интерпретировать строку как число и в восьмеричном, и в десятичном представлении. Поскольку поведение функции четко не определено, следует избегать использования функции `parseInt()` для интерпретации строк, начинающихся с «0», или явно указывать основание системы счисления.

Если методы `parseInt()` и `parseFloat()` оказываются не в состоянии выполнить преобразование, они возвращают значение `NaN`:

```
parseInt("eleven");    // Вернет NaN
parseFloat("$72.47");  // Вернет NaN
```

3.3. Логические значения

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

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

```
a == 4
```

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

Логические значения обычно используются в управляющих конструкциях JavaScript. Например, инструкция `if/else` в JavaScript выполняет одно действие, если логическое значение равно `true`, и другое действие, если `false`. Обычно сравнение, создающее логическое значение, непосредственно объединяется с инструкцией, в которой оно используется. Результат выглядит так:

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

Здесь выполняется проверка, равна ли переменная `a` числу 4. Если да, к значению переменной `b` добавляется 1; в противном случае число 1 добавляется к значению переменной `a`.

Вместо того чтобы интерпретировать два возможных логических значения как `true` и `false`, иногда удобно рассматривать их как «включено» (`true`) и «выключено» (`false`) или «да» (`true`) и «нет» (`false`).

3.3.1. Преобразование логических значений

Логические значения легко преобразуются в значения других типов, причем нередко такое преобразование выполняется автоматически.[1] Если логическое значе-

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

ние используется в числовом контексте, тогда значение `true` преобразуется в число 1, а `false` – в 0. Если логическое значение используется в строковом контексте, тогда значение `true` преобразуется в строку `"true"`, а `false` – в строку `"false"`.

Когда в качестве логического значения используется число, оно преобразуется в значение `true`, если оно не равно значениям 0 или `NaN`, которые преобразуются в логическое значение `false`. Когда в качестве логического значения используется строка, она преобразуется в значение `true`, если это не пустая строка, в противном случае в результате преобразования получается значение `false`. Специальные значения `null` и `undefined` преобразуются в `false`, а любые функция, объект или массив, значения которых отличны от `null`, преобразуются в `true`.

Если вы предпочитаете выполнять преобразование явно, можно воспользоваться функцией `Boolean()`:

```
var x_as_boolean = Boolean(x);
```

Другой способ явного преобразования заключается в использовании двойного оператора логического отрицания:

```
var x_as_boolean = !!x;
```

3.4. Функции

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

JavaScript-программы могут также определять собственные функции, содержащие, например, такой код:

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

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

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

Важной чертой JavaScript является то, что функции представляют собой значения, которыми можно манипулировать в JavaScript-коде. Во многих языках, в том числе в Java, функции – это всего лишь синтаксические элементы языка, но не тип данных: их можно определять и вызывать. То обстоятельство, что функции

в JavaScript представляют собой настоящие значения, придает языку большую гибкость. Это означает, что функции могут храниться в переменных, массивах и объектах, а также передаваться в качестве аргументов другим функциям. Очень часто это бывает очень удобно. Более подробно об определении и вызове функций, а также об использовании их в качестве значений рассказывается в главе 8.

Поскольку функции представляют собой значения, такие же, как числа и строки, они могут присваиваться свойствам объектов. Когда функция присваивается свойству объекта (объектный тип данных и свойства объекта описаны в разделе 3.5), она часто называется *методом* этого объекта. Методы – важная часть объектно-ориентированного программирования. Им посвящена глава 7.

3.4.1. Функциональные литералы

В предыдущем разделе мы видели определение функции `square()`. С помощью этого синтаксиса обычно описывается большинство функций в JavaScript-программах. Однако стандарт ECMAScript v3 предоставляет синтаксис (реализованный в JavaScript 1.2 и более поздних версиях) для определения функциональных литералов. Функциональный литерал задается с помощью ключевого слова `function`, за которым следуют необязательное имя функции, список аргументов функции, заключенный в круглые скобки, и тело функции в фигурных скобках. Другими словами, функциональный литерал выглядит так же, как определение функции, правда, у него может не быть имени. Самое большое различие состоит в том, что функциональные литералы могут входить в другие JavaScript-выражения. То есть функцию `square()` не обязательно задавать в виде определения:

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

Ее можно задать с помощью функционального литерала:

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

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

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

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

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

3.5. Объекты

Объект – это коллекция именованных значений, которые обычно называют *свойствами* (*properties*) объекта. (Иногда они называются *полями* объекта, но упо-

требление этого термина может сбить с толку.) Чтобы сослаться на свойство объекта, надо указать имя объекта, затем точку и имя свойства. Например, если объект под названием `image` имеет свойства `width` и `height`, мы можем сослаться на эти свойства следующим образом:

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

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

```
document.myform.button
```

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

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

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

Объекты в JavaScript могут выступать в качестве ассоциативных массивов, т. е. могут ассоциировать произвольные значения с произвольными строками. При такой работе с объектом обычно требуется другой синтаксис для доступа к его свойствам: строка, содержащая имя требуемого свойства, заключается в квадратные скобки. Тогда к свойствам объекта `image`, упомянутого ранее, можно обратиться посредством следующего кода:

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

Ассоциативные массивы – это мощный тип данных; они полезны при реализации ряда технологий программирования. Об объектах, их традиционном применении и применении в качестве ассоциативных массивов рассказано в главе 7.

3.5.1. Создание объектов

Как мы увидим в главе 7, объекты создаются путем вызова специальных функций-конструкторов. Все следующие строки создают новые объекты:

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

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

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

3.5.2. Объектные литералы

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

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

Объектные литералы могут быть вложенными. Например:

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

Наконец, значениями свойств в объектных литералах не обязательно должны быть константы – это могут быть произвольные JavaScript-выражения. Кроме того, в качестве имен свойств в объектных литералах допускается использовать строковые значения:

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

3.5.3. Преобразование объектов

Когда непустой объект используется в логическом контексте, результатом преобразования является значение `true`. Когда объект используется в строковом контексте, преобразование выполняется методом `toString()` объекта и в дальнейших вычислениях участвует строка, возвращаемая этим методом. Когда объект используется в числовом контексте, сначала вызывается метод объекта `valueOf()`. Если этот метод возвращает числовое значение примитивного типа, в дальнейших вычислениях участвует это значение. Однако в большинстве случаев метод `valueOf()` возвращает сам объект. В такой ситуации объект сначала преобразуется в строку вызовом метода `toString()`, а затем выполняется попытка преобразовать строку в число.

Проблема преобразования объектов в значения примитивных типов имеет свои тонкости, и мы еще вернемся к ней в конце главы.

3.6. Массивы

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

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

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

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

Обратите внимание: описываемые здесь массивы отличаются от ассоциативных массивов (см. раздел 3.5). Здесь обсуждаются «настоящие» массивы, которые индексируются неотрицательными целыми числами. Ассоциативные массивы индексируются строками. Следует также отметить, что в JavaScript не поддерживаются многомерные массивы (хотя допускается существование массивов из массивов). И наконец, поскольку JavaScript является нетипизированным языком, элементы массива не обязательно должны иметь одинаковый тип, как в типизированных языках, подобных Java. Подробнее о массивах мы поговорим в главе 7.

3.6.1. Создание массивов

Массив может быть создан с помощью функции-конструктора `Array()`. Созданному массиву допустимо присваивать любое количество индексированных элементов:

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

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

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

Если передать конструктору `Array()` только одно число, оно определит длину массива. Таким образом, следующее выражение создает новый массив с 10 неопределенными элементами:

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

3.6.2. Литералы массивов

В JavaScript определяется синтаксис литералов для создания и инициализации массивов. Литерал, или инициализатор, массива – это список разделенных запятыми значений, заключенных в квадратные скобки. Значения в скобках последовательно присваиваются элементам массива с индексами, начиная с нуля. Например, программный код, создающий и инициализирующий массив из предыдущего раздела, можно записать следующим образом:

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

Как и объектные литералы, литералы массивов могут быть вложенными:

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

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

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

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

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

3.7. Значение null

Ключевое слово `null` в JavaScript имеет специальный смысл. Обычно считается, что у значения `null` объектный тип и оно говорит об отсутствии объекта. Значение `null` уникально и отличается от любых других. Если переменная равна `null`, следовательно, в ней не содержится допустимого объекта, массива, числа, строки или логического значения.[1]

Когда значение `null` используется в логическом контексте, оно преобразуется в значение `false`, в числовом контексте оно преобразуется в значение 0, а в строковом контексте — в строку `"null"`.

3.8. Значение undefined

Еще одно специальное значение, иногда используемое в JavaScript, – `undefined`. Оно возвращается при обращении либо к переменной, которая была объявлена, но которой никогда не присваивалось значение, либо к свойству объекта, которое не существует. Заметьте, что специальное значение `undefined` – это не то же самое, что `null`.

Хотя значения `null` и `undefined` не эквивалентны друг другу, оператор эквивалентности `==` считает их равными. Рассмотрим следующее выражение:

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

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

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

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

```
var undefined;
```

---

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

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

Когда значение undefined используется в логическом контексте, оно преобразуется в значение false. В числовом контексте – в значение NaN, а в строковом – в строку "undefined".

3.9. Объект Date

В предыдущих разделах мы описали все фундаментальные типы данных, поддерживаемые JavaScript. Значения даты и времени не относятся к этим фундаментальным типам, однако в JavaScript имеется класс объектов, представляющих дату и время, и этот класс можно использовать для работы с этим типом данных. Объект Date в JavaScript создается с помощью оператора new и конструктора Date() (оператор new будет введен в главе 5, а в главе 7 вы больше узнаете о создании объектов):

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

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

```
xmas.setFullYear(xmas.getFullYear() + 1);   // Заменяем дату датой следующего Рождества.
var weekday = xmas.getDay();               // В 2007 году Рождество выпадает на вторник.
document.write("Сегодня: " + now.toLocaleString());  // Текущие дата и время.
```

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

Полное описание объекта Date и его методов вы найдете в третьей части книги.

3.10. Регулярные выражения

Регулярные выражения предоставляют богатый и мощный синтаксис описания текстовых шаблонов. Они применяются для поиска соответствия заданному шаблону и реализации операций поиска и замены. В JavaScript для формирования регулярных выражений принят синтаксис языка Perl.

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

Однако в отличие от объекта Date, объекты RegExp имеют синтаксис литералов и могут задаваться непосредственно в коде JavaScript-программы. Текст между парой символов слэша образует литерал регулярного выражения. За вторым

символом слэша в паре могут также следовать одна или несколько букв, изменяющих смысл шаблона. Например:

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

Грамматика регулярных выражений сложна и подробно описана в главе 11. Сейчас вам важно лишь знать, как литерал регулярного выражения выглядит в JavaScript-коде.

3.11. Объекты Error

В ECMAScript v3 определяется несколько классов для представления ошибок. При возникновении ошибки времени выполнения интерпретатор JavaScript «генерирует» объект одного из этих типов. (Вопросы генерации и перехвата ошибок обсуждаются в главе 6 при описании инструкций `throw` и `try`.) Каждый объект ошибки имеет свойство `message`, которое содержит зависящее от реализации сообщение об ошибке. Заранее определены следующие типы объектов ошибок – `Error`, `EvalError`, `RangeError`, `ReferenceError`, `SyntaxError`, `TypeError` и `URIError`. Подробнее об этих классах рассказывается в третьей части книги.

3.12. Преобразование типов

Поскольку все типы данных уже обсуждались в предыдущих разделах, здесь мы рассмотрим, как значения каждого типа преобразуются в значения других типов. Основное правило заключается в следующем: если значение одного типа используется в контексте, требующем значение некоего другого типа, интерпретатор JavaScript автоматически пытается преобразовать это значение. Так, например, если в логическом контексте используется число, оно преобразуется в значение логического типа. Если в контексте строки используется объект, он преобразуется в строковое значение. Если в числовом контексте используется строка, интерпретатор JavaScript пытается преобразовать ее в число. В табл. 3.3 приводится информация о том, как производится преобразование значений, когда значение некоторого типа задействовано в определенном контексте.

Таблица 3.3. Автоматическое преобразование типов

Тип значения	Контекст, в котором используется значение			
	Строковый	**Числовой**	**Логический**	**Объектный**
Неопределенное значение	`"undefined"`	`NaN`	`false`	**Error**
`null`	`"null"`	`0`	`false`	**Error**
Непустая строка	**Как есть**	Числовое значение строки или `NaN`	`true`	**Объект** `String`
Пустая строка	**Как есть**	`0`	`false`	**Объект** `String`
`0`	`"0"`	**Как есть**	`false`	**Объект** `Number`
`NaN`	`"NaN"`	**Как есть**	`false`	**Объект** `Number`

Таблица 3.3 (продолжение)

Тип значения	Контекст, в котором используется значение			
	Строковый	Числовой	Логический	Объектный
Infinity	"Infinity"	Как есть	true	Объект Number
-Infinity	"-Infinity"	Как есть	true	Объект Number
Любое другое число	Строковое представление числа	Как есть	true	Объект Number
true	"true"	1	Как есть	Объект Boolean
false	"false"	0	Как есть	Объект Boolean
Объект	toString()	valueOf(), toString() или NaN	true	Как есть

3.13. Объекты-обертки для элементарных типов данных

Ранее в этой главе мы обсуждали строки, и тогда я обратил ваше внимание на странную особенность этого типа данных: для работы со строками используется объектная нотация.[1] Например, типичная операция со строками может выглядеть следующим образом:

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

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

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

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

JavaScript может гибко преобразовывать один тип в другой. Когда мы используем строку в объектном контексте, т. е. когда пытаемся обратиться к свойству или методу строки, JavaScript создает внутри себя объект-обертку для строкового значения. Этот объект String используется вместо базового строкового значения. Для объекта определены свойства и методы, поэтому удается задействовать зна-

[1] Этот раздел содержит достаточно сложный материал, который при первом прочтении можно пропустить.

чение базового типа в объектном контексте. То же самое, конечно, верно и для других базовых типов и соответствующих им объектов-оберток; мы просто не работаем с другими типами в объектном контексте так же часто, как со строками.

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

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

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

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

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

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

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

Имейте в виду, все, что говорилось в этом разделе о строковых значениях и объектах String, также применимо к числовым и логическим значениям и соответствующим объектам Number и Boolean. Более подробную информацию об этих классах можно получить из соответствующих статей в третьей части книги.

Наконец, следует отметить, что любые строки, числа или логические значения могут быть преобразованы в соответствующий объект-обертку с помощью функции Object():

```
var number_wrapper = Object(3);
```

---

[1] Однако при этом метод eval() рассматривает строковые значения и объекты String по-разному, и если непреднамеренно передать ему объект String вместо значения базового строкового типа, он поведет себя не так, как вы предполагаете.

3.14. Преобразование объектов в значения элементарных типов

Обычно объекты преобразуются в значения элементарных типов достаточно прямолинейным способом, о котором рассказывалось в разделе 3.5.3. Однако на вопросах преобразования объектов следует остановиться более подробно.[1]

Прежде всего следует заметить, что попытка преобразования непустых объектов в логическое значение дает в результате значение `true`. Это справедливо для любых объектов (включая массивы и функции), даже для объектов-оберток, которые представляют элементарные типы, при другом способе преобразования дающие значение `false`. Например, все нижеследующие объекты преобразуются в значение `true` при использовании их в логическом контексте:

```
new Boolean(false) // Внутреннее значение - false,  но объект преобразуется в true
new Number(0)
new String("")
new Array( )
```

В табл. 3.3 описывается порядок преобразования объектов в числовые значения, когда сначала вызывается метод объекта `valueOf()`. Большинство объектов по умолчанию наследуют метод `valueOf()` от базового объекта `Object`, который возвращает сам объект. Поскольку по умолчанию метод `valueOf()` не возвращает значение элементарного типа, далее интерпретатор JavaScript пытается преобразовать объект в число вызовом метода `toString()` с последующим преобразованием строки в число.

В случае массивов это приводит к весьма интересным результатам. Метод `toString()` у массивов преобразует элементы массива в строки и возвращает результат операции конкатенации этих строк, разделяя отдельные элементы массива запятыми. Таким образом, пустой массив преобразуется в пустую строку, что в результате дает число 0! Кроме того, если массив состоит из единственного элемента, содержащего число *n*, весь массив преобразуется в строковое представление числа *n*, которое затем будет вновь преобразовано в число *n*. Если массив содержит более одного элемента или единственный элемент массива не является числом, результатом преобразования будет значение `NaN`.

Тип преобразования зависит от контекста, в котором это преобразование производится. Существуют такие ситуации, когда невозможно однозначно определить контекст. Оператор `+` и операторы сравнения (`<`, `<=`, `>` и `>=`) могут оперировать как числами, так и строками, таким образом, когда в одной из таких операций участвует объект, возникает неоднозначность: в значение какого типа следует преобразовать объект – в строку или в число. В большинстве случаев интерпретатор JavaScript сначала пытается преобразовать объект с помощью метода `valueOf()`. Если этот метод возвращает значение элементарного типа (как правило, число), тогда используется это значение. Однако чаще всего метод `valueOf()` возвращает непреобразованный объект, и тогда интерпретатор JavaScript пытается преобразовать объект в строку вызовом метода `toString()`.

---

[1] Этот раздел содержит достаточно сложный материал, который при первом прочтении можно пропустить.

Однако здесь есть одно исключение из правил: когда с оператором + используется объект Date, преобразование сразу начинается с вызова метода toString(). Это исключение обусловлено тем обстоятельством, что объект Date обладает собственными реализациями методов toString() и valueOf(). Однако когда объект Date указывается с оператором +, чаще всего подразумевается операция конкатенации строк, а при выполнении операции сравнения практически всегда требуется определить, какая из двух дат является более ранней по времени.

Большинство объектов либо вообще не имеют метода valueOf(), либо этот метод не возвращает значение требуемого элементарного типа. Когда объект используется с оператором +, обычно выполняется операция конкатенации строк вместо сложения чисел. Аналогично, когда объект участвует в операциях сравнения, обычно производится сравнение строковых значений, а не чисел.

Объекты, обладающие собственной реализацией метода valueOf(), могут вести себя иначе. Если вы переопределите метод valueOf(), чтобы он возвращал число, над объектом можно будет выполнять арифметические и другие числовые операции, но операция сложения объекта со строкой может не дать желаемого результата, поскольку метод valueOf() возвратит значение элементарного типа, и метод toString() вызван уже не будет. В результате к строке будет добавляться строковое представление числа, возвращаемого методом valueOf().

Наконец, следует запомнить, что метод valueOf() не вызывает метод toNumber(). Строго говоря, назначение этого метода состоит в том, чтобы преобразовать объект в разумное значение элементарного типа; по этой причине в некоторых объектах методы valueOf() возвращают строку.

3.15. По значению или по ссылке

В JavaScript, как и в других языках программирования, имеется возможность манипулировать данными тремя способами.[1] Первый способ – это копирование данных. Например, значение можно присвоить новой переменной. Второй способ – передать значение функции или методу. Третий – сравнить его с другим значением, чтобы проверить, равны ли эти значения. Для понимания языка программирования совершенно необходимо разобраться, как в нем выполняются эти три действия.

Существует два фундаментальных способа манипулирования данными: *по значению* и *по ссылке*. Когда выполняется манипулирование данной величиной по значению, это означает, что в операции участвует собственно *значение* данной величины. В операции присваивания создается копия фактического значения, после чего эта копия сохраняется в переменной, в свойстве объекта или в элементе массива. Копия и оригинал – это два совершенно независимых друг от друга значения, которые хранятся раздельно. Когда некоторая величина передается функции по значению, это означает, что функции передается копия. Если функция изменит полученное значение, эти изменения коснутся только копии и никак не затронут оригинал. Наконец, когда величина сравнивается по значению с другой величиной, два различных набора данных должны содержать одно

[1] Этот раздел содержит достаточно сложный материал, который при первом прочтении можно пропустить.

и то же значение (это обычно подразумевает, что для выявления эквивалентности величин производится их побайтное сравнение).

Другой способ манипулирования значением – по ссылке. В этом случае существует только одна фактическая копия значения, а манипулирование производится посредством ссылок на это значение.[1] Когда действия со значением производятся по ссылке, переменные хранят не само значение, а лишь ссылки на него. Именно эта ссылочная информация копируется, передается и участвует в операциях сравнения. Таким образом, в операции присваивания по ссылке участвует сама ссылка, а не копия значения и не само значение. После присваивания переменная будет ссылаться на то же самое значение, что и оригинальная переменная. Обе ссылки считаются абсолютно равноправными и в равной степени могут использоваться для манипулирования значением. Если значение изменяется с помощью одной ссылки, эти изменения будут наблюдаться с помощью другой ссылки. То же происходит, когда значение передается функции по ссылке. В функцию попадает ссылка на значение, а функция может использовать ее для изменения самого значения. Любые такие изменения становятся видимыми за пределами функции. Наконец, когда выполняется операция сравнения по ссылке, происходит сравнение двух ссылок, чтобы проверить, не ссылаются ли они на одно и то же значение. Ссылки на два разных значения, даже эквивалентные (т. е. состоящие из тех же самых байтов данных), не могут считаться равными.

Это два абсолютно разных способа манипулирования значениями, и разобраться в них совершенно необходимо. В табл. 3.4 приводится краткое описание вышеизложенного. Данная дискуссия, посвященная манипулированию данными по ссылке и по значению, носила достаточно общий характер, но она с небольшими отличиями вполне применима ко всем языкам программирования. В следующих разделах описываются характерные отличия, свойственные языку JavaScript. Там рассказывается о том, какими типами данных манипулировать по значению, а какими – по ссылке.

Таблица 3.4. Передача данных по ссылке и по значению

	По значению	**По ссылке**
Копирование	Выполняется копирование самого значения – образуются две независимые друг от друга копии.	Копируется только ссылка на значение. Если значение будет изменено с помощью вновь созданной копии ссылки, эти изменения будут наблюдаться при использовании оригинальной ссылки.
Передача	Функции передается отдельная копия значения. Изменение этой копии не оказывает никакого влияния на значение за пределами функции.	Функции передается ссылка на значение. Если внутри функции значение будет изменено с помощью полученной ссылки, эти изменения будут наблюдаться и за ее пределами.

[1] Программисты на C и все те, кому знакома концепция указателей, должны понимать основную идею ссылок в данном контексте. Тем не менее следует отметить, что JavaScript не поддерживает работу с указателями.

	По значению	По ссылке
Сравнение	Сравниваются два разных значения (часто побайтно), чтобы определить, равны ли они.	Сравниваются две ссылки, чтобы определить, ссылаются ли они на одно и то же значение. Ссылки на разные значения рассматриваются как неравные, даже если сами значения совершенно идентичны.

3.15.1. Элементарные и ссылочные типы

Главное правило JavaScript заключается в следующем: операции над *элементарными типами* производятся по значению, а над *ссылочными типами*, как и следует из их названия, – по ссылке. Числа и логические величины – это элементарные типы в JavaScript. Элементарные потому, что состоят из небольшого и фиксированного числа байтов, операции над которыми с легкостью выполняются низкоуровневым интерпретатором JavaScript. Представителями ссылочных типов являются объекты. Массивы и функции – это специализированные типы объектов и потому также являются ссылочными типами. Эти типы данных могут состоять из произвольного числа свойств или элементов, поэтому оперировать ими не так просто, как значениями элементарных типов, имеющими фиксированные размеры. Поскольку размеры массивов и объектов могут быть чрезвычайно велики, операции по значению над ними могут привести к неоправданному копированию и сравнению гигантских объемов памяти.

А что можно сказать о строках? Строки могут иметь произвольную длину, поэтому их вполне можно было бы рассматривать как ссылочный тип. Тем не менее в JavaScript строки обычно рассматриваются как элементарный тип просто потому, что они не являются объектами. В действительности строки не вписываются в двухполярный элементарно-ссылочный мир. Подробнее я остановлюсь на строках чуть позже.

Лучший способ выяснить различия между данными, операции над которыми производятся по ссылке и по значению, состоит в изучении практического примера. Тщательно разберитесь в следующем примере, обращая особое внимание на комментарии. В примере 3.1 выполняется копирование, передача и сравнение чисел. Поскольку числа являются элементарными типами, данный пример является иллюстрацией операций, выполняемых по значению.

Пример 3.1. Копирование, передача и сравнение величин по значению

```
// Первой рассматривается операция копирования по значению
var n = 1;    // Переменная n хранит значение 1
var m = n;    // Копирование по значению: переменная m хранит другое значение 1

// Данная функция используется для иллюстрации операции передачи величины по значению
// Как вы увидите, функция работает не совсем так, как хотелось бы
function add_to_total(total, x)
{
    total = total + x; // Эта строка изменяет лишь внутреннюю копию total
}

// Теперь производится обращение к функции, которой передаются по значению числа,
// содержащиеся в переменных n и m. Копия значения из переменной n внутри функции
// доступна под именем total. Функция складывает копии значений переменных m и n,
```

```
// записывая результат в копию значения переменной n. Однако это не оказывает
// никакого влияния на оригинальное значение переменной n за пределами функции.
// Таким образом, в результате вызова этой функции мы не получаем никаких изменений.
add_to_total(n, m);

// Сейчас мы проверим операцию сравнения по значению.
// В следующей строке программы литерал 1 представляет собой совершенно
// независимое числовое значение, "зашитое" в текст программы. Мы сравниваем
// его со значением, хранящимся в переменной n. В данном случае, чтобы
// убедиться в равенстве двух чисел, выполняется их побайтовое сравнение.
if (n == 1) m = 2; // n содержит то же значение, что и литерал 1;
                   // таким образом в переменную m будет записано значение 2
```

Теперь рассмотрим пример 3.2. В этом примере операции копирования, передачи и сравнения выполняются над объектами. Поскольку объекты относятся к ссылочным типам, все действия над ними производятся по ссылке. В данном примере использован объект `Date`, подробнее о котором можно прочитать в третьей части книги.

Пример 3.2. Копирование, передача и сравнение величин по ссылке

```
// Здесь создается объект, который соответствует дате Рождества в 2007 году
// Переменная xmas хранит ссылку на объект, а не сам объект
var xmas = new Date(2007, 11, 25);

// Затем выполняется копирование ссылки, получается вторая ссылка на оригинальный объект
var solstice = xmas; // Обе переменные ссылаются на тот же самый объект

// Здесь выполняется изменение объекта с помощью новой ссылки
solstice.setDate(21);

// Изменения можно наблюдать при использовании первой ссылки
xmas.getDate( ); // Возвращает 21, а не первоначальное значение 25

// То же происходит при передаче объектов и массивов в функции.
// Следующая функция складывает значения всех элементов массива.
// Функции передается ссылка на массив, а не копия массива.
// Благодаря этому функция может изменять содержимое массива, переданного
// по ссылке, и эти изменения будут видны после возврата из функции.
function add_to_totals(totals, x)
{
    totals[0] = totals[0] + x;
    totals[1] = totals[1] + x;
    totals[2] = totals[2] + x;
}

// Наконец, далее выполняется операция сравнения по ссылке.
// При сравнении двух переменных, созданных ранее, обнаруживается,
// что они эквивалентны, поскольку ссылаются на один и тот же объект даже
// при том, что производилось изменение даты по одной из них:
(xmas == solstice) // Возвращает значение true

// Две переменные, созданные позднее, ссылаются на разные объекты,
// каждый из которых содержит одну и ту же дату.
var xmas = new Date(2007, 11, 25);
var solstice_plus_4 = new Date(2007, 11, 25);

// Но согласно правилу "сравнения по ссылке" ссылки на разные объекты
```

```
// не считаются эквивалентными!
(xmas != solstice_plus_4) // Возвращает значение true
```

Прежде чем закончить обсуждение темы выполнения операций над объектами и массивами по ссылке, добавим немного ясности. Фраза «передача по ссылке» может иметь несколько смыслов. Для некоторых из вас эта фраза означает такой способ вызова функции, который позволяет изменять эти значения внутри функции и наблюдать эти изменения за ее пределами. Однако данный термин трактуется в этой книге в несколько ином смысле. Здесь просто подразумевается, что функции передается ссылка на массив или объект, но не сам объект. Функция же с помощью этой ссылки получает возможность изменять свойства объекта или элементы массива, и эти изменения сохраняются по выходе из функции. Те из вас, кто знаком с другими трактовками этого термина, могут заявить, что объекты и массивы передаются по значению, правда, этим значением фактически является ссылка на объект, а не сам объект. Пример 3.3 наглядно иллюстрирует эту проблему.

Пример 3.3. Ссылки передаются по значению

```
// Здесь приводится другая версия функции add_to_totals(). Хотя она не работает,
// потому что вместо изменения самого массива она изменяет ссылку на этот массив.
function add_to_totals2(totals, x)
{
    newtotals = new Array(3);
    newtotals[0] = totals[0] + x;
    newtotals[1] = totals[1] + x;
    newtotals[2] = totals[2] + x;
    totals = newtotals; // Эта строка не оказывает влияния
                        // на массив за пределами функции
}
```

3.15.2. Копирование и передача строк

Как упоминалось ранее, строки не вписываются в двухполярный элементарно-ссылочный мир. Поскольку строки не являются объектами, вполне естественно предположить, что они относятся к элементарному типу. Если строки рассматривать как элементарный тип данных, тогда в соответствии с описанными выше правилами операции над ними должны производиться по значению. Но поскольку строки могут иметь произвольную длину, это может приводить к непроизводительному расходованию системных ресурсов на операции копирования и побайтового сравнения. Таким образом, не менее естественно было бы предположить, что строки реализованы как ссылочный тип данных.

Вместо того чтобы строить предположения, можно попробовать написать небольшой фрагмент на языке JavaScript и решить проблему экспериментальным путем. Если строки копируются и передаются по ссылке, должна иметься возможность изменять их содержимое с помощью ссылки, хранящейся в другой переменной, или в результате передачи строки в функцию.

Однако при попытке написать такой фрагмент для проведения эксперимента вы столкнетесь с серьезной проблемой: в JavaScript невозможно изменить содержимое строки. Существует метод `charAt()`, который возвращает символ из заданной позиции в строке, но нет соответствующего ему метода `setCharAt()`, позволяющего ввести в эту позицию другой символ. Это не упущение. Строки в JavaScript

преднамеренно *неизменяемы* – в JavaScript отсутствуют элементы языка, с помощью которых можно было бы изменять символы в строке.

Поскольку строки являются неизменяемыми, вопрос остается открытым, т. к. нет никакого способа проверить, как передаются строки – по ссылке или по значению. Можно предположить, что с целью повышения эффективности интерпретатор JavaScript реализован так, чтобы строки передавались по ссылке, но это так и останется всего лишь предположением, поскольку нет возможности проверить его экспериментально.

3.15.3. Сравнение строк

Несмотря на отсутствие возможности определить, как копируются строки, по ссылке или по значению, существует возможность написать такой фрагмент на JavaScript, с помощью которого можно выяснить, как именно выполняется операция сравнения – по ссылке или по значению. В примере 3.4 приводится фрагмент, выполняющий такую проверку.

Пример 3.4. Как сравниваются строки, по ссылке или по значению?

```
// Определить, как сравниваются строки, по ссылке или по значению,
// довольно просто. Здесь сравниваются совершенно разные строки, содержащие
// одинаковые последовательности символов. Если сравнение выполняется
// по значению, они должны интерпретироваться как эквивалентные, если же они
// сравниваются по ссылке, результат должен быть противоположным:
var s1 = "hello";
var s2 = "hell" + "o";
if (s1 == s2) document.write("Строки сравниваются по значению");
```

Данный эксперимент доказывает, что строки сравниваются по значению. Это может оказаться сюрпризом для некоторых программистов, работающих с языками C, C++ и Java, где строки относятся к ссылочным типам и сравниваются по ссылке. При необходимости сравнить в этих языках фактическое содержимое строк приходится использовать специальные методы или функции. Язык JavaScript относится к языкам высокого уровня и потому предполагает, что когда выполняется сравнение строк, скорее всего, имеется в виду сравнение по значению. Таким образом, несмотря на то, что с целью достижения более высокой эффективности строки в JavaScript (по-видимому) копируются и передаются по ссылке, тем не менее операция сравнения выполняется по значению.

3.15.4. По ссылке или по значению: подведение итогов

Таблица 3.5 кратко иллюстрирует то, как выполняются операции над различными типами данных в JavaScript.

Таблица 3.5. Операции над типами данных в JavaScript

Тип	Копирование	Передача	Сравнение
Число	По значению	По значению	По значению
Логическое значение	По значению	По значению	По значению
Строка	Не изменяется	Не изменяется	По значению
Объект	По ссылке	По ссылке	По ссылке

4

Переменные

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

```
i = 2;
```

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

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

Это почти все, что надо знать о переменных. Но для полного понимания механизма их работы в JavaScript следует освоить и другие понятия, и пары строк кода здесь недостаточно! В данной главе рассматриваются вопросы типизации, объявления, области видимости, содержимого и разрешения имен переменных, а также сборки мусора и двойственности понятия «переменная/свойство».[1]

4.1. Типизация переменных

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

```
i = 10;
i = "десять";
```

---

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

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

Особенностью JavaScript, вытекающей из отсутствия типизации, является то, что язык в случае необходимости легко и автоматически преобразует значения из одного типа в другой. Например, если вы попытаетесь дописать число к строке, JavaScript автоматически преобразует число в соответствующую строку, которая может быть добавлена к имеющейся. Более подробно преобразования типов рассматриваются в главе 3.

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

4.2. Объявление переменных

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

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

Можно объявить несколько переменных:

```
var i, sum;
```

Кроме того, объявление переменной можно совмещать с ее инициализацией:

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

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

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

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

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

---

[1] Если этого не сделать, то переменная неявно будет объявлена самим интерпретатором JavaScript.

4.2.1. Повторные и опущенные объявления

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

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

4.3. Область видимости переменной

Область видимости (*scope*) переменной – это та часть программы, для которой эта переменная определена. *Глобальная* переменная имеет глобальную область видимости – она определена для всей JavaScript-программы. Переменные, объявленные внутри функции, определены только в ее теле. Они называются *локальными* и имеют локальную область видимости. Параметры функций также считаются локальными переменными, определенными только в теле этой функции.

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

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

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

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

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

Определения функций могут быть вложенными. Каждая функция имеет собственную локальную область видимости, поэтому может быть несколько вложенных уровней локальных областей видимости. Например:

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

4.3.1. Отсутствие блочной области видимости

Обратите внимание: в отличие от C, C++ и Java, в JavaScript нет области видимости на уровне блоков. Все переменные, объявленные внутри функции, независимо от того, где именно это сделано, определены во *всей* функции. В следующем фрагменте переменные `i`, `j` и `k` имеют одинаковые области видимости: все три определены во всем теле функции. Это было бы не так, если бы код был написан на C, C++ или Java:

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

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

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

Кто-то может подумать, что в результате первого вызова `alert()` будет напечатано слово «глобальная», т. к. инструкция `var`, объявляющая локальную перемен-

ную, еще не была выполнена. Однако согласно правилу определения областей видимости все происходит не так. Локальная переменная определена во всем теле функции, значит, глобальная переменная с тем же именем скрыта во всем теле функции. Хотя локальная переменная определена везде, до выполнения инструкции `var` она не инициализирована. Поэтому функция `f` в предыдущем примере эквивалентна следующему фрагменту:

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

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

4.3.2. Неопределенные и неинициализированные переменные

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

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

В следующем фрагменте иллюстрируются некоторые различия между неопределенными и неинициализированными переменными:

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

4.4. Элементарные и ссылочные типы

Следующая тема, которую мы рассмотрим, – содержимое переменных. Мы часто говорим, что переменные содержат значения. Что же они содержат в действительности? Чтобы ответить на этот, казалось бы, простой вопрос, мы должны снова взглянуть на типы данных, поддерживаемые JavaScript. Эти типы можно разделить на две группы: элементарные и ссылочные.

Числа, логические значения, а также значения `null` и `undefined` – это элементарные типы. Объекты, массивы и функции – это ссылочные типы.

Элементарный тип имеет фиксированный размер. Например, число занимает восемь байтов, а логическое значение может быть представлено всего одним битом.

Числовой тип – самый большой из элементарных типов. Если для каждой JavaScript-переменной зарезервировано в памяти восемь байтов, переменная может непосредственно содержать значение любого элементарного типа.[1]

Однако ссылочные типы – это другое дело. Например, объекты могут быть любой длины – они не имеют фиксированного размера. То же самое относится и к массивам: массив может иметь любое число элементов. Аналогично функция может содержать любой объем JavaScript-кода. Поскольку данные типы не имеют фиксированного размера, их значения не могут непосредственно храниться в восьми байтах памяти, связанных с каждой переменной. Поэтому в переменной хранится *ссылка* на это значение. Обычно эта ссылка представляет собой какой-либо указатель или адрес в памяти. Ссылка – это не само значение, но она сообщает переменной, где это значение можно найти.

Различие между элементарными и ссылочными типами существенно, т. к. они ведут себя по-разному. Рассмотрим следующий код, оперирующий числами (элементарный тип):

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

В этом фрагменте нет ничего необычного. Теперь посмотрим, что произойдет, если слегка изменить код, заменив числа массивами (ссылочный тип):

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

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

Если разница между элементарным и ссылочным типами вам внове, просто постарайтесь держать в уме содержимое переменной. Переменные содержат фактические значения элементарных типов, но лишь ссылки на значения ссылочных типов. Разное поведение базовых и ссылочных типов более подробно изучается в разделе 3.15.

Вы могли заметить, что я не указал, относятся ли строки в JavaScript к базовым или к ссылочным типам. Строки – это необычный случай. Они имеют переменную длину и потому, очевидно, не могут храниться непосредственно в переменных фиксированного размера. Исходя из соображений эффективности можно ожидать, что интерпретатор JavaScript будет копировать ссылки на строки, а не их фактическое содержимое. В то же время во многих отношениях строки ведут себя как элементарные типы. Вопрос о том, к какому типу принадлежат строки, элементарному

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

или ссылочному, спорный, т. к. строки на самом деле *неизменны*: нет возможности избирательно изменить содержимое внутри строкового значения. Это значит, что нельзя составить пример, похожий на предыдущий, в котором массивы копировались бы по ссылке. В конце концов, не имеет значения, как рассматривать строки, как неизменный ссылочный тип, ведущий себя как элементарный, или как элементарный тип, реализованный с использованием механизма ссылочного типа.

4.5. Сборка мусора

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

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

В таких языках, как C и C++, память приходится освобождать вручную. Именно программист отвечает за отслеживание всех создаваемых объектов и, когда они больше не требуются, – за их ликвидацию (освобождение памяти). Это бывает довольно обременительно и часто приводит к ошибкам.[1]

В JavaScript, где не надо вручную освобождать память, реализована технология, называемая *сборкой мусора* (*garbage collection*). Интерпретатор JavaScript может обнаружить, что объект никогда более не будет использоваться программой. Определив, что объект недоступен (т. е. больше нет способа получения ссылки на него), интерпретатор выясняет, что объект более не нужен, и занятая им память может быть освобождена.[2] Рассмотрим следующие строки кода:

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

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

---

[1] Это не совсем строго: для локальных (объявленных в функции) переменных, размещаемых в стеке, какой бы сложной структуры ни были переменные, автоматически вызывается деструктор и производится освобождение памяти. Точно так же ведут себя контейнеры STL, или «собственные данные потока». Утверждение автора в абсолютной степени относится только к объектам, динамически распределяемым операторами `new` и `delete`. – *Примеч. науч. ред.*

[2] Описываемая схема сборки мусора, известная как «подсчет числа ссылок», может иметь серьезные проблемы в более сложных программах при появлении объектов с циклическими ссылками – объекты никогда не будут освобождены. Эта проблема хорошо изучена в Perl; способы борьбы см. в описании языка. – *Примеч. науч. ред.*

Сборка мусора выполняется автоматически и невидима для программиста. О сборке мусора он должен знать ровно столько, сколько ему требуется, чтобы доверять ее работе, – он не должен думать, куда делись все старые объекты.

4.6. Переменные как свойства

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

4.6.1. Глобальный объект

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

Интерпретатор JavaScript инициализирует ряд свойств глобального объекта, ссылающихся на предопределенные значения и функции. Так, свойства `Infinity`, `parseInt` и `Math` ссылаются на число «бесконечность», предопределенную функцию `parseInt()` и предопределенный объект `Math`. Более подробно прочитать об этих глобальных значениях можно в третьей части книги.

В коде верхнего уровня (т. е. JavaScript-коде, который не является частью функции) сослаться на глобальный объект можно посредством ключевого слова `this`. Внутри функций ключевое слово `this` имеет другое применение, которое описано в главе 8.

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

4.6.2. Локальные переменные – объект вызова

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

4.6.3. Контексты исполнения в JavaScript

Начиная исполнять функцию, интерпретатор JavaScript создает для нее новый *контекст исполнения* (*execution context*), т. е. контекст, в котором выполняется

любой фрагмент JavaScript-кода. Важная часть контекста – объект, в котором определены переменные. Поэтому код JavaScript-программы, не являющийся частью какой-либо функции, работает в контексте исполнения, в котором для определений переменных используется глобальный объект. А любая JavaScript-функция работает в собственном уникальном контексте исполнения с собственным объектом вызова, в котором определены локальные переменные.

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

Вам не обязательно уже сейчас полностью понимать, как связываются вместе контексты исполнения отдельных окон и фреймов в клиентском JavaScript. Эту тему мы подробно рассмотрим при обсуждении интеграции JavaScript с веб-броузерами в главе 13. Сейчас достаточно знать, что гибкость JavaScript позволяет одному интерпретатору JavaScript исполнять сценарии в различных глобальных контекстах исполнения и что этим контекстам не нужно быть совершенно раздельными – они могут ссылаться друг на друга.

Последнее утверждение надо рассмотреть подробнее. Если JavaScript-код в одном контексте исполнения может читать и писать значения свойств и выполнять функции, определенные в другом контексте исполнения, то становятся актуальными вопросы безопасности. Возьмем в качестве примера клиентский JavaScript. Предположим, что окно броузера A запускает сценарий или содержит информацию из вашей локальной сети, а окно B запускает сценарий с некоторого произвольного сайта в Интернете. Скорее всего, мы не захотим предоставлять коду в окне B доступ к свойствам в окне A. Ведь тогда этот код получит возможность прочитать важную корпоративную информацию и, например, украсть ее. Следовательно, безопасный запуск JavaScript-кода должен обеспечить специальный механизм, предотвращающий доступ из одного контекста исполнения в другой, если такой доступ не разрешен. Мы вернемся к этой теме в разделе 13.8.

4.7. Еще об области видимости переменных

Когда мы в первый раз обсуждали понятие области видимости переменной, я определил его только на основе лексической структуры JavaScript-кода: глобальные переменные имеют глобальную область видимости, а переменные, объявленные в функции, – локальную. Если одно определение функции вложено

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

в другое, то переменные, объявленные в этой вложенной функции, имеют вложенную локальную область видимости. Теперь, когда мы знаем, что глобальные переменные представляют собой свойства глобального объекта, а локальные – свойства особого объекта вызова, мы можем вернуться к понятию области видимости переменной и переосмыслить его. Это даст нам хорошую возможность по-новому взглянуть на существование переменных во многих контекстах и глубже понять, как работает JavaScript.

В JavaScript с каждым контекстом исполнения связана *цепочка областей видимости* (*scope chain*), представляющая собой список, или цепочку, объектов. Когда JavaScript-коду требуется найти значение переменной `x` (этот процесс называется разрешением имени переменной), он начинает поиск в первом (наиболее глубоком) объекте цепочки. Если в этом объекте отыскивается свойство с именем `x`, то используется значение этого свойства. Если в первом объекте не удается найти свойство с именем `x`, то JavaScript продолжает поиск в следующем объекте цепочки. Если во втором объекте тоже не найдено свойство с именем `x`, поиск продолжается в следующем объекте, и т. д.

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

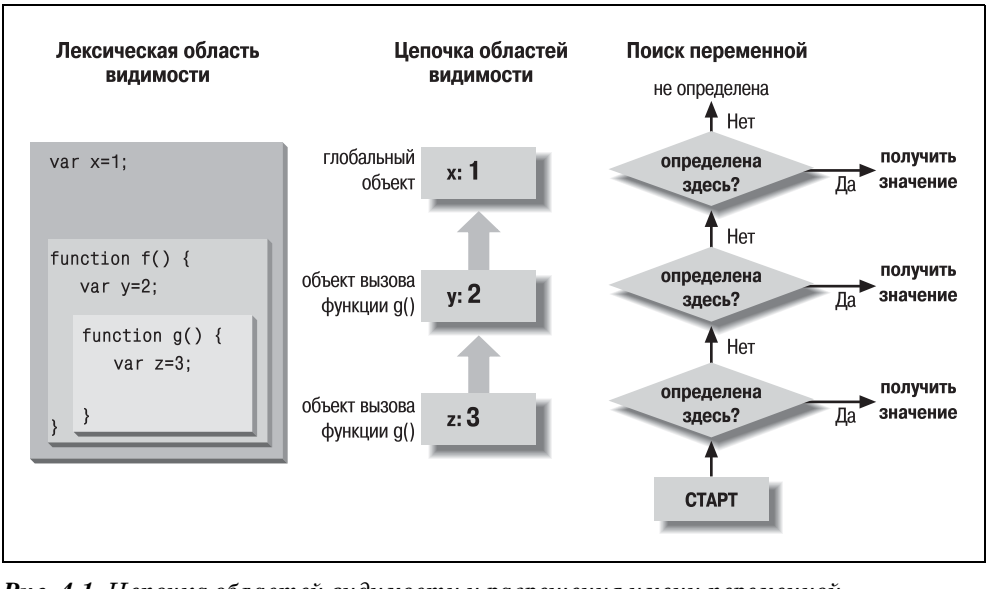

Рис. 4.1. *Цепочка областей видимости и разрешения имени переменной*

5

Выражения и операторы

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

5.1. Выражения

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

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

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

Эти выражения не очень интересны. Путем объединения простых выражений могут создаваться более сложные (и интересные) выражения. Например, мы видели, что и `1.7` – это выражение, и `i` – это выражение. Следующий пример тоже представляет собой выражение:

```
i + 1.7
```

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

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

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

5.2. Обзор операторов

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

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

Таблица 5.1. JavaScript-операторы

P	A	Оператор	Типы операндов	Выполняемая операция
15	L	`.`	Объект, идентификатор	Обращение к свойству
	L	`[]`	Массив, целое число	Индексация массива
	L	`()`	Функция, аргументы	Вызов функции
	R	`new`	Вызов конструктора	Создание нового объекта
14	R	`++`	Левостороннее выражение	Префиксный или постфиксный инкремент (унарный)
	R	`--`	Левостороннее выражение	Префиксный или постфиксный декремент (унарный)
	R	`-`	Число	Унарный минус (смена знака)
	R	`+`	Число	Унарный плюс (нет операции)
	R	`~`	Целое число	Поразрядное дополнение (унарный)
	R	`!`	Логическое значение	Логическое дополнение (унарный)
	R	`delete`	Левостороннее значение	Аннулирование определения свойства (унарный)
	R	`typeof`	Любой	Возвращает тип данных (унарный)

P	A	Оператор	Типы операндов	Выполняемая операция
	R	`void`	Любой	Возвращает неопределенное значение (унарный)
13	L	`*`, `/`, `%`	Числа	Умножение, деление, остаток
12	L	`+`, `-`	Числа	Сложение, вычитание
	L	`+`	Строки	Конкатенация строк
11	L	`<<`	Целые числа	Сдвиг влево
	L	`>>`	Целые числа	Сдвиг вправо с расширением знакового разряда
	L	`>>>`	Целые числа	Сдвиг вправо с дополнением нулями
10	L	`<`, `<=`	Числа или строки	Меньше чем, меньше или равно
	L	`>`, `>=`	Числа или строки	Больше чем, больше или равно
	L	`instanceof`	Объект, конструктор	Проверка типа объекта
	L	`in`	Строка, объект	Проверка наличия свойства
9	L	`==`	Любой	Проверка на равенство
	L	`!=`	Любой	Проверка на неравенство
	L	`===`	Любой	Проверка на идентичность
	L	`!==`	Любой	Проверка на неидентичность
8	L	`&`	Целые числа	Поразрядная операция И
7	L	`^`	Целые числа	Поразрядная операция исключающего ИЛИ
6	L	`|`	Целые числа	Поразрядная операция ИЛИ
5	L	`&&`	Логические значения	Логическое И
4	L	`||`	Логические значения	Логическое ИЛИ
3	R	`?:`	Логическое значение, любое, любое	Условный трехместный оператор
2	R	`=`	Левостороннее значение, любое	Присваивание
	R	`*=`, `/=`, `%=`, `+=`, `-=`, `<<=`, `>>=`, `>>>=`, `&=`, `^=`, `|=`	Левостороннее значение, любое	Присваивание с операцией
1	L	`,`	Любой	Множественное вычисление

5.2.1. Количество операндов

Операторы могут быть разбиты на категории по количеству требуемых им операндов. Большинство JavaScript-операторов, таких как оператор `+`, о котором мы уже говорили, *двухместные*. Такие операторы объединяют два выражения в одно, более сложное. Таким образом, эти операторы работают с двумя операндами. JavaScript поддерживает также несколько унарных операторов, которые

преобразуют одно выражение в другое, более сложное. Оператор «минус» в выражении -3 представляет собой *унарный оператор*, выполняющий смену знака для операнда 3. И наконец, JavaScript поддерживает один *тернарный оператор*, условный оператор ?:, который объединяет в одно значение три выражения.

5.2.2. Тип операндов

Создавая JavaScript-выражения, необходимо обращать внимание на типы данных, передаваемых операторам, и на типы данных, которые они возвращают. Различные операторы требуют, чтобы операнды возвращали значения определенного типа. Например, нельзя выполнить умножение строк, поэтому выражение "a" * "b" не является допустимым в JavaScript. Однако интерпретатор JavaScript по мере возможности будет пытаться преобразовывать выражение в требуемый тип, поэтому выражение "3" * "5" вполне допустимо. Его значением будет число 15, а не строка "15". Более подробно о преобразованиях типов в JavaScript рассказывалось в разделе 3.12.

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

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

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

5.2.3. Приоритет операторов

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

Рассмотрим следующее выражение:

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

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

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

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

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

5.2.4. Ассоциативность операторов

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

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

Теперь обратите внимание на такие (практически бессмысленные) выражения:

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

Они эквивалентны следующим выражениям:

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

Причина в том, что унарные операторы, операторы присваивания и условные тернарные операторы имеют ассоциативность справа налево.

5.3. Арифметические операторы

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

Сложение (+)

Оператор «плюс» складывает числовые операнды или выполняет конкатенацию строк. Если одним из операндов является строка, другой операнд преобразуется в строку и выполняется конкатенация. Операнды-объекты преобразу-

ются в числа или строки, которые могут быть сложены или конкатенированы. Преобразование выполняется с помощью методов `valueOf()` и/или `toString()`.

Вычитание (-)

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

Умножение (*)

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

Деление (/)

Оператор / делит первый операнд на второй. Нечисловые операнды он пытается преобразовать в числа. Те, кто привык к языкам программирования, различающим целые и вещественные числа, могут ожидать получения целочисленного результата при делении одного целого на другое. Однако в JavaScript все числа вещественные, поэтому результатом любого деления является значение с плавающей точкой. Операция `5/2` дает `2.5`, а не `2`. Результат деления на ноль – плюс или минус бесконечность, а `0/0` дает `NaN`.

Деление по модулю (%)

Оператор % вычисляет остаток, получаемый при целочисленном делении первого операнда на второй. Если заданы нечисловые операнды, то оператор пытается преобразовать их в числа. Знак результата совпадает со знаком первого операнда, например `5 % 2` дает 1. Оператор деления по модулю обычно применяется к целым операндам, но работает и для вещественных значений. Например, `-4.3 % 2.1` дает результат `-0.1`.

Унарный минус (-)

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

Унарный плюс (+)

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

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

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

Инкремент (++)

Этот оператор инкрементирует (т. е. увеличивает на единицу) свой единственный операнд, который должен быть переменной, элементом массива или свойством объекта. Если значение этой переменной, элемента массива или свойства не является числом, оператор сначала пытается преобразовать его

в число. Точное поведение этого оператора зависит от его положения по отношению к операнду. Если поставить его перед операндом (префиксный оператор инкремента), то к операнду прибавляется 1, а результатом является увеличенное значение операнда. Если же он размещается после операнда (постфиксный оператор инкремента), то к операнду прибавляется 1, однако результатом является *первоначальное значение* операнда. Если увеличиваемое значение не является числом, оно в процессе вычисления преобразуется в число. Например, следующий код делает переменные `i` и `j` равными 2:

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

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

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

Данный оператор в обеих своих формах чаще всего применяется для увеличения счетчика, управляющего циклом. Обратите внимание: нельзя вставлять перевод строки между префиксным или постфиксным оператором инкремента и его операндом, поскольку точки с запятой в JavaScript вставляются автоматически. Если это сделать, интерпретатор JavaScript будет рассматривать операнд как полноценную инструкцию и вставит после него точку с запятой.

Декремент (`--`)

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

5.4. Операторы равенства

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

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

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

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

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

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

При определении идентичности двух значений оператор === руководствуется следующими правилами:

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

- Если оба значения равны `null` или `undefined`, то они идентичны.

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

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

В качестве примера проверки на равенство рассмотрим сравнение:

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

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

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

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

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

5.5. Операторы отношения

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

5.5.1. Операторы сравнения

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

Меньше (`<`)

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

Больше (`>`)

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

Меньше или равно (`<=`)

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

Больше или равно (`>=`)

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

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

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

- Если один из операндов равен или преобразуется в NaN, то результатом оператора сравнения является false.

Имейте в виду, что сравнение строк выполняется строго посимвольно, для числовых значений каждого символа из кодировки Unicode. В некоторых случаях стандарт Unicode допускает кодирование эквивалентных строк с применением различных последовательностей символов, но операторы сравнения в JavaScript не обнаруживают этих различий в кодировках; предполагается, что все строки представлены в нормализованной форме. Обратите внимание: сравнение строк производится с учетом регистра символов, т. е. в кодировке Unicode (по крайней мере, для подмножества ASCII) все прописные буквы «меньше» всех строчных букв. Это правило может приводить к непонятным результатам. Например, согласно оператору < строка "Zoo" меньше строки "aardvark".

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

Операторы <= (меньше или равно) и >= (больше или равно) определяют «равенство» двух значений не при помощи операторов равенства или идентичности. Оператор «меньше или равно» определяется просто как «не больше», а оператор «больше или равно» – как «не меньше». Единственное исключение имеет место, когда один из операндов представляет собой значение NaN (или преобразуется в него); в этом случае все четыре оператора сравнения возвращают false.

5.5.2. Оператор in

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

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

5.5.3. Оператор instanceof

Оператор instanceof требует, чтобы левым операндом был объект, а правым – имя класса объектов. Результатом оператора будет true, если объект, указанный слева, представляет собой экземпляр класса, указанного справа; в противном случае результатом будет false. В главе 9 мы увидим, что в JavaScript классы объектов определяются инициализировавшей их функцией-конструктором. Следовательно, правый операнд instanceof должен быть именем функции-конструктора. Обратите внимание: все объекты представляют собой экземпляры класса Object. Например:

```
var d = new Date();  // Создаем новый объект с помощью конструктора Date()
d instanceof Date;   // Равно true; объект d был создан с помощью
                     // функции Date()
```

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

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

5.6. Строковые операторы

Как уже говорилось в предыдущих разделах, существует несколько операторов, ведущих себя особым образом, когда в качестве операндов выступают строки.

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

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

Следующие инструкции дают в результате строку `"22"`:

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

Операторы `<`, `<=`, `>` и `>=` сравнивают две строки и определяют, в каком порядке они следуют друг за другом. Сравнение основано на алфавитном порядке. Как было отмечено в разделе 5.1.1, этот алфавитный порядок базируется на используемой в JavaScript кодировке Unicode. В этой кодировке все прописные буквы латинского алфавита идут раньше, чем все строчные буквы (прописные «меньше» строчных), что может приводить к неожиданным результатам.

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

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

```
1 + 2      // Сложение. Результат равен 3.
"1" + "2"  // Конкатенация. Результат равен "12".
"1" + 2    // Конкатенация; 2 преобразуется в "2". Результат равен "12".
11 < 3     // Численное сравнение. Результат равен false.
"11" < "3" // Строковое сравнение. Результат равен true.
```

```
"11" < 3   // Численное сравнение; "11" преобразуется в 11. Результат равен false.
"one" < 3  // Численное сравнение; "one" преобразуется в NaN. Результат равен false.
```

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

```
s = 1 + 2 + " слепых мышей";   // Равно "3 слепых мышей"
t = "слепых мышей: " + 1 + 2;  // Равно " слепых мышей: 12"
```

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

```
s = (1 + 2) + "слепых мышей";   // Результат первой операции - число; второй - строка
t = ("слепых мышей: " + 1) + 2; // Результаты обеих операций - строки
```

5.7. Логические операторы

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

5.7.1. Логическое И (&&)

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

Реальное поведение этого оператора несколько сложнее. Он начинает работу с вычисления левого операнда. Если получившееся значение может быть преобразовано в `false` (если левый операнд равен `null`, `0`, `""` или `undefined`), оператор возвращает значение левого выражения. В противном случае оператор вычисляет правый операнд и возвращает значение этого выражения.[1]

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

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

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

---

[1] В JavaScript 1.0 и 1.1, если в результате вычисления левого операнда получается значение `false`, оператор `&&` возвращает непреобразованное значение левого операнда.

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

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

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

5.7.2. Логическое ИЛИ (||)

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

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

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

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

В то же время иногда можно встретить конструкции, где оператор `||` используется со значениями, не являющимися логическими, и где учитывается тот факт, что оператор возвращает значение, также не являющееся логическим. Суть такого подхода основана на том, что оператор `||` выбирает первое значение из предложенных альтернатив, которое не является значением `null` (т. е. первое значение, которое преобразуется в логическое значение `true`). Далее приводится пример такой конструкции:

```
// Если переменная max_width определена, используется ее значение.
// В противном случае значение извлекается из объекта preferences.
// Если объект (или его свойство max_with) не определен, используется
// значение константы, жестко зашитой в текст программы.
var max = max_width || preferences.max_width || 500;
```

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

5.7.3. Логическое НЕ (!)

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

5.8. Поразрядные операторы

Несмотря на то, что все числа в JavaScript вещественные, поразрядные операторы требуют в качестве операндов целые числа. Они работают с такими операндами с помощью 32-разрядного целого представления, а не эквивалентного представления с плавающей точкой. Четыре из этих операторов выполняют поразрядные операции булевой алгебры, аналогично описанным ранее логическим операторам, но рассматривая каждый бит операнда как отдельное логическое значение. Три других поразрядных оператора применяются для сдвига битов влево и вправо.

Если операнды не являются целыми числами или слишком велики и не помещаются в 32-разрядное целое, поразрядные операторы просто «втискивают» операнды в 32-разрядное целое, отбрасывая дробную часть операнда и любые биты старше 32-го. Операторы сдвига требуют, чтобы значение правого операнда было целым числом от 0 до 31. После преобразования этого операнда в 32-разрядное целое вышеописанным образом они отбрасывают любые биты старше 5-го, получая число в соответствующем диапазоне.

Те, кто не знаком с двоичными числами и двоичным представлением десятичных целых чисел, могут пропустить операторы, рассматриваемые в этом разделе. Они требуются для низкоуровневых манипуляций с двоичными числами и достаточно редко применяются при программировании на JavaScript. Далее приводится список поразрядных операторов:

Поразрядное И (`&`)

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

Поразрядное ИЛИ (`|`)

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

Поразрядное исключающее ИЛИ (`^`)

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

Поразрядное НЕ (~)

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

Сдвиг влево (<<)

Оператор << сдвигает все биты в первом операнде влево на количество позиций, указанное во втором операнде, который должен быть целым числом в диапазоне от 0 до 31. Например, в операции `a << 1` первый бит в `a` становится вторым битом, второй бит становится третьим и т. д. Новым первым битом становится ноль, значение 32-го бита теряется. Сдвиг значения влево на одну позицию эквивалентен умножению на 2, на две позиции – умножению на 4, и т. д. Например, `7 << 1` равно 14.

Сдвиг вправо с сохранением знака (>>)

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

Сдвиг вправо с заполнением нулями (>>>)

Оператор >>> аналогичен оператору >> за исключением того, что при сдвиге старшие разряды заполняются нулями независимо от знака первого операнда. Например, `-1 >> 4` равно `-1`, а `-1 >>> 4` равно `268435455` (`0x0fffffff`).

5.9. Операторы присваивания

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

```
i = 0
```

В JavaScript можно не рассматривать такую строку как выражение, которое имеет результат, но это действительно выражение и формально знак = представляет собой оператор.

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

Поскольку = представляет собой оператор, его можно включать в более сложные выражения. Так, в одном выражении можно совместить операции присваивания и проверки значения:

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

При этом следует отчетливо понимать, что между операторами = и == есть разница!

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

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

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

5.9.1. Присваивание с операцией

Помимо обычного оператора присваивания (=) JavaScript поддерживает несколько других операторов-сокращений, объединяющих присваивание с некоторой другой операцией. Например, оператор += выполняет сложение и присваивание. Следующие выражения эквивалентны:

```
total += sales_tax
total = total + sales_tax
```

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

Из подобных ему операторов можно назвать -=, *=, &= и др. Все операторы присваивания с операцией перечислены в табл. 5.2.

Таблица 5.2. Операторы присваивания

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

В большинстве случаев следующие выражения эквивалентны (здесь *op* означает оператор):

```
a op= b
a = a op b
```

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

5.10. Прочие операторы

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

5.10.1. Условный оператор (?:)

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

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

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

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

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

Эта запись эквивалентна следующей конструкции if, но более компактна:

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

5.10.2. Оператор typeof

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

Результатом оператора typeof будет строка "number", "string" или "boolean", если его операндом является число, строка или логическое значение соответственно.

Для объектов, массивов и (как ни странно) значения null результатом будет строка "object". Для операндов-функций результатом будет строка "function", а для неопределенного операнда – строка "undefined".

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

Оператор typeof может применяться, например, в таких выражениях:

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

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

```
typeof(i)
```

Для всех объектных типов и типов массивов результатом оператора typeof является строка "object", поэтому он может быть полезен только для того, чтобы отличить объекты от базовых типов. Для того чтобы отличить один объектный тип от другого, следует обратиться к другим приемам, таким как использование оператора instanceof или свойства constructor (подробности вы найдете в описании свойства Object.constructor, в третьей части книги).

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

5.10.3. Оператор создания объекта (new)

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

```
new конструктор(аргументы)
```

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

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

Оператор new сначала создает новый объект с неопределенными свойствами, а затем вызывает заданную функцию-конструктор, передавая ей указанные аргументы, а также только что созданный объект в качестве значения ключевого слова this. С помощью этого слова функция-конструктор может инициализиро-

вать новый объект любым необходимым образом. В главе 7 оператор new, ключевое слово this и функции-конструкторы рассмотрены более подробно.

Оператор new может также применяться для создания массивов с помощью синтаксиса new Array(). Подробнее о создании объектов и массивов и работе с ними мы поговорим в главе 7.

5.10.4. Оператор delete

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

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

Обратите внимание: удаленное свойство, переменная или элемент массива не просто устанавливается в undefined. Когда свойство удалено, оно прекращает существование. Эта тема обсуждалась в разделе 4.3.2.

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

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

5.10.5. Оператор void

Унарный оператор void указывается перед своим единственным операндом, тип которого может быть любым. Действие этого оператора необычно: он отбрасыва-

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

ет значение операнда и возвращает undefined. Чаще всего этот оператор применяется на стороне клиента в URL-адресе с признаком псевдопротокола javascript:, где позволяет вычислять выражение ради его побочных действий, не отображая в броузере вычисленное значение.

Например, можно использовать оператор void в HTML-теге:

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

Другое применение void – это намеренная генерация значения undefined. Оператор void определяется в ECMAScript v1 и реализуется в JavaScript 1.1. В ECMAScript v3 определяется глобальное свойство undefined, реализованное в JavaScript 1.5. Однако для сохранения обратной совместимости лучше обращаться к выражению вроде void 0, а не к свойству undefined.

5.10.6. Оператор «запятая»

Оператор «запятая» (,) очень прост. Он вычисляет свой левый операнд, вычисляет свой правый операнд и возвращает значение правого операнда, т. е. следующая строка

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

возвращает значение 2 и практически эквивалентна записи:

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

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

5.10.7. Операторы доступа к массивам и объектам

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

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

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

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

Большинство операторов допускают произвольные выражения для всех своих операндов, если только тип операнда в данном случае допустим. Оператор «точ-

ка» представляет собой исключение: правый операнд должен быть идентификатором. Ничего другого не допускается.

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

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

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

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

Оператор [] обычно применяется для обращения к элементам массива. Для доступа к свойствам объекта он менее удобен, чем оператор «точка», т. к. требует заключения имени свойства в кавычки. Однако когда объект выступает в роли ассоциативного массива, а имена свойств генерируются динамически, оператор «точка» использоваться не может и следует применять оператор []. Чаще всего такая ситуация возникает в случае применения цикла for/in, рассмотренного в главе 6. Например, в следующем фрагменте для вывода имен и значений всех свойств объекта o используются цикл for/in и оператор []:

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

5.10.8. Оператор вызова функции

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

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

6

Инструкции

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

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

6.1. Инструкции-выражения

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

```
s = "Привет " + name;
i *= 3;
```

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

```
counter++;
```

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

```
delete o.x;
```

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

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

Эти вызовы клиентских функций представляют собой выражения, однако они влияют на веб-броузер, поэтому являются также и инструкциями.

Если функция не имеет каких-либо побочных эффектов, нет смысла вызывать ее, если только она не является частью инструкции присваивания. Например, никто не станет просто вычислять косинус и отбрасывать результат:

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

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

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

Опять же обратите внимание: каждая строка этих примерах завершается точкой с запятой.

6.2. Составные инструкции

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

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

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

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

Исполняя составную инструкцию, интерпретатор JavaScript просто исполняет одну за другой составляющие ее инструкции в том порядке, в котором они запи-

саны. Обычно интерпретатор исполняет все инструкции, однако в некоторых случаях выполнение составной инструкции может быть внезапно прервано. Это происходит, если в составной инструкции содержится инструкция `break`, `continue`, `return` или `throw` и если при выполнении возникает ошибка либо вызов функции приводит к ошибке или генерации необрабатываемого исключения. Об этих внезапных прерываниях работы мы узнаем больше в последующих разделах.

6.3. Инструкция if

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

```
if (выражение)
    инструкция
```

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

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

Аналогично:

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

Несмотря на кажущуюся избыточность, скобки вокруг выражения являются обязательной частью синтаксиса инструкции `if`. Как было упомянуто в предыдущем разделе, мы всегда можем заменить одиночную инструкцию блоком инструкций. Поэтому инструкция `if` может выглядеть так:

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

Отступы, присутствующие в этих примерах, не обязательны. Дополнительные пробелы и табуляции игнорируются в JavaScript, и поскольку мы ставили после каждой отдельной инструкции точку с запятой, эти примеры могли быть записаны в одну строку. Оформление текста с использованием символов перевода строки и отступов, как это показано здесь, облегчает чтение и понимание кода.

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

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

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

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

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

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

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

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

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

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

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

6.4. Инструкция else if

Мы видели, что инструкция `if/else` используется для проверки условия и выполнения одного из двух фрагментов кода в зависимости от результата провер-

ки. Но что если требуется выполнить один из многих фрагментов кода? Возможный способ сделать это состоит в применении инструкции `else if`. Формально это не JavaScript-инструкция, а лишь распространенный стиль программирования, состоящий в применении повторяющихся инструкций `if/else`:

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

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

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

6.5. Инструкция switch

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

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

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

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

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

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

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

Ниже приводится более реальный пример использования инструкции `switch`; он преобразует значение в строку способом, зависящим от типа значения:

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

```
            return x.toString().toUpperCase();
        default:        // Любой другой тип преобразуем обычным способом
            return x.toString()
    }
}
```

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

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

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

Обратите внимание: использование выражений `case`, имеющих побочные эффекты, такие как вызовы функций и присваивания, не является хорошей практикой программирования, т. к. при каждом исполнении инструкции `switch` вычисляются не все выражения `case`. Когда побочные эффекты возникают лишь в некоторых случаях, трудно понять и предсказать поведение программы. Безопаснее всего ограничиваться в выражениях `case` константными выражениями.

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

6.6. Инструкция while

Так же как инструкция `if` является базовой управляющей инструкцией, позволяющей интерпретатору JavaScript принимать решения, инструкция `while` – это

---

[1] Это существенное отличие инструкции `switch` в JavaScript от инструкции `switch` в C, C++ и Java. В этих языках выражения `case` должны быть константами, вычисляемыми на этапе компиляции, иметь тип `integer` или другой перечислимый тип, причем один и тот же тип для всех констант.

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

базовая инструкция, позволяющая JavaScript выполнять повторяющиеся действия. Она имеет следующий синтаксис:

```
while (выражение)
    инструкция
```

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

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

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

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

6.7. Цикл do/while

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

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

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

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

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

6.8. Инструкция for

Цикл, начинающийся с инструкции for, часто оказывается более удобным, чем while. Инструкция for использует шаблон, общий для большинства циклов (в том числе приведенного ранее примера цикла while). Большинство циклов имеют некоторую переменную-счетчик. Эта переменная инициализируется перед началом цикла и проверяется в выражении, вычисляемом перед каждой итерацией цикла. И наконец, переменная-счетчик инкрементируется или изменяется каким-либо другим образом в конце тела цикла, непосредственно перед повторным вычислением выражения.

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

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

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

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

Другими словами, выражение *инициализация* вычисляется один раз перед началом цикла. Это выражение, как правило, является выражением с побочными эффектами (обычно присваиванием), т. к. от него должна быть какая-то польза.

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

JavaScript также допускает, чтобы выражение *инициализация* было инструкцией объявления переменной `var`, поэтому можно одновременно объявить и проинициализировать счетчик цикла. Выражение *проверка* вычисляется перед каждой итерацией и определяет, будет ли выполняться тело цикла. Если результат проверки равен `true`, выполняется *инструкция*, являющаяся телом цикла. В конце цикла вычисляется выражение *инкремент*. И это выражение, чтобы приносить пользу, должно быть выражением с побочными эффектами. Обычно это либо выражение присваивания, либо выражение, использующее оператор `++` или `--`.

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

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

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

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

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

6.9. Инструкция for/in

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

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

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

Элементы массива можно перебирать простым увеличением индексной переменной при каждом исполнении тела цикла `while` или `for`. Инструкция `for/in` предоставляет средство перебора всех свойств объекта. Тело цикла `for/in` исполняется единожды для каждого свойства объекта. Перед исполнением тела цикла имя одного из свойств объекта присваивается переменной в виде строки. В теле цикла эту переменную можно использовать для получения значения свойства

объекта с помощью оператора []. Например, следующий цикл for/in печатает имена и значения всех свойств объекта:

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

Обратите внимание: переменной в цикле for/in может быть любое выражение, если только результатом его является нечто, подходящее для левой части присваивания. Это выражение вычисляется при каждом вызове тела цикла, т. е. каждый раз оно может быть разным. Так, скопировать имена всех свойств объекта в массив можно следующим образом:

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

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

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

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

Цикл for/in на самом деле не перебирает все свойства всех объектов. Так же как некоторые свойства объектов помечаются как доступные только для чтения или постоянные (не удаляемые), свойства могут помечаться как неперечислимые. Такие свойства не перечисляются циклом for/in. Если все свойства, определенные пользователем, перечисляются, то многие встроенные свойства, включая все встроенные методы, не перечисляются. Как мы увидим в главе 7, объекты могут наследовать свойства от других объектов. Унаследованные свойства, которые определены пользователем, также перечисляются циклом for/in.

6.10. Метки

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

идентификатор: *инструкция*

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

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

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

6.11. Инструкция break

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

```
break;
```

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

JavaScript допускает указание имени метки за ключевым словом `break`:

```
break имя_метки;
```

Обратите внимание: *имя_метки* — это просто идентификатор; за ним не указывается двоеточие, как в случае определения метки инструкции.

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

Как обсуждалось в главе 2, между ключевым словом `break` и именем метки перевод строки не допускается. Дело в том, что интерпретатор JavaScript автоматически вставляет пропущенные точки с запятой. Если разбить строку кода между ключевым словом `break` и следующей за ним меткой, интерпретатор предположит, что имелась в виду простая форма этой инструкции без метки, и добавит точку с запятой.

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

Следующий фрагмент выполняет поиск определенного значения среди элементов массива. Цикл прерывается естественным образом, когда доходит до конца

массива; если искомое значение найдено, он прерывается с помощью инструкции break:

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

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

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

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

6.12. Инструкция continue

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

```
continue;
```

Инструкция continue может также использоваться с меткой:

```
continue имя_метки;
```

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

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

- В цикле while указанное в начале цикла *выражение* проверяется снова, и если оно равно true, тело цикла выполняется сначала.
- В цикле do/while исполнение переходит в конец цикла, где перед повторным исполнением цикла снова проверяется условие.
- В цикле for вычисляется выражение инкремента и снова проверяется выражение проверки, чтобы определить, следует ли выполнять следующую итерацию.

- В цикле for/in цикл начинается заново с присвоением указанной переменной имени следующего свойства.

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

В следующем примере показано использование инструкции continue без метки для выхода из текущей итерации цикла в случае ошибки:

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

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

6.13. Инструкция var

Инструкция var позволяет явно объявить одну или несколько переменных. Инструкция имеет следующий синтаксис:

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

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

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

Инструкция var определяет каждую из перечисленных переменных путем создания свойства с этим именем в объекте вызова функции, в которой она находится, или в глобальном объекте, если объявление находится не в теле функции. Свойство или свойства, создаваемые с помощью инструкции var, не могут быть удалены оператором delete. Обратите внимание: помещение инструкции var внутрь инструкции with (см. раздел 6.18) не изменяет ее поведения.

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

Кроме того, инструкция `var` может являться частью циклов `for` и `for/in`. Например:

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

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

6.14. Инструкция function

Инструкция `function` в JavaScript определяет функцию. Она имеет следующий синтаксис:

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

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

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

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

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

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

function hypotenuse(x, y) {
    return Math.sqrt(x*x + y*y); // Инструкция return описана далее
}

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

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

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

Тот факт, что функции определяются на этапе синтаксического анализа, а не во время исполнения, приводит к некоторым интересным эффектам. Рассмотрим следующий фрагмент:

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

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

В главе 8 мы узнаем о функциях больше.

6.15. Инструкция return

Как вы помните, вызов функции с помощью оператора () представляет собой выражение. Все выражения имеют значения, и инструкция return служит для определения значения, возвращаемого функцией. Это значение становится значением выражения вызова функции. Инструкция return имеет следующий синтаксис:

```
return выражение;
```

Инструкция return может располагаться только в теле функции. Присутствие ее в любом другом месте является синтаксической ошибкой. Когда выполняется инструкция return, вычисляется выражение и его значение возвращается в качестве значения функции. Инструкция return прекращает исполнение функции, даже если в теле функции остались другие инструкции. Инструкция return используется для возвращения значения следующим образом:

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

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

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

```
    // Здесь находится оставшаяся часть функции...
}
```

Если в функции выполняется инструкция `return` без выражения или если выполнение функции прекращается по причине достижения конца тела функции, значение выражения вызова функции оказывается неопределенным (`undefined`).

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

6.16. Инструкция throw

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

Инструкция `throw` имеет следующий синтаксис:

```
throw выражение;
```

Результатом выражения может быть значение любого типа. Однако обычно это объект `Error` или экземпляр одного из подклассов `Error`. Также бывает удобно использовать в качестве выражения строку, содержащую сообщение об ошибке или числовое значение, обозначающее некоторый код ошибки. Вот пример кода, в котором инструкция `throw` применяется для генерации исключения:

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

Когда генерируется исключение, интерпретатор JavaScript немедленно прерывает нормальное исполнение программы и переходит к ближайшему[2] обработчику исключений. В обработчиках исключений используется конструкция `catch` инструкции `try/catch/finally`, описание которой приведено в следующем разделе. Если блок кода, в котором возникло исключение, не имеет соответствующей конструкции `catch`, интерпретатор анализирует следующий внешний блок кода

---

[1] Инструкции `throw` и `try/catch/finally` в JavaScript напоминают соответствующие инструкции в C++ и Java.

[2] К самому внутреннему по вложенности охватывающему обработчику исключений. – *Примеч. науч. ред.*

и проверяет, связан ли с ним обработчик исключений. Это продолжается до тех пор, пока обработчик не будет найден. Если исключение генерируется в функции, не содержащей инструкции try/catch/finally, предназначенной для его обработки, то исключение распространяется на код, вызвавший функцию. Так исключения распространяются по лексической структуре методов JavaScript вверх по стеку вызовов. Если обработчик исключения так и не будет найден, исключение рассматривается как ошибка и о ней сообщается пользователю.

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

6.17. Инструкция try/catch/finally

Инструкция try/catch/finally реализует механизм обработки исключений в JavaScript. Конструкция try в этой инструкции просто определяет блок кода, в котором обрабатываются исключения. За блоком try следует конструкция catch с блоком инструкций, вызываемых, когда где-либо в блоке try возникает исключение. За конструкцией catch следует блок finally, содержащий код зачистки, который гарантированно выполняется независимо от того, что происходит в блоке try. И блок catch, и блок finally не являются обязательными, однако после блока try должен обязательно присутствовать хотя бы один из них. Блоки try, catch и finally начинаются и заканчиваются фигурными скобками. Это обязательная часть синтаксиса и она не может быть опущена, даже если между ними содержится только одна инструкция. Как и инструкция throw, инструкция try/catch/finally стандартизована в ECMAScript v3 и реализована в JavaScript 1.4.

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

```
try {
    // Обычно этот код без сбоев работает от начала до конца.
    // Но в какой-то момент в нем может генерироваться исключение
    // либо непосредственно с помощью инструкции throw, либо косвенно
    // вызовом метода, генерирующего исключение.
}
catch (e) {
    // Инструкции в этом блоке выполняются тогда и только тогда, когда
    // в блоке try генерируется исключение. Эти инструкции могут
    // использовать локальную переменную e, ссылающуюся на объект Error
    // или на другое значение, указанное в инструкции throw. Этот блок может
    // либо каким-либо образом обработать исключение, либо проигнорировать
    // его, делая что-то другое, либо заново сгенерировать исключение
    // с помощью инструкции throw.
}
finally {
    // Этот блок содержит инструкции, которые выполняются всегда, независимо от того,
    // что произошло в блоке try. Они выполняются, если блок try прерван:
```

```
    //   1) нормальным образом, достигнув конца блока
    //   2) из-за инструкции break, continue или return
    //   3) с исключением, обработанным приведенным ранее блоком catch
    //   4) с неперехваченным исключением, которое продолжает свое
    //      распространение на более высокие уровни
}
```

Далее приводится более реалистичный пример инструкции try/catch. В нем вызываются метод factorial(), определенный в предыдущем разделе, и методы prompt() и alert() клиентского языка JavaScript для организации ввода и вывода:

```
try {
    // Просим пользователя ввести число
    var n = prompt("Введите положительное число", "");
    // Вычисляем факториал числа, предполагая, что входные данные корректны
    var f = factorial(n);
    // Показываем результат
    alert(n + "! = " + f);
}
catch (ex) { // Если введенные данные некорректны, мы попадем сюда
    // Сообщаем пользователю об ошибке
    alert(ex);
}
```

Это пример инструкции try/catch без конструкции finally. Хотя finally используется не так часто, как catch, тем не менее иногда эта конструкция оказывается полезной. Однако ее поведение требует дополнительных объяснений. Блок finally гарантированно исполняется, если исполнялась хотя бы какая-то часть блока try, независимо от того каким образом завершился код в блоке try. Эта возможность обычно используется для зачистки после выполнения кода в предложении try.

В обычной ситуации управление доходит до конца блока try, а затем переходит к блоку finally, который выполняет всю необходимую зачистку. Если управление вышло из блока try из-за инструкций return, continue или break, перед передачей управления в другое место кода исполняется блок finally.

Если в блоке try возникает исключение и имеется соответствующий блок catch для его обработки, управление сначала передается в блок catch, а затем – в блок finally. Если отсутствует локальный блок catch, то управление сначала передается в блок finally, а затем переходит на ближайший внешний блок catch, который может обработать исключение.

Если сам блок finally передает управление с помощью инструкции return, continue, break или throw или путем вызова метода, генерирующего исключение, незаконченная команда на передачу управления отменяется и выполняется новая. Например, если блок finally генерирует исключение, это исключение заменяет любое сгенерированное исключение. Если в блоке finally имеется инструкция return, происходит нормальный выход из метода, даже если генерировалось исключение, которое не было обработано.

Инструкции try и finally могут использоваться вместе без конструкции catch. В этом случае блок finally – это просто код зачистки, который будет гарантированно исполнен независимо от наличия в блоке try инструкции break, continue

или return. Например, в следующем коде используется инструкция try/finally, гарантирующая, что счетчик цикла будет инкрементирован в конце каждой итерации, даже если итерация внезапно прервется инструкцией continue:

```
var i = 0, total = 0;
while(i < a.length) {
    try {
        if ((typeof a[i] != "number") || isNaN(a[i])) // Если это не число,
            continue;  // переходим к следующей итерации цикла.
        total += a[i]; // В противном случае добавляем число к общей сумме.
    }
    finally {
        i++;           // Всегда увеличиваем i, даже если ранее была инструкция continue.
    }
}
```

6.18. Инструкция with

В главе 4 мы обсуждали область видимости переменных и цепочку областей видимости – список объектов, в которых выполняется поиск при разрешении имени переменной. Инструкция with используется для временного изменения цепочки областей видимости. Она имеет следующий синтаксис:

```
with (объект)
    инструкция
```

Эта инструкция добавляет *объект* в начало цепочки областей видимости, исполняет *инструкцию*, а затем восстанавливает цепочку в ее первоначальном состоянии.

На практике инструкция with помогает значительно сократить объем набираемого текста. В клиентском языке JavaScript часто работают с глубоко вложенными иерархиями объектов. Например, для доступа к элементам HTML-формы вам, возможно, придется пользоваться такими выражениями:

```
frames[1].document.forms[0].address.value
```

Если надо обратиться к этой форме несколько раз, можно воспользоваться инструкцией with для добавления формы в цепочку областей видимости:

```
with(frames[1].document.forms[0]) {
    // Здесь обращаемся к элементам формы непосредственно, например:
    name.value = "";
    address.value = "";
    email.value = "";
}
```

Это сокращает объем текста программы – больше не надо указывать фрагмент frames[1].document.forms[0] перед каждым именем свойства. Этот объект представляет собой временную часть цепочки областей видимости и автоматически участвует в поиске, когда JavaScript требуется разрешить такой идентификатор, как address.

Несмотря на удобство этой конструкции в некоторых случаях, ее использование не приветствуется. JavaScript-код с инструкцией with сложен в оптимизации и поэтому может работать медленнее, чем эквивалентный код, написанный без

нее. Кроме того, определения функций и инициализация переменных в теле инструкции `with` могут приводить к странным и трудным для понимания результатам.[1] По этим причинам использовать инструкцию `with` не рекомендуется.

К тому же существуют и другие абсолютно законные способы уменьшения объема набираемого текста. Так, предыдущий пример можно переписать следующим образом:

```
var form = frames[1].document.forms[0];
form.name.value = "";
form.address.value = "";
form.email.value = "";
```

6.19. Пустая инструкция

И наконец, последняя из допустимых в JavaScript инструкций – пустая инструкция. Она выглядит следующим образом:

```
;
```

Выполнение пустой инструкции, очевидно, не имеет никакого эффекта и не производит никаких действий. Можно подумать, что особых причин для ее применения нет, однако изредка пустая инструкция может быть полезна, когда требуется создать цикл, имеющий пустое тело. Например:

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

Обратите внимание, что случайное указание точки с запятой после правой круглой скобки в циклах `for` и `while` или в инструкции `if` может привести к неприятным ошибкам, которые сложно обнаружить. Например, следующий фрагмент вряд ли делает то, что предполагал его автор:

```
if ((a == 0) || (b == 0)); // Ой! Эта строка ничего не делает...
    o = null;              // а эта строка выполняется всегда.
```

Когда пустая инструкция применяется специально, код желательно снабжать исчерпывающими комментариями. Например:

```
for(i=0; i < a.length; a[i++] = 0) /* Пустое тело цикла */ ;
```

6.20. Итоговая таблица JavaScript-инструкций

В этой главе мы представили все инструкции языка JavaScript. В табл. 6.1 содержится перечень этих инструкций с указанием синтаксиса и назначения каждой из них.

[1] Эти результаты и их причины слишком сложны, чтобы объяснять их здесь.

Таблица 6.1. Синтаксис JavaScript-инструкций

Инструкция	Синтаксис	Назначение
break	break; break *имя_метки*;	Выход из самого внутреннего цикла инструкции switch или инструкции с именем *имя_метки*
case	case *выражение*:	Метка для инструкции внутри конструкции switch
continue	continue; continue *имя_метки*;	Перезапуск самого внутреннего цикла или цикла, помеченного меткой *имя_метки*
default	default:	Отметка инструкции по умолчанию внутри инструкции switch
do/while	do *инструкция* while (*выражение*);	Альтернатива циклу while
Пустая инструкция	;	Ничего не делает
for	for (*инициализация*; *проверка*; *инкремент*) *инструкция*	Простой в использовании цикл
for/in	for (*переменная* in *объект*) инструкция	Цикл по свойствам объекта

function	function *имя_функции*([*арг1*[..., *аргn*]])
	{
	  инструкции
	}
		Объявление функции

if/else	if (*выражение*) инструкция1 [else *инструкция2*]	Условное исполнение фрагмента программы
Метка	*идентификатор*: *инструкция*	Присваивание *инструкции* имени идентификатор
return	return [*выражение*];	Возврат из функции или задание возвращаемого функцией значения, равным *выражению*

switch	switch (*выражение*) {
	  инструкции
	}
		Многопозиционное ветвление для инструкций, помеченных метками *case* и *default*

throw	throw *выражение*;	Генерация исключения

Инструкция	Синтаксис	Назначение

try	try {
	  инструкции
	}
	catch (*идентификатор*) {
	  инструкции
	}
	finally {
	  инструкции
	}
		Перехват исключения

var	var *имя_1* [= *значение_1*] [. . . , *имя_n* [= *значение_n*]];	Объявление и инициализация переменных
while	while (*выражение*) инструкция	Базовая конструкция для цикла
with	with (*объект*) инструкция	Расширение цепочки областей видимости (не рекомендуется к применению)

7

Объекты и массивы

В главе 3 говорилось, что объекты и массивы – это два фундаментальных и наиболее важных типа данных в JavaScript. Объекты и массивы отличаются от элементарных типов данных, таких как строки или числа, тем, что они представляют не единственное значение, а целые их наборы. Объекты являются коллекциями именованных свойств, а массивы представляют собой специализированные объекты, которые ведут себя как упорядоченные коллекции пронумерованных значений. В этой главе мы детально рассмотрим объекты и массивы языка JavaScript.

7.1. Создание объектов

Объекты – это составной тип данных, они объединяют множество значений в единый модуль и позволяют сохранять и извлекать значения по их именам. Говоря другими словами, объекты – это неупорядоченные коллекции *свойств*, каждое из которых имеет свои имя и значение. Именованные значения, хранящиеся в объекте, могут быть данными элементарных типов, такими как числа или строки, или сами могут быть объектами.

Самый простой способ создания объектов заключается во включении в программу литерала объекта. *Литерал объекта* – это заключенный в фигурные скобки список свойств (пар «имя–значение»), разделенных запятыми. Имя каждого свойства может быть JavaScript-идентификатором или строкой, а значением любого свойства может быть константа или JavaScript-выражение. Несколько примеров создания объектов:

```
var empty = {}; // Объект без свойств
var point = { x:0, y:0 };
var circle = { x:point.x, y:point.y+1, radius:2 };
var homer = {
    "name": "Homer Simpson",
    "age": 34,
    "married": true,
    "occupation": "plant operator",
```

```
    'email': "homer@example.com"
};
```

Литерал объекта — это выражение, которое создает и инициализирует новый объект всякий раз, когда производится вычисление этого выражения. Таким образом, с помощью единственного литерала объекта можно создать множество новых объектов, если этот литерал поместить в тело цикла или функции, которая будет вызываться многократно.

С помощью оператора `new` можно создать другую разновидность объектов. За этим оператором должно быть указано имя функции-конструктора, выполняющей инициализацию свойств объекта. Например:

```
var a = new Array( ); // Создать пустой массив
var d = new Date( );  // Создать объект с текущими временем и датой
var r = new RegExp("javascript", "i"); // Создать объект регулярного выражения
```

Продемонстрированные здесь функции `Array()`, `Date()` и `RegExp()` являются встроенными конструкторами базового языка JavaScript. (Конструктор `Array()` описан далее в этой главе, описание других конструкторов можно найти в третьей части книги.) Конструктор `Object()` создает пустой объект, как если бы использовался литерал `{}`.

Существует возможность определять собственные конструкторы для инициализации вновь создаваемых объектов необходимым вам способом. Как это делается, показано в главе 9.

7.2. Свойства объектов

Обычно для доступа к значениям свойств объекта используется оператор `.` (точка). Значение в левой части оператора должно быть ссылкой на объект, к свойствам которого требуется получить доступ. Обычно это просто имя переменной, содержащей ссылку на объект, но это может быть любое допустимое в JavaScript выражение, являющееся объектом. Значение в правой части оператора должно быть именем свойства. Это должен быть идентификатор, а не строка или выражение. Так, обратиться к свойству `p` объекта `o` можно посредством выражения `o.p`, а к свойству `radius` объекта `circle` — посредством выражения `circle.radius`. Свойства объекта работают как переменные: в них можно сохранять значения и считывать их. Например:

```
// Создаем объект. Сохраняем ссылку на него в переменной.
var book = new Object();

// Устанавливаем свойство в объекте.
book.title = "JavaScript: полное руководство"

// Устанавливаем другие свойства. Обратите внимание на вложенные объекты.
book.chapter1 = new Object();
book.chapter1.title = "Введение в JavaScript";
book.chapter1.pages = 11;
book.chapter2 = { title: "Лексическая структура", pages: 6 };

// Читаем значения некоторых свойств из объекта.
alert("Заголовок: " + book.title + "\n\t" +
```

```
        "Глава 1 " + book.chapter1.title + "\n\t" +
        "Глава 2 " + book.chapter2.title);
```

Важно обратить внимание на один момент в этом примере – новое свойство объекта можно добавить, просто присвоив этому свойству значение. Если переменные должны объявляться с помощью ключевого слова `var`, то для свойств объекта такой необходимости (и возможности) нет. К тому же после создания свойства объекта (в результате присваивания) значение свойства можно будет изменить в любой момент простым присваиванием ему нового значения:

```
book.title = "JavaScript: Книга с носорогом"
```

7.2.1. Перечисление свойств

Цикл `for/in`, который обсуждался в главе 6, предоставляет средство, позволяющее перебрать, или перечислить, свойства объекта. Это обстоятельство можно использовать при отладке сценариев или при работе с объектами, которые могут иметь произвольные свойства с заранее неизвестными именами. В следующем фрагменте демонстрируется функция, которая выводит список имен свойств объекта:

```
function DisplayPropertyNames(obj) {
    var names = "";
    for(var name in obj) names += name + "\n";
    alert(names);
}
```

Обратите внимание, что цикл `for/in` не перечисляет свойства в каком-либо заданном порядке, и хотя он перечисляет все свойства, определенные пользователем, некоторые предопределенные свойства и методы он не перечисляет.

7.2.2. Проверка существования свойств

Для проверки факта наличия того или иного свойства у объекта может использоваться оператор `in` (см. главу 5). С левой стороны от оператора помещается имя свойства в виде строки, с правой стороны – проверяемый объект. Например:

```
// Если объект o имеет свойство с именем "x", установить его
if ("x" in o) o.x = 1;
```

Однако потребность в операторе `in` возникает не так часто, потому что при обращении к несуществующему свойству возвращается значение `undefined`. Таким образом, указанный фрагмент обычно записывается следующим образом:

```
// Если свойство x существует и его значение
// не равно undefined, установить его.
if (o.x !== undefined) o.x = 1;
```

Обратите внимание: есть вероятность, что свойство фактически существует, но еще не определено. Например, если записать такую строку:

```
o.x = undefined
```

то свойство `x` будет существовать, но не будет иметь значения. В этом случае в первом из показанных фрагментов в свойство `x` будет записано значение 1, во втором – нет.

Кроме того, обратите внимание, что вместо обычного оператора != был использован оператор !==. Операторы !== и === различают значения undefined и null, хотя иногда в этом нет необходимости:

```
// Если свойство doSomething существует и не содержит значение null
// или undefined, тогда предположить, что это функция и ее следует вызвать!
if (o.doSomething) o.doSomething();
```

7.2.3. Удаление свойств

Для удаления свойства объекта предназначен оператор delete:

```
delete book.chapter2;
```

Обратите внимание, что при удалении свойства его значение не просто устанавливается в значение undefined; оператор delete действительно удаляет свойство из объекта. Цикл for/in демонстрирует это отличие: он перечисляет свойства, которым было присвоено значение undefined, но не перечисляет удаленные свойства.

7.3. Объекты как ассоциативные массивы

Как мы знаем, доступ к свойствам объекта осуществляется посредством оператора «точка». Доступ к свойствам объекта возможен также при помощи оператора [], который обычно применяется при работе с массивами. Таким образом, следующие два JavaScript-выражения имеют одинаковое значение:

```
object.property
object["property"]
```

Важное различие между этими двумя синтаксисами, на которое следует обратить внимание, состоит в том, что в первом варианте имя свойства представляет собой идентификатор, а во втором – строку. Скоро мы узнаем, почему это так важно.

В Java, C, C++ и подобных языках со строгой типизацией объект может иметь только фиксированное число свойств, и имена этих свойств должны быть определены заранее. Поскольку JavaScript – слабо типизированный язык, к нему данное правило неприменимо; программа может создавать любое количество свойств в любом объекте. Однако в случае использования оператора «точка» для доступа к свойству объекта имя свойства задается идентификатором. Идентификаторы должны быть частью текста JavaScript-программы – они не являются типом данных и ими нельзя манипулировать из программы.

В то же время при обращении к свойству объекта с помощью нотации массивов [] имя свойства задается в виде строки. Строки в JavaScript – это тип данных, поэтому они могут создаваться и изменяться во время работы программы. И поэтому в JavaScript можно, например, написать следующий код:

```
var addr = "";
for(i = 0; i < 4; i++) {
    addr += customer["address" + i] + '\n';
}
```

В этом фрагменте читаются и объединяются в одну строку свойства address0, address1, address2 и address3 объекта customer.

Этот короткий пример демонстрирует гибкость нотации массивов при обращении к свойствам объекта с помощью строковых выражений. Мы могли бы написать этот пример и с помощью оператора «точка», но есть ситуации, где подойдет только нотация массивов. Предположим, что вы пишете программу, обращающуюся к сетевым ресурсам для вычисления текущей суммы инвестиций пользователя на фондовом рынке. Программа разрешает пользователю вводить названия любых имеющихся у него акций, а также количество каждого вида акций. Можно организовать хранение этой информации при помощи объекта с именем `portfolio`, имеющего по одному свойству для акций каждого вида. Имя свойства – это название акции, а значение свойства – количество акций данного вида. Другими словами, если, например, у пользователя имеется 50 акций IBM, то свойство `portfolio.ibm` имеет значение 50.

Частью этой программы должен быть цикл, запрашивающий у пользователя название имеющихся у него акций, а затем количество акций данного типа. Внутри цикла должен быть код, похожий на следующий:

```
var stock_name = get_stock_name_from_user();
var shares = get_number_of_shares();
portfolio[stock_name] = shares;
```

Поскольку пользователь вводит названия акций во время исполнения программы, нет способа узнать имена свойств заранее. А раз имена свойств при написании программы неизвестны, то доступ к свойствам объекта `portfolio` при помощи оператора «точка» невозможен. Однако можно обратиться к оператору `[]`, т. к. в нем для имени свойства используется строковое значение (которое может изменяться во время выполнения), а не идентификатор (который должен быть задан непосредственно в тексте программы).

Когда объект используется в такой форме, его часто называют *ассоциативным массивом* – структурой данных, позволяющей связывать произвольные значения с произвольными строками. Нередко для описания этой ситуации используется термин *отображение* (*map*): JavaScript-объекты отображают строки (имена свойств) на их значения.

Использование точки (.) для доступа к свойствам делает их похожими на статические объекты в языках C++ и Java, и они прекрасно работают в этой роли. Но они также предоставляют мощное средство для связи значений с произвольными строками. В этом отношении JavaScript-объекты значительно больше похожи на массивы в Perl, чем на объекты в C++ или Java.

В главе 6 был введен цикл `for/in`. Настоящая мощь этой JavaScript-конструкции становится понятной при ее использовании с ассоциативными массивами. Возвращаясь к примеру с портфелем акций, после ввода пользователем данных по своему портфелю вычислить текущую общую стоимость последнего можно при помощи следующего кода:

```
var value = 0;
for (stock in portfolio) {
    // Для каждого вида акций в портфеле получаем стоимость
    // одной акции и умножаем ее на число акций.
    value += get_share_value(stock) * portfolio[stock];
}
```

Здесь не обойтись без цикла `for/in`, поскольку названия акций заранее неизвестны. Это единственный способ извлечения имен этих свойств из ассоциативного массива (JavaScript-объекта) по имени `portfolio`.

7.4. Свойства и методы универсального класса Object

Как уже отмечалось, все объекты в JavaScript наследуют свойства и методы класса `Object`. При этом специализированные классы объектов, как, например, те, что создаются с помощью конструкторов `Date()` или `RegExp()`, определяют собственные свойства и методы, но все объекты независимо от своего происхождения помимо всего прочего поддерживают свойства и методы, определенные классом `Object`. По причине их универсальности эти свойства и методы представляют особый интерес.

7.4.1. Свойство constructor

В JavaScript любой объект имеет свойство `constructor`, которое ссылается на функцию-конструктор, используемую для инициализации объекта. Например, если объект `d` создается с помощью конструктора `Date()`, то свойство `d.constructor` ссылается на функцию `Date`:

```
var d = new Date( );
d.constructor == Date; // Равно true
```

Функция-конструктор определяет категорию, или класс, объекта, поэтому свойство `constructor` может использоваться для определения типа любого заданного объекта. Например, тип неизвестного объекта можно выяснить таким способом:

```
if ((typeof o == "object") && (o.constructor == Date))
    // Какие-то действия с объектом Date...
```

Проверить значение свойства `constructor` можно с помощью оператора `instanceof`, т. е. приведенный фрагмент можно записать несколько иначе:

```
if ((typeof o == "object") && (o instanceof Date))
    // Какие-то действия с объектом Date...
```

7.4.2. Метод toString()

Метод `toString()` не требует аргументов; он возвращает строку, каким-либо образом представляющую тип и/или значение объекта, для которого он вызывается. Интерпретатор JavaScript вызывает этот метод объекта во всех тех случаях, когда ему требуется преобразовать объект в строку. Например, это происходит, когда используется оператор + для конкатенации строки с объектом, или при передаче объекта такому методу, как `alert()` или `document.write()`.

Метод `toString()` по умолчанию не очень информативен. Например, следующий фрагмент просто записывает в переменную `s` строку `"[object Object]"`:

```
var s = { x:1, y:1 }.toString( );
```

Этот метод по умолчанию не отображает особенно полезной информации, поэтому многие классы определяют собственные версии метода `toString()`. Например,

когда массив преобразуется в строку, мы получаем список элементов массива, каждый из которых преобразуется в строку, а когда в строку преобразуется функция, мы получаем исходный код этой функции.

В главе 9 описывается, как можно переопределить метод `toString()` для своих собственных типов объектов.

7.4.3. Метод toLocaleString()

В ECMAScript v3 и JavaScript 1.5 класс `Object` в дополнение к методу `toString()` определяет метод `toLocaleString()`. Назначение последнего состоит в получении локализованного строкового представления объекта. По умолчанию метод `toLocaleString()`, определяемый классом `Object`, никакой локализации не выполняет; он всегда возвращает в точности такую же строку, что и `toString()`. Однако подклассы могут определять собственные версии метода `toLocaleString()`. В ECMAScript v3 классы `Array`, `Date` и `Number` определяют версии метода `toLocaleString()`, возвращающие локализованные значения.

7.4.4. Метод valueOf()

Метод `valueOf()` во многом похож на метод `toString()`, но вызывается, когда интерпретатору JavaScript требуется преобразовать объект в значение какого-либо элементарного типа, отличного от строки, – обычно в число. Интерпретатор JavaScript вызывает этот метод автоматически, если объект используется в контексте значения элементарного типа. По умолчанию метод `valueOf()` не выполняет ничего, что представляло бы интерес, но некоторые встроенные категории объектов переопределяют метод `valueOf()` (например, `Date.valueOf()`). В главе 9 описывается, как можно переопределить метод `valueOf()` в собственных типах объектов.

7.4.5. Метод hasOwnProperty()

Метод `hasOwnProperty()` возвращает `true`, если для объекта определено не унаследованное свойство с именем, указанным в единственном строковом аргументе метода. В противном случае он возвращает `false`. Например:

```
var o = {};
o.hasOwnProperty("undef");    // false: свойство не определено
o.hasOwnProperty("toString"); // false: toString - это унаследованное свойство
Math.hasOwnProperty("cos");   // true: объект Math имеет свойство cos
```

Порядок наследования свойств описывается в главе 9.

Метод `hasOwnProperty()` определяется стандартом ECMAScript v3 и реализован в JavaScript 1.5 и более поздних версиях.

7.4.6. Метод propertyIsEnumerable()

Метод `propertyIsEnumerable()` возвращает `true`, если в объекте определено свойство с именем, указанным в единственном строковом аргументе метода, и это свойство может быть перечислено циклом `for/in`. В противном случае метод возвращает `false`. Например:

```
var o = { x:1 };
```

```
o.propertyIsEnumerable("x");       // true: свойство существует и является перечислимым
o.propertyIsEnumerable("y");       // false: свойство не существует
o.propertyIsEnumerable("valueOf"); // false: свойство неперечислимое
```

Метод `propertyIsEnumerable()` определяется стандартом ECMAScript v3 и реализован в JavaScript 1.5 и более поздних версиях.

Обратите внимание: все свойства объекта, определяемые пользователем, являются перечислимыми. Неперечислимыми обычно являются унаследованные свойства (тема наследования свойств рассматривается в главе 9), поэтому практически всегда этот метод возвращает то же значение, что и метод `hasOwnProperty()`.

7.4.7. Метод isPrototypeOf()

Метод `isPrototypeOf()` возвращает `true`, если объект, которому принадлежит метод, является прототипом объекта, передаваемого методу в качестве аргумента. В противном случае метод возвращает `false`. Например:

```
var o = {};
Object.prototype.isPrototypeOf(o);        // true: o.constructor == Object
Object.isPrototypeOf(o);                  // false
o.isPrototypeOf(Object.prototype);        // false
Function.prototype.isPrototypeOf(Object); // true: Object.constructor == Function
```

7.5. Массивы

Массив – это тип данных, содержащий (хранящий) пронумерованные значения. Каждое пронумерованное значение называется *элементом* массива, а число, с которым связывается элемент, называется его *индексом*. Так как JavaScript – это нетипизированный язык, элемент массива может иметь любой тип, причем разные элементы одного массива могут иметь разные типы. Элементы массива могут даже содержать другие массивы, что позволяет создавать массивы массивов.

На протяжении всей книги мы часто рассматриваем объекты и массивы как отдельные типы данных. Это полезное и разумное упрощение – в JavaScript объекты и массивы можно рассматривать как разные типы для большинства задач программирования. Однако, чтобы хорошо понять поведение объектов и массивов, следует знать правду: массив – это не что иное, как объект с тонким слоем дополнительной функциональности. Это можно увидеть, определив тип массива с помощью оператора `typeof` – будет получена строка `"object"`.

Легче всего создать массив можно с помощью литерала, который представляет собой простой список разделенных запятыми элементов массива в квадратных скобках. Например:

```
var empty = [];                // Пустой массив
var primes = [2, 3, 5, 7, 11]; // Массив с пятью числовыми элементами
var misc = [ 1.1, true, "a" ]; // 3 элемента разных типов
```

Значения в литерале массива не обязательно должны быть константами – это могут быть любые выражения:

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

Литералы массивов могут содержать литералы объектов или литералы других массивов:

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

Во вновь созданном массиве первое значение литерала массива сохраняется в элементе с индексом 0, второе значение – в элементе с индексом 1, и т. д. Если в литерале значение элемента опущено, будет создан элемент с неопределенным значением:

```
var count = [1,,3]; // Массив из 3 элементов, средний элемент не определен.
var undefs = [,,];  // Массив из 2 элементов, оба не определены.
```

Другой способ создания массива состоит в вызове конструктора `Array()`. Вызывать конструктор можно тремя разными способами:

- Вызов конструктора без аргументов:

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

  В этом случае будет создан пустой массив, эквивалентный литералу `[]`.

- Конструктору явно указываются значения первых *n* элементов массива:

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

  В этом случае конструктор получает список аргументов. Каждый аргумент определяет значение элемента и может иметь любой тип. Нумерация элементов массива начинается с 0. Свойство `length` массива устанавливается равным количеству элементов, переданных конструктору.

- Вызов с единственным числовым аргументом, определяющим длину массива:

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

  Эта форма позволяет создать массив с заданным количеством элементов (каждый из которых имеет значение `undefined`) и устанавливает свойство `length` массива равным указанному значению. Эта форма обращения к конструктору `Array()` может использоваться для предварительного размещения массива, если его длина известна заранее. В этой ситуации литералы массивов не очень удобны.

7.6. Чтение и запись элементов массива

Доступ к элементам массива осуществляется с помощью оператора `[]`. Слева от скобок должна присутствовать ссылка на массив. Внутри скобок должно находиться произвольное выражение, имеющее неотрицательное целое значение. Этот синтаксис пригоден как для чтения, так и для записи значения элемента массива. Следовательно, все приведенные далее JavaScript-инструкции допустимы:

```
value = a[0];
a[1] = 3.14;
i = 2;
a[i] = 3;
a[i + 1] = "hello";
a[a[i]] = a[0];
```

В некоторых языках первый элемент массива имеет индекс 1. Однако в JavaScript (как в C, C++ и Java) первый элемент массива имеет индекс 0.

Как уже отмечалось, оператор [] может также использоваться для доступа к именованным свойствам объекта:

```
my['salary'] *= 2;
```

Поскольку массивы являются специализированным классом объектов, существует возможность определять нечисловые свойства объекта и обращаться к ним посредством операторов . (точка) и [].

Обратите внимание, что индекс массива должен быть неотрицательным числом, меньшим $2^{32}-1$. Если число слишком большое, отрицательное или вещественное (или это логическое, объектное или другое значение), JavaScript преобразует его в строку и рассматривает результирующую строку как имя свойства объекта, а не как индекс массива. Таким образом, следующая строка создаст новое свойство с именем "-1.23", а не новый элемент массива:

```
a[-1.23] = true;
```

7.6.1. Добавление новых элементов в массив

В таких языках, как C и Java, массив имеет фиксированное число элементов, которое должно быть задано при создании массива. Это не относится к JavaScript – массив в JavaScript может иметь любое количество элементов, и это количество можно в любой момент изменить.

Чтобы добавить новый элемент в массив, достаточно присвоить ему значение:

```
a[10] = 10;
```

Массивы в JavaScript могут быть *разреженными*. Это значит, что индексы массива не обязательно принадлежат непрерывному диапазону чисел; реализация JavaScript может выделять память только для тех элементов массива, которые фактически в нем хранятся. Поэтому в результате выполнения следующего фрагмента интерпретатор JavaScript скорее всего выделит память только для элементов массива с индексами 0 и 10 000, но не выделит ее для 9 999 элементов, находящихся между ними:

```
a[0] = 1;
a[10000] = "это элемент 10,000";
```

Обратите внимание: элементы массива могут также добавляться к объектам:

```
var c = new Circle(1,2,3);
c[0] = "это элемент массива в объекте!"
```

Этот пример просто определяет новое свойство объекта с именем "0". Однако добавление элемента массива в объект не делает объект массивом.

7.6.2. Удаление элементов массива

Оператор delete записывает в элемент массива значение undefined, при этом сам элемент массива продолжает свое существование. Для удаления элементов так, чтобы остающиеся элементы сместились к началу массива, необходимо воспользоваться одним из методов массива. Метод Array.shift() удаляет первый элемент

массива, метод Array.pop() — последний элемент массива, метод Array.splice() — непрерывный диапазон элементов. Эти функции описываются далее в этой главе, а также в третьей части книги.

7.6.3. Длина массива

Все массивы, как созданные с помощью конструктора Array(), так и определенные с помощью литерала массива, имеют специальное свойство length, устанавливающее количество элементов в массиве. Поскольку массивы могут иметь неопределенные элементы, более точная формулировка звучит так: свойство length *всегда* на единицу больше, чем самый большой номер элемента массива. В отличие от обычных свойств объектов, свойство length массива автоматически обновляется, оставаясь корректным при добавлении новых элементов в массив. Это обстоятельство иллюстрирует следующий фрагмент:

```
var a = new Array();  // a.length == 0 (ни один элемент не определен)
a = new Array(10);    // a.length == 10 (определены пустые элементы 0-9)
a = new Array(1,2,3); // a.length == 3 (определены элементы 0-2)
a = [4, 5];           // a.length == 2 (определены элементы 0 и 1)
a[5] = -1;            // a.length == 6 (определены элементы 0, 1 и 5)
a[49] = 0;            // a.length == 50 (определены элементы 0, 1, 5 и 49)
```

Помните, что индексы массива должны быть меньше $2^{32}-1$, т. е. максимально возможное значение свойства length равно $2^{32}-1$.

7.6.4. Обход элементов массива

Наиболее часто свойство length используется для перебора элементов массива в цикле:

```
var fruits = ["манго", "банан", "вишня", "персик"];
for(var i = 0; i < fruits.length; i++)
    alert(fruits[i]);
```

Конечно, в этом примере предполагается, что элементы массива расположены непрерывно и начинаются с элемента 0. Если это не так, перед обращением к каждому элементу массива нужно проверять, определен ли он:

```
for(var i = 0; i < fruits.length; i++)
    if (fruits[i] != undefined) alert(fruits[i]);
```

Аналогичный подход может использоваться для инициализации элементов массива, созданного вызовом конструктора Array():

```
var lookup_table = new Array(1024);
for(var i = 0; i < lookup_table.length; i++)
    lookup_table[i] = i * 512;
```

7.6.5. Усечение и увеличение массива

Свойство length массива доступно как для чтения, так и для записи. Если установить свойство length в значение, меньшее текущего, массив укорачивается до новой длины; любые элементы, не попадающие в новый диапазон индексов, отбрасываются, и их значения теряются.

Если сделать свойство length большим, чем его текущее значение, в конец массива добавляются новые неопределенные элементы, увеличивая массив до нового размера.

Обратите внимание: хотя объектам могут быть присвоены элементы массива, объекты не имеют свойства length. Это свойство и его специальное поведение – наиболее важная особенность, свойственная массивам. Другие особенности, отличающие массивы от объектов, – это различные методы, определяемые классом Array и описываемые в разделе 7.7.

7.6.6. Многомерные массивы

JavaScript не поддерживает «настоящие» многомерные массивы, но позволяет неплохо имитировать их при помощи массивов из массивов. Для доступа к элементу данных в массиве массивов достаточно использовать оператор [] дважды. Например, предположим, что переменная matrix – это массив массивов чисел. Любой элемент matrix[x] – это массив чисел. Для доступа к определенному числу в массиве надо написать matrix[x][y]. Вот конкретный пример, в котором двухмерный массив используется в качестве таблицы умножения:

```
// Создать многомерный массив
var table = new Array(10);            // В таблице 10 строк
for(var i = 0; i < table.length; i++)
    table[i] = new Array(10);         // В каждой строке 10 столбцов

// Инициализация массива
for(var row = 0; row < table.length; row++) {
    for(col = 0; col < table[row].length; col++) {
        table[row][col] = row*col;
    }
}

// Расчет произведения 5*7 с помощью многомерного массива
var product = table[5][7];            // 35
```

7.7. Методы массивов

Помимо оператора [] с массивами можно работать посредством различных методов, предоставляемых классом Array. Эти методы представлены в следующих разделах. Многие из методов позаимствованы из языка программирования Perl; программистам, работавшим с Perl, они могут показаться знакомыми. Как обычно, здесь приведен только их обзор, а полные описания находятся в третьей части книги.

7.7.1. Метод join()

Метод Array.join() преобразует все элементы массива в строки и объединяет их. Можно указать необязательный строковый аргумент, предназначенный для разделения элементов в результирующей строке. Если разделитель не задан, используется запятая. Например, следующий фрагмент дает в результате строку "1,2,3":

```
var a = [1, 2, 3]; // Создает новый массив с указанными тремя элементами
var s = a.join();  // s == "1,2,3"
```

В следующем примере задается необязательный разделитель, что приводит к несколько иному результату:

```
s = a.join(", "); // s == "1, 2, 3"
```

Обратите внимание на пробел после запятой.

Метод `Array.join()` является обратным по отношению к методу `String.split()`, создающему массив путем разбиения строки на фрагменты.

7.7.2. Метод reverse()

Метод `Array.reverse()` меняет порядок следования элементов в массиве на противоположный и возвращает массив с переставленными элементами. Он делает это на месте, другими словами, этот метод не создает новый массив с переупорядоченными элементами, а переупорядочивает их в уже существующем массиве. Например, следующий фрагмент, где используются методы `reverse()` и `join()`, дает в результате строку `"3,2,1"`:

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

7.7.3. Метод sort()

Метод `Array.sort()` на месте сортирует элементы массива и возвращает отсортированный массив. Если метод `sort()` вызывается без аргументов, то он сортирует элементы массива в алфавитном порядке (при необходимости временно преобразуя их в строки для выполнения сравнения):

```
var a = new Array("banana", "cherry", "apple");
a.sort();
var s = a.join(", ");     // s == "apple, banana, cherry"
```

Неопределенные элементы переносятся в конец массива.

Для сортировки в каком-либо ином порядке, отличном от алфавитного, можно передать методу `sort()` в качестве аргумента функцию сравнения. Эта функция устанавливает, какой из двух ее аргументов должен присутствовать раньше в отсортированном списке. Если первый аргумент должен предшествовать второму, функция сравнения возвращает отрицательное число. Если первый аргумент в отсортированном массиве должен следовать за вторым, то функция возвращает число, большее нуля. А если два значения эквивалентны (т. е. порядок их расположения не важен), функция сравнения возвращает 0. Поэтому, например, для сортировки элемента в числовом порядке, а не в алфавитном, можно сделать следующее:

```
var a = [33, 4, 1111, 222];
a.sort();                   // Алфавитный порядок: 1111, 222, 33, 4
a.sort(function(a,b) {      // Числовой порядок: 4, 33, 222, 1111
           return a-b;      // Возвращает значение < 0, 0, или > 0
       });                  // в зависимости от порядка сортировки a и b
```

Обратите внимание, насколько удобно использовать в этом фрагменте функциональный литерал. Функция сравнения вызывается только один раз, поэтому нет необходимости давать ей имя.

В качестве еще одного примера сортировки элементов массива вы можете выполнить алфавитную сортировку массива строк без учета регистра символов, передав методу функцию сравнения, преобразующую перед сравнением оба своих аргумента в нижний регистр (с помощью метода `toLowerCase()`). Можно придумать и другие функции сортировки, сортирующие числа в различном экзотическом порядке: обратном числовом, нечетные числа перед четными и т. д. Более интересные возможности, конечно же, открываются, когда сравниваемые элементы массива представляют собой объекты, а не простые типы, такие как числа и строки.

7.7.4. Метод concat()

Метод `Array.concat()` создает и возвращает новый массив, содержащий элементы исходного массива, для которого был вызван метод `concat()`, последовательно дополненный значениями всех аргументов, переданных методу `concat()`. Если какой-либо из этих аргументов сам является массивом, в результирующий массив добавляются его элементы. Однако обратите внимание, что рекурсивного разделения массивов из массивов не происходит. Вот несколько примеров:

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

7.7.5. Метод slice()

Метод `Array.slice()` возвращает фрагмент, или подмассив, указанного массива. Два аргумента метода определяют начало и конец возвращаемого фрагмента. Возвращаемый массив содержит элемент, номер которого указан в качестве первого аргумента, плюс все последующие элементы, вплоть до (но не включая) элемента, номер которого указан во втором аргументе. Если указан только один аргумент, возвращаемый массив содержит все элементы от начальной позиции до конца массива. Если какой-либо из аргументов отрицателен, он задает номер элемента массива относительно конца массива. Так, аргумент, равный -1, задает последний элемент массива, а аргумент, равный -3, – третий элемент массива с конца. Вот несколько примеров:

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

7.7.6. Метод splice()

Метод `Array.splice()` – это универсальный метод для вставки или удаления элементов массива. Он изменяет массив на месте, а не возвращает новый массив, как это делают методы `slice()` и `concat()`. Обратите внимание: `splice()` и `slice()` имеют очень похожие имена, но выполняют разные операции.

Метод `splice()` может удалять элементы из массива, вставлять новые элементы в массив или выполнять обе операции одновременно. Элементы массива при не-

обходимости смещаются, чтобы после вставки или удаления образовывалась непрерывная последовательность. Первый аргумент splice() задает позицию в массиве, с которой начинается вставка и/или удаление. Второй аргумент задает количество элементов, которые должны быть удалены (вырезаны) из массива. Если второй аргумент опущен, удаляются все элементы массива от начального до конца массива. Метод splice() возвращает массив удаленных элементов или (если ни один из элементов не был удален) пустой массив. Например:

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

Первые два аргумента splice() задают элементы массива, подлежащие удалению. За этими аргументами может следовать любое количество дополнительных аргументов, задающих элементы, которые будут вставлены в массив, начиная с позиции, заданной первым аргументом. Например:

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

Обратите внимание, что, в отличие от concat(), метод splice() не разбивает на отдельные элементы вставляемые аргументы-массивы. То есть если методу передается массив для вставки, он вставляет сам массив, а не элементы этого массива.

7.7.7. Методы push() и pop()

Методы push() и pop() позволяют работать с массивами как со стеками. Метод push() добавляет один или несколько новых элементов в конец массива и возвращает его новую длину. Метод pop() выполняет обратную операцию – удаляет последний элемент массива, уменьшает длину массива и возвращает удаленное им значение. Обратите внимание: оба эти метода изменяют массив на месте, а не создают его модифицированную копию. Комбинация push() и pop() позволяет в JavaScript с помощью массива реализовать стек с дисциплиной обслуживания «первым вошел – последним вышел». Например:

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

7.7.8. Методы unshift() и shift()

Методы unshift() и shift() ведут себя во многом так же, как push() и pop(), за исключением того, что они вставляют и удаляют элементы в начале массива, а не в его конце. Метод unshift() смещает существующие элементы в сторону больших индексов для освобождения места, добавляет элемент или элементы в начало массива и возвращает новую длину массива. Метод shift() удаляет и возвра-

щает первый элемент массива, смещая все последующие элементы вперед на одну позицию для занятия свободного места в начале массива. Например:

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

Обратите внимание на поведение метода `unshift()` при вызове с несколькими аргументами. Аргументы вставляются не по одному, а все сразу (как в случае с методом `splice()`). Это значит, что в результирующем массиве они будут следовать в том же порядке, в котором были указаны в списке аргументов. Будучи вставленными по одному, они бы расположились в обратном порядке.

7.7.9. Методы toString() и toLocaleString()

У массива, как и у любого другого объекта в JavaScript, имеется метод `toString()`. Для массива этот метод преобразует каждый из его элементов в строку (вызывая в случае необходимости методы `toString()` для элементов массива) и выводит список этих строк через запятую. Отметьте, что результат не включает квадратных скобок или каких-либо других разделителей вокруг значений массива. Например:

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

Обратите внимание: `toString()` возвращает ту же строку, что и метод `join()` при вызове его без аргументов.

Метод `toLocaleString()` – это локализованная версия `toString()`. Каждый элемент массива преобразуется в строку вызовом метода `toLocaleString()` элемента, а затем результирующие строки конкатенируются с использованием специфического для региона (и определенного реализацией) разделителя.

7.7.10. Дополнительные методы массивов

Броузер Firefox Mozilla 1.5 включает в себя новую версию JavaScript 1.6, в которую был добавлен набор дополнительных методов массивов, получивших название *дополнений к массивам* (*array extras*). Из наиболее примечательных можно назвать методы `indexOf()` и `lastIndexOf()`, позволяющие быстро отыскать в массиве заданное значение (описание аналогичного им метода `String.indexOf()` можно найти в третьей части книги). Кроме того, в состав набора входят еще несколько интересных методов: метод `forEach()` вызывает указанную функцию для каждого элемента в массиве; метод `map()` возвращает массив, полученный в результате передачи всех элементов массива указанной функции; метод `filter()` возвращает массив элементов, для которых заданная функция возвратила значение `true`.

На момент написания этих строк набор дополнительных методов массивов был доступен только в броузере Firefox и пока еще не является стандартом ни офици-

ально, ни де-факто. Здесь эти методы не описываются. Однако если вы предполагаете заниматься разработкой сценариев только для Firefox или в вашем распоряжении имеется библиотека, содержащая эти достаточно просто реализуемые методы, то подробное их описание можно найти на сайте *http://developer.mozilla.org*.

7.8. Объекты, подобные массивам

Массивы в JavaScript являются особенными, потому что их свойство `length` обладает особенным поведением:

- Значение этого свойства автоматически изменяется при добавлении к массиву новых элементов.
- Изменение этого свойства в программе приводит к усечению или увеличению массива.

Массивы в JavaScript являются экземплярами класса `Array` (`instanceof Array`), и для работы с ними могут использоваться методы этого класса.

Все эти характеристики являются уникальными для JavaScript-массивов, но они не главное, что определяет массив. Бывает полезно организовать работу с произвольным объектом, как со своего рода массивом – через свойство `length` и соответствующие неотрицательные целочисленные свойства. Такие объекты, «подобные массивам», иногда используются для решения практических задач. Хотя с ними нельзя работать через методы массивов или ожидать специфического поведения свойства `length`, можно организовать перебор свойств объекта теми же программными конструкциями, которые используются при работе с настоящими массивами. Оказывается, что значительное число алгоритмов для работы с массивами вполне пригодно для работы с объектами, подобными массивам. Пока вы не будете пытаться добавлять элементы в массив или изменять свойство `length`, вы вполне сможете обрабатывать объекты, подобные массивам, как обычные массивы.

В следующем фрагменте создается обычный объект и к нему добавляются дополнительные свойства, которые превращают его в объект, подобный массиву, после чего производится перебор «элементов» получившегося псевдомассива.

```
var a = {}; // Для начала создать обычный пустой объект

// Добавить свойства, которые сделают его похожим на массив
var i = 0;
while(i < 10) {
    a[i] = i * i;
    i++;
}
a.length = i;

// Теперь можно перебрать свойства объекта, как если бы это был настоящий массив
var total = 0;
for(var j = 0; j < a.length; j++)
    total += a[j];
```

Объект `Argument`, который описывается в разделе 8.2.2, является объектом, подобным массиву. В клиентском языке JavaScript такие объекты возвращают многие методы объектной модели документа (DOM), например метод `document.getElementsByTagName()`.

8

Функции

Функция – это блок программного кода на языке JavaScript, который определяется один раз и может *вызываться* многократно. Функции могут иметь *параметры*, или *аргументы*, – локальные переменные, значения которых определяются при вызове функции. Функции часто используют свои аргументы для вычисления *возвращаемого значения*, которое является значением выражения вызова функции. Если функция вызывается *в контексте* объекта, она называется *методом*, а сам объект передается ей в виде неявного аргумента. Вероятно, вы уже знакомы с концепцией функции, если встречались с такими понятиями, как *подпрограмма* и *процедура*.

В этой главе мы сосредоточимся на определении и вызове собственных JavaScript-функций. Важно помнить, что JavaScript поддерживает некоторое количество встроенных функций, таких как `eval()` и `parseInt()` или метод `sort()` класса `Array`. В клиентском языке JavaScript определяются другие функции, например `document.write()` и `alert()`. Встроенные JavaScript-функции применяются точно так же, как и функции, определенные пользователем. О функциях, упомянутых здесь, более подробно можно узнать в третьей и четвертой частях книги.

Функции и объекты в JavaScript тесно связаны между собой. По этой причине мы отложим обсуждение некоторых возможностей функций до главы 9.

8.1. Определение и вызов функций

Как мы видели в главе 6, самый распространенный способ определения функции – использование инструкции `function`. Она состоит из ключевого слова `function`, за которым следуют:

- имя функции;
- заключенный в круглые скобки необязательный список имен параметров, разделенных запятыми;
- JavaScript-инструкции, составляющие тело функции, заключенные в фигурные скобки.

В примере 8.1 показаны определения некоторых функций. Хотя эти функции короткие и простые, они содержат все перечисленные здесь элементы. Обратите внимание: в функциях может быть определено различное количество аргументов, функции также могут содержать или не содержать инструкцию `return`. Инструкция `return` была описана в главе 6; она прекращает выполнение функции и возвращает значение указанного в ней выражения (если оно есть) вызывающей стороне; при отсутствии выражения инструкция возвращает значение `undefined`. Если функция не содержит инструкцию `return`, она просто выполняет все инструкции в своем теле и возвращает неопределенное значение (`undefined`).

Пример 8.1. Определение JavaScript-функций

```
// Функция-обертка, иногда ее удобно использовать вместо document.write().
// В этой функции отсутствует инструкция return, поэтому она не возвращает значение.
function print(msg)
{
    document.write(msg, "<br>");
}

// Функция, вычисляющая и возвращающая расстояние между двумя точками.
function distance(x1, y1, x2, y2)
{
    var dx = x2 - x1;
    var dy = y2 - y1;
    return Math.sqrt(dx*dx + dy*dy);
}

// Рекурсивная функция (вызывающая сама себя), вычисляющая факториалы.
// Вспомните, что x! - это произведение x и всех положительных целых чисел, меньших x.
function factorial(x)
{
    if (x <= 1)
        return 1;
    return x * factorial(x-1);
}
```

Будучи один раз определенной, функция может вызываться с помощью оператора `()`, описанного в главе 5. Как вы помните, скобки указываются после имени функции, а необязательный список значений (или выражений) аргументов указывается в скобках через запятую (фактически перед круглыми скобками может указываться любое JavaScript-выражение, которое возвращает значение-функцию). Функции, определенные в примере 8.1, могут быть вызваны следующим образом:

```
print("Привет, " + name);
print("Добро пожаловать на мою домашнюю страницу!");
total_dist = distance(0,0,2,1) + distance(2,1,3,5);
print("Вероятность этого равна: " + factorial(5)/factorial(13));
```

При вызове функции вычисляются все выражения, указанные между скобками, и полученные значения используются в качестве аргументов функции. Эти значения присваиваются параметрам, имена которых перечислены в определении функции, и функция работает с ними, ссылаясь на эти параметры по указанным именам. Обратите внимание: эти переменные-параметры определены, только

пока выполняется функция; они не сохраняются после завершения ее работы (за одним важным исключением, которое описывается в разделе 8.8).

JavaScript – язык с нестрогой типизацией, поэтому тип параметров функций указывать не требуется, и JavaScript не проверяет, соответствует ли тип данных требованиям функции. Если тип аргумента важен, вы можете проверить его самостоятельно с помощью оператора `typeof`. Кроме того, JavaScript не проверяет, правильное ли количество параметров передано функции. Если аргументов больше, чем требуется функции, то дополнительные значения просто игнорируются. Если аргументов меньше, то отсутствующим присваивается значение `undefined`. Некоторые функции написаны так, что могут достаточно терпимо относиться к нехватке аргументов, другие ведут себя некорректно, если получают меньшее число аргументов, чем предполагалось. Далее в этой главе мы познакомимся с приемом, позволяющим проверить, правильное ли количество аргументов передано в функцию, и организовать доступ к этим аргументам по их порядковым номерам в списке аргументов, а не по именам.

Обратите внимание: в функции `print()` из примера 8.1 нет инструкции `return`, поэтому она всегда возвращает значение `undefined`, и использовать ее в качестве части более сложного выражения не имеет смысла. А функции `distance()` и `factorial()` могут вызываться в более сложных выражениях, что было показано в предыдущих примерах.

8.1.1. Вложенные функции

В JavaScript допускается вложение определений функций в другие функции. Например:

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

Вложенные функции могут определяться только в коде функций верхнего уровня. Это значит, что определения функций не могут находиться, например, внутри циклов или условных инструкций.[1] Обратите внимание: эти ограничения распространяются только на объявления функций с помощью инструкции `function`. Функциональные литералы (которые описываются в следующем разделе) могут присутствовать внутри любых выражений.

8.1.2. Функциональные литералы

JavaScript позволяет определять функции в виде *функциональных литералов*. Как говорилось в главе 3, функциональный литерал – это выражение, определяющее неименованную функцию. Синтаксис функционального литерала во многом напоминает синтаксис инструкции `function`, за исключением того, что он ис-

[1] Различные реализации JavaScript могут иметь менее строгие требования к определениям функций, чем указано в стандарте. Например, реализации Netscape JavaScript 1.5 допускают наличие «условных определений функций» внутри инструкций `if`.

пользуется как выражение, а не как инструкция, и ему не требуется имя функции. Следующие две строки кода определяют две более или менее идентичные функции с помощью инструкции function и функционального литерала:

```
function f(x) { return x*x; }        // инструкция function
var f = function(x) { return x*x; }; // функциональный литерал
```

Функциональные литералы создают неименованные функции, но синтаксис допускает указание имени функции, что может пригодиться при написании рекурсивных функций, вызывающих самих себя. Например:

```
var f = function fact(x) { if (x <= 1) return 1; else return x*fact(x -1); };
```

Эта строка кода определяет неименованную функцию и сохраняет ссылку на нее в переменной f. Она на самом деле *не создает* функцию с именем fact, но позволяет телу функции ссылаться с помощью этого имени на саму себя. Заметим, однако, что именованные функциональные литералы до версии JavaScript 1.5 работали не вполне корректно.

Функциональные литералы создаются JavaScript-выражениями, а не инструкциями, и потому могут использоваться более гибко. Это особенно подходит для функций, которые вызываются только один раз и не должны иметь имени. Например, функция, определенная с помощью выражения функционального литерала, может быть сохранена в переменной, передана другой функции или даже вызвана непосредственно:

```
f[0] = function(x) { return x*x; };  // Определить и сохранить функцию в переменной
a.sort(function(a,b){return a-b;});  // Определить функцию; передать ее другой функции
var tensquared = (function(x) {return x*x;})(10); // Определить и вызывать
```

8.1.3. Именование функций

В качестве имени функции может использоваться любой допустимый JavaScript-идентификатор. Старайтесь выбирать функциям достаточно описательные, но не длинные имена. Искусство сохранения баланса между краткостью и информативностью приходит с опытом. Правильно подобранные имена функций могут существенно повысить удобочитаемость (а значит, и простоту сопровождения) ваших программ.

Чаще всего в качестве имен функций выбираются глаголы или фразы, начинающиеся с глаголов. По общепринятому соглашению имена функций начинаются со строчной буквы. Если имя состоит из нескольких слов, в соответствии с одним из соглашений они отделяются друг от друга символом подчеркивания, примерно так: like_this(), по другому соглашению все слова, кроме первого, начинаются с прописной буквы, примерно так: likeThis(). Имена функций, которые, как предполагается, реализуют внутреннюю, скрытую от посторонних глаз функциональность, иногда начинаются с символа подчеркивания.

В некоторых стилях программирования или в четко определенных программных платформах бывает полезно давать наиболее часто используемым функциям очень короткие имена. Примером может служить платформа Prototype клиентского языка JavaScript (*http://prototype.conio.net*), в которую весьма элегантно вписалась функция с именем $() (да-да, просто знак доллара) в качестве замены сложному для набора с клавиатуры имени document.getElementById(). (В главе 2

уже говорилось, что в идентификаторах JavaScript допускается использовать знаки доллара и подчеркивания.)

8.2. Аргументы функций

Функции в JavaScript могут вызываться с произвольным числом аргументов независимо от того, сколько аргументов указано в определении именованной функции. Поскольку функции являются слабо типизированными, отсутствует возможность задавать типы входных аргументов, в связи с чем считается допустимым передавать значения любых типов любым функциям. Все эти вопросы обсуждаются в следующих подразделах.

8.2.1. Необязательные аргументы

Когда функция вызывается с меньшим количеством аргументов, чем описывается в определении, недостающие аргументы получают значение `undefined`. Иногда бывает удобным учесть необязательность некоторых аргументов – тех, которые могут быть опущены при вызове функции. В этом случае желательно предусмотреть возможность присваивания по умолчанию достаточно разумных значений аргументам, которые были опущены (или переданы со значением `null`). Например:

```
// Добавить в массив a перечислимые имена свойств объекта o и вернуть массив a.
// Если массив a не указан или равен null, создать и вернуть новый массив a
function copyPropertyNamesToArray(o, /* необязательный */ a) {
    if (!a) a = []; // Если массив не определен или получено
                    // значение null, создать новый пустой массив a
    for(var property in o) a.push(property);
    return a;
}
```

Когда функция определена таким образом, появляются более широкие возможности обращения к ней:

```
// Получить имена свойств объектов o и p
var a = copyPropertyNamesToArray(o); // Получить свойства объекта o
                                     // в виде нового массива
copyPropertyNamesToArray(p,a);       // добавить к массиву свойства объекта p
```

Вместо инструкции `if` в первой строке этой функции можно использовать оператор `||` следующим образом:

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

В главе 5 уже говорилось, что оператор `||` возвращает первый аргумент, если он равен `true` или преобразуется в логическое значение `true`. В противном случае возвращается второй аргумент. В данном случае он вернет `a`, если переменная `a` определена и не содержит значение `null` даже в том случае, если `a` – это пустой массив. В противном случае он вернет новый пустой массив.

Обратите внимание: при объявлении функций необязательные аргументы должны завершать список аргументов, чтобы их можно было опустить. Программист, который будет писать обращение к вашей функции, не сможет передать второй аргумент и при этом опустить первый. В этом случае он вынужден будет явно передать в первом аргументе значение `undefined` или `null`.

8.2.2. Списки аргументов переменной длины: объект Arguments

В теле функции идентификатор arguments всегда имеет особый смысл; arguments – это специальное свойство объекта вызова, ссылающееся на объект, известный как объект Arguments. Объект Arguments – это нечто вроде массива (см. раздел 7.8), позволяющего извлекать переданные функции значения по номеру, а не по имени. Объект Arguments также определяет дополнительное свойство callee, которое описано в следующем разделе.

Хотя JavaScript-функция определяется с фиксированным количеством именованных аргументов, при вызове ей может быть передано любое их число. Объект Arguments обеспечивает полный доступ к значениям аргументов, даже если у некоторых из них нет имени. Предположим, что была определена функция f, которая требует один аргумент, x. Если вызвать эту функцию с двумя аргументами, то первый будет доступен в функции по имени параметра x или как arguments[0]. Второй аргумент доступен только как arguments[1]. Кроме того, как и у всех массивов, у arguments имеется свойство length, указывающее на количество содержащихся в массиве элементов. То есть в теле функции f, вызываемой с двумя аргументами, arguments.length имеет значение 2.

Объект arguments может использоваться с самыми разными целями. Следующий пример показывает, как с его помощью проверить, была ли функция вызвана с правильным числом аргументов, – ведь JavaScript этого за вас не сделает:

```
function f(x, y, z)
{
    // Сначала проверяется, правильное ли количество аргументов передано
    if (arguments.length != 3) {
        throw new Error("функция f вызвана с " + arguments.length +
                        "аргументами, а требуется 3.");
    }
    // А теперь сам код функции...
}
```

Объект arguments иллюстрирует важную возможность JavaScript-функций: они могут быть написаны таким образом, чтобы работать с любым количеством аргументов. Далее приводится пример, показывающий, как можно написать простую функцию max(), принимающую любое число аргументов и возвращающую значение самого большого из них (аналогично ведет себя встроенная функция Math.max()):

```
function max(/*...*/)
{
    var m = Number.NEGATIVE_INFINITY;
    // Цикл по всем аргументам, поиск и сохранение наибольшего из них
    for(var i = 0; i < arguments.length; i++)
        if (arguments[i] > m) m = arguments[i];
    // Возвращаем максимальный
    return m;
}

var largest = max(1, 10, 100, 2, 3, 1000, 4, 5, 10000, 6);
```

Функции, подобные этой и способные принимать произвольное число аргументов, называются *функциями с переменным числом аргументов* (*variadic functions, variable arity functions* или *varargs functions*). Этот термин возник вместе с появлением языка программирования C.

Обратите внимание: функции с переменным числом аргументов не должны допускать возможность вызова с пустым списком аргументов. Будет вполне разумным использовать объект `arguments[]` при написании функции, ожидающей получить фиксированное число обязательных именованных аргументов, за которыми может следовать произвольное число необязательных неименованных аргументов.

Не следует забывать, что `arguments` фактически не является массивом – это объект `Arguments`. В каждом объекте `Arguments` имеются пронумерованные элементы массива и свойство `length`, но с технической точки зрения – это не массив. Лучше рассматривать его как объект, имеющий некоторые пронумерованные свойства. Спецификация ECMAScript не требует от объекта `Arguments` реализации какого-либо специфического для массива поведения. Хотя, например, допускается присваивать значение свойству `arguments.length`, ECMAScript не требует этого для реального изменения числа элементов массива, определенных в объекте. (Особое поведение свойства `length` для настоящих объектов `Array` описывается в разделе 7.6.3.)

У объекта `Arguments` есть одна очень необычная особенность. Когда у функции имеются именованные аргументы, элементы массива объекта `Arguments` являются синонимами локальных переменных, содержащих аргументы функции. Массив `arguments[]` и именованные аргументы – это два разных средства обращения к одной переменной. Изменение значения аргумента через имя аргумента меняет значение, извлекаемое через массив `arguments[]`. Изменение значения аргумента через массив `arguments[]` меняет значение, извлекаемое по имени аргумента. Например:

```
function f(x) {
    print(x);             // Выводит начальное значение аргумента
    arguments[0] = null; // Изменяя элементы массива, мы также изменяем x
    print(x);             // Теперь выводит "null"
}
```

Определенно, это не совсем то поведение, которое можно было бы ожидать от настоящего массива. В этом случае `arguments[0]` и `x` могли бы ссылаться на одно и то же значение, но изменение одной ссылки не должно оказывать влияния на другую.

Наконец, следует учитывать, что `arguments` – это всего лишь обычный JavaScript-идентификатор, а не зарезервированное слово. Если функция определит аргумент или локальную переменную с таким же именем, объект `Arguments` станет недоступным. По этой причине следует считать слово `arguments` зарезервированным и стараться избегать создавать переменные с таким именем.

8.2.2.1. Свойство callee

Помимо элементов своего массива объект `Arguments` определяет свойство `callee`, ссылающееся на исполняемую в данный момент функцию. Его можно использовать, например, для рекурсивного вызова неименованных функций. Вот пример неименованного функционального литерала, вычисляющего факториал:

```
function(x) {
    if (x <= 1) return 1;
    return x * arguments.callee(x - 1);
}
```

8.2.3. Использование свойств объекта в качестве аргументов

Когда функция имеет более трех аргументов, становится трудно запоминать правильный порядок их следования. Чтобы предотвратить ошибки и избавить программиста от необходимости заглядывать в справочное руководство всякий раз, когда он намеревается вставить в программу вызов такой функции, можно предусмотреть возможность передачи аргументов в виде пар «имя–значение» в произвольном порядке. Чтобы реализовать такую возможность, при определении функции следует учесть передачу объекта в качестве единственного аргумента. Благодаря такому стилю, пользователи функции смогут передавать функции литерал объекта, в котором будут определяться необходимые пары «имя–значение». В следующем фрагменте приводится пример такой функции, а также демонстрируется возможность определения значений по умолчанию для опущенных аргументов:

```
// Скопировать length элементов из массива from в массив to.
// Копирование начинается с элемента from_start в массиве from
// и выполняется в элементы, начиная с to_start в массиве to.
// Запомнить порядок следования аргументов такой функции довольно сложно.
function arraycopy(/* array */ from, /* index */ from_start,
                   /* array */ to,   /* index */ to_start,
                   /* integer */ length)
{
    // здесь находится реализация функции
}

// Эта версия функции чуть менее эффективная, но не требует
// запоминать порядок следования аргументов, а аргументы from_start
// и to_start по умолчанию принимают значение 0.
function easycopy(args) {
    arraycopy(args.from,
              args.from_start || 0, // Обратите внимание, как назначаются
                                    // значения по умолчанию
              args.to,
              args.to_start || 0,
              args.length);
}

// Далее следует пример вызова функции easycopy():
var a = [1,2,3,4];
var b = new Array(4);
easycopy({from: a, to: b, length: 4});
```

8.2.4. Типы аргументов

Поскольку JavaScript является слабо типизированным языком, аргументы методы объявляются без указания их типов, а во время передачи значений функ-

ции не производится никакой проверки их типов. Вы можете сделать свой программный код самодокументируемым, выбирая описательные имена для аргументов функций и включая указание на тип в комментарии, как это сделано в только что рассмотренном примере с функцией `arraycopy()`. Для необязательных аргументов можно добавлять в комментарий слово «необязательный» («optional»). А если метод предусматривает возможность принимать произвольное число аргументов, можно использовать многоточие:

```
function max(/* число... */) { /* тело функции */ }
```

Как отмечалось в главе 3, в случае необходимости JavaScript выполняет преобразование типов. Таким образом, если вы создали функцию, которая ожидает получить строковый аргумент, а затем вызываете ее с аргументом какого-нибудь другого типа, значение аргумента просто преобразуется в строку, когда функция пытается обратиться к нему как к строке. В строку может быть преобразован любой элементарный тип, и все объекты имеют методы `toString()` (правда, не всегда полезные), тем самым устраняется вероятность появления ошибки.

Однако такой подход может использоваться не всегда. Рассмотрим еще раз метод `arraycopy()`, продемонстрированный ранее. Он ожидает получить массив в первом аргументе. Любое обращение к функции окажется неудачным, если первым аргументом будет не массив (или, возможно, объект, подобный массиву). Если функция должна вызываться чаще, чем один-два раза, следует добавить в нее проверку соответствия типов аргументов. При передаче аргументов ошибочных типов должно генерироваться исключение, которое зафиксирует этот факт. Гораздо лучше сразу же прервать вызов функции в случае передачи аргументов ошибочных типов, чем продолжать исполнение, которое потерпит неудачу, когда, например, функция попытается получить доступ к элементу массива с помощью числового аргумента, как в следующем фрагменте:

```
// Возвращает сумму элементов массива (или объекта, подобного массиву) a.
// Все элементы массива должны быть числовыми, при этом значения null
// и undefined игнорируются.
function sum(a) {
    if ((a instanceof Array) || // если это массив
        (a && typeof a == "object" && "length" in a)) { // или объект, подобный массиву
        var total = 0;
        for(var i = 0; i < a.length; i++) {
            var element = a[i];
            if (!element) continue; // игнорировать значения null и undefined
            if (typeof element == "number") total += element;
            else throw new Error("sum(): все элементы должны быть числами");
        }
        return total;
    }
    else throw new Error("sum(): аргумент должен быть массивом");
}
```

Метод `sum()` весьма строго относится к проверке типов входных аргументов и генерирует исключения с достаточно информативными сообщениями, если типы входных аргументов не соответствуют ожидаемым. Тем не менее он остается достаточно гибким, обслуживая наряду с настоящими массивами объекты, подобные массивам, и игнорируя элементы, имеющие значения `null` и `undefined`.

JavaScript – чрезвычайно гибкий и к тому же слабо типизированный язык, благодаря чему можно писать функции, которые достаточно терпимо относятся к количеству и типам входных аргументов. Далее приводится метод `flexsum()`, реализующий такой подход (и, вероятно, являющийся примером другой крайности). Например, он принимает любое число входных аргументов и рекурсивно обрабатывает те из них, которые являются массивами. Вследствие этого он может принимать переменное число аргументов или массив в виде аргумента. Кроме того, он прилагает максимум усилий, чтобы преобразовать нечисловые аргументы в числа, прежде чем сгенерировать исключение:

```
function flexisum(a) {
    var total = 0;
    for(var i = 0; i < arguments.length; i++) {
        var element = arguments[i];
        if (!element) continue; // Игнорировать значения null и undefined

        // Попытаться преобразовать аргумент в число n исходя из типа аргумента
        var n;
        switch(typeof element) {
        case "number":
            n = element;                // Преобразование не требуется
            break;
        case "object":
            if (element instanceof Array) // Рекурсивный обход массива
                n = flexisum.apply(this, element);
            else n = element.valueOf( );  // Для других объектов вызвать valueOf
            break;
        case "function":
            n = element( );               // Попытаться вызвать функцию
            break;
        case "string":
            n = parseFloat(element);      // Попытаться преобразовать строку
            break;
        case "boolean":
            n = NaN; // Логические значения преобразовать невозможно
            break;
        }
        // Если было получено нормально число - добавить его к сумме.
        if (typeof n == "number" && !isNaN(n)) total += n;
        // В противном случае сгенерировать исключение
        else
            throw new Error("sum(): ошибка преобразования  " + element + " в число");
    }
    return total;
}
```

8.3. Функции как данные

Самые важные особенности функций заключаются в том, что они могут определяться и вызываться, что было показано в предыдущем разделе. Определение и вызов функций – это синтаксические средства JavaScript и большинства других языков программирования. Однако в JavaScript функции – это не только

синтаксические конструкции, но и данные, а это означает, что они могут присваиваться переменным, храниться в свойствах объектов или элементах массивов, передаваться как аргументы функциями и т. д.[1]

Чтобы понять, как в JavaScript функции могут быть одновременно синтаксическими конструкциями и данными, рассмотрим следующее определение функции:

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

Это определение создает новый объект функции и присваивает его переменной `square`. Имя функции действительно нематериально – это просто имя переменной, содержащей функцию. Функция может быть присвоена другой переменной, и при этом работать так же, как и раньше:

```
var a = square(4); // a содержит число 16
var b = square;    // b теперь ссылается на ту же функцию, что и square
var c = b(5);      // c содержит число 25
```

Функции могут быть также присвоены не только глобальным переменным, но и свойствам объектов. В этом случае их называют методами:

```
var o = new Object;
o.square = function(x) { return x*x; }; // функциональный литерал
y = o.square(16);                       // y равно 256
```

У функций даже не обязательно должны быть имена, например в случае присваивании их элементам массива:

```
var a = new Array(3);
a[0] = function(x) { return x*x; }
a[1] = 20;
a[2] = a[0](a[1]); // a[2] содержит 400
```

Синтаксис вызова функции в последнем примере выглядит необычно, однако это вполне допустимый вариант применения оператора `()` в JavaScript!

В примере 8.2 подробно показано, что можно делать, когда функции выступают в качестве данных. Этот пример демонстрирует, каким образом функции могут передаваться другим функциям. Хотя пример может показаться вам несколько сложным, комментарии объясняют, что происходит, и он вполне достоин тщательного изучения.

Пример 8.2. Использование функций как данных

```
// Здесь определяются несколько простых функций
function add(x,y) { return x + y; }
function subtract(x,y) { return x - y; }
function multiply(x,y) { return x * y; }
function divide(x,y) { return x / y; }

// Эта функция принимает одну из вышеприведенных функций
// в качестве аргумента и вызывает ее для двух операндов
```

[1] Это может показаться не столь интересным, если вы незнакомы с такими языками, как Java, в которых функции являются частью программы, но не могут программой управляться.

```
function operate(operator, operand1, operand2)
{
    return operator(operand1, operand2);
}

// Вот так можно вызвать эту функцию для вычисления значения выражения (2+3) + (4*5):
var i = operate(add, operate(add, 2, 3), operate(multiply, 4, 5));

// Ради примера, мы реализуем эти функции снова, на этот раз с помощью
// функциональных литералов внутри литерала объекта.
var operators = {
    add:      function(x,y) { return x+y; },
    subtract: function(x,y) { return x-y; },
    multiply: function(x,y) { return x*y; },
    divide:   function(x,y) { return x/y; },
    pow:      Math.pow // Для предопределенных функций это тоже работает
};

// Эта функция принимает имя оператора, отыскивает оператор в объекте,
// а затем вызывает его для переданных операндов. Обратите внимание
// на синтаксис вызова функции оператора.
function operate2(op_name, operand1, operand2)
{
    if (typeof operators[op_name] == "function")
        return operators[op_name](operand1, operand2);
    else throw "неизвестный оператор";
}

// Вот так мы можем вызвать эту функцию для вычисления значения
// ("hello" + " " + "world"):
var j = operate2("add", "hello", operate2("add", " ", "world"))
// Используем предопределенную функцию Math.pow():
var k = operate2("pow", 10, 2)
```

Если предыдущий пример не убедил вас в удобстве передачи функций в качестве аргументов другим функциям и других способов использования функций как значений, обратите внимание на функцию `Array.sort()`. Она сортирует элементы массива. Существует много возможных порядков сортировки (числовой, алфавитный, по датам, по возрастанию, по убыванию и т. д.), поэтому функция `sort()` принимает в качестве необязательного аргумента другую функцию, которая сообщает о том, как выполнять сортировку. Эта функция делает простую работу – получает два элемента массива, сравнивает их, а затем возвращает результат, указывающий, какой из элементов должен быть первым. Этот аргумент функции делает метод `Array.sort()` совершенно универсальным и бесконечно гибким – он может сортировать любой тип данных в любом мыслимом порядке! (Пример использования функции `Array.sort()` вы найдете в разделе 7.7.3.)

8.4. Функции как методы

Метод – это не что иное, как функция, которая хранится в свойстве объекта и вызывается посредством этого объекта. Не забывайте, что функции – это всего лишь значения данных, а в именах, с которыми они определены, нет ничего необычного. Поэтому функции могут присваиваться любым переменным, равно

как и свойствам объектов. Например, если имеется функция f и объект o, вполне возможно так определить метод с именем m:

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

Определив в объекте o метод m(), к нему можно обратиться следующим образом:

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

Или, если метод m() ожидает получить два аргумента:

```
o.m(x, x+2);
```

Методы обладают одним очень важным свойством: объект, посредством которого вызывается метод, становится значением ключевого слова this в теле метода. То есть когда вызывается метод o.m(), в теле метода можно получить доступ к объекту o с помощью ключевого слова this. Это утверждение демонстрируется в следующем примере:

```
var calculator = {          // Литерал объекта
    operand1: 1,
    operand2: 1,
    compute: function( ) {
        this.result = this.operand1 + this.operand2;
    }
};
calculator.compute();       // Сколько будет 1+1?
print(calculator.result);   // Выводит результат
```

Ключевое слово this играет очень важную роль. Любая функция, вызываемая как метод, получает в свое распоряжение дополнительный неявный аргумент – объект, посредством которого эта функция была вызвана. Как правило, методы выполняют некоторые действия над этим объектом, таким образом, синтаксис вызова методов наглядно отражает тот факт, что функция оперирует объектом. Сравните следующие две строки программы:

```
rect.setSize(width, height);
setRectSize(rect, width, height);
```

Гипотетически функции, вызывающиеся в этих двух строках, могут производить абсолютно идентичные действия над объектом rect (гипотетическим), но синтаксис вызова метода в первой строке более наглядно демонстрирует, что в центре внимания находится объект rect. (Если первая строка не показалась вам более естественной, это означает, что у вас еще нет опыта объектно-ориентированного программирования.)

Когда функция вызывается как функция, а не как метод, ключевое слово this ссылается на глобальный объект. Самое странное, что это верно даже для функций (если они вызываются как функции), вложенных в методы, которые в свою очередь вызываются как методы. Ключевое слово this имеет одно значение в объемлющей функции и ссылается на глобальный объект в теле вложенной функции (что интуитивно совершенно не очевидно).

Обратите внимание: this – это именно ключевое слово, а не имя переменной или свойства. Синтаксис JavaScript не допускает возможность присваивания значений элементу this.

8.5. Функция-конструктор

Конструктор – это функция, которая выполняет инициализацию свойств объекта и предназначена для использования совместно с инструкцией `new`. Подробное описание конструкторов приводится в главе 9. Однако коротко можно отметить, что инструкция `new` создает новый объект `Function`, после чего вызывает функцию-конструктор, передавая ей вновь созданный объект в качестве значения ключевого слова `this`.

8.6. Свойства и методы функций

Мы видели, что в JavaScript-программах функции могут использоваться как значения. Инструкция `typeof` возвращает для функций строку `"function"`, однако в действительности функции в JavaScript – это особого рода объекты. А раз функции являются объектами, то они имеют свойства и методы – так же как, например, объекты `RegExp` и `Date`.

8.6.1. Свойство length

Как мы видели, в теле функции свойство `length` массива `arguments` определяет количество аргументов, переданных этой функции. Однако свойство `length` самой функции имеет другой смысл. Это доступное только для чтения свойство возвращает количество аргументов, которое функция *ожидает* получить, т. е. объявленных в ее списке параметров. Вспомним, что функция может вызываться с любым количеством аргументов, которые могут быть извлечены через массив `arguments`, независимо от того, сколько их объявлено. Свойство `length` объекта `Function` в точности определяет, сколько объявленных параметров имеется у функции. Обратите внимание: в отличие от свойства `arguments.length`, указанное свойство `length` доступно как внутри, так и вне тела функции.

В следующем фрагменте определяется функция с именем `check()`, получающая массив аргументов от другой функции. Она сравнивает свойство `arguments.length` со свойством `Function.length` (доступным как `arguments.callee.length`) и проверяет, передано ли функции столько аргументов, сколько она ожидает. Если это не так, генерируется исключение. За функцией `check()` следует тестовая функция `f()`, демонстрирующая порядок вызова функции `check()`:

```
function check(args) {
    var actual = args.length;          // Фактическое число аргументов
    var expected = args.callee.length; // Ожидаемое число аргументов
    if (actual != expected) {       // Если числа не совпадают, генерируется исключение
        throw new Error("неверное число аргументов: ожидается: " +
        expected + "; фактически передано " + actual);
    }
}

function f(x, y, z) {
    // Проверяем, соответствует ли ожидаемому фактическое количество
    // аргументов. Если не соответствует, генерируем исключение
    check(arguments);
    // Теперь выполняем оставшуюся часть функции обычным образом
```

```
    return x + y + z;
}
```

8.6.2. Свойство prototype

Любая функция имеет свойство `prototype`, ссылающееся на предопределенный *объект-прототип*. Этот объект, который вступает в игру, когда функция используется в качестве конструктора с оператором `new`, играет важную роль в процессе определения новых типов объектов. Мы подробно изучим свойство `prototype` в главе 9.

8.6.3. Определение собственных свойств функций

Когда функции требуется переменная, значение которой должно сохраняться между ее вызовами, часто оказывается удобным использовать свойство объекта `Function`, позволяющее не занимать пространство имен определениями глобальных переменных. Предположим, что надо написать функцию, возвращающую уникальный идентификатор при каждом своем вызове. Функция никогда не должна возвращать одно и то же значение дважды. Чтобы обеспечить это, функция запоминает последнее возвращенное значение, и эта информация сохраняется между ее вызовами. Хотя указанная информация может храниться в глобальной переменной, в этом нет никакой необходимости, и лучше сохранять ее в свойстве объекта `Function`, т. к. эта информация используется только самой функцией. Вот пример функции, которая возвращает уникальное целое значение при каждом вызове:

```
// Создаем и инициализируем "статическую" переменную.
// Объявления функций обрабатываются до исполнения кода,
// поэтому мы действительно можем выполнить это присваивание
// до объявления функции
uniqueInteger.counter = 0;

// Сама функция. Она возвращает разные значения при каждом
// вызове и использует собственное "статическое" свойство
// для отслеживания последнего возвращенного значения.
function uniqueInteger() {
    // Наращиваем и возвращаем значение "статической" переменной
    return uniqueInteger.counter++;
}
```

8.6.4. Методы apply и call()

В ECMAScript есть два метода, определенные для всех функций, — `call()` и `apply()`. Эти методы позволяют вызывать функцию так, будто она является методом некоторого объекта. Первый аргумент методов `call()` и `apply()` — это объект, для которого выполняется функция; этот аргумент становится значением ключевого слова `this` в теле функции. Все оставшиеся аргументы `call()` — это значения, передаваемые вызываемой функции. Так, чтобы передать функции `f()` два числа и вызвать ее как метод объекта `o`, можно использовать следующий прием:

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

Это аналогично следующим строкам программы:

```
o.m = f;
o.m(1,2);
delete o.m;
```

Метод `apply()` похож на метод `call()`, за исключением того, что передаваемые функции аргументы задаются в виде массива:

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

Например, чтобы найти наибольшее число в массиве чисел, можно вызвать метод `apply()` для передачи элементов массива функции `Math.max()`:

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

8.7. Практические примеры функций

В этом разделе приводятся примеры нескольких функций для работы с объектами и массивами, имеющие практическую ценность. Пример 8.3 содержит функции для работы с объектами.

Пример 8.3. Функции для работы с объектами

```
// Возвращает массив, содержащий имена перечислимых свойств объекта "o"
function getPropertyNames(/* объект */o) {
    var r = [];
    for(name in o) r.push(name);
    return r;
}

// Копирует перечислимые свойства объекта "from" в объект "to".
// Если аргумент "to" равен null, создается новый объект.
// Функция возвращает объект "to" или вновь созданный объект.
function copyProperties(/* объект */ from, /* необязательный объект */ to) {
    if (!to) to = {};
    for(p in from) to[p] = from[p];
    return to;
}

// Копирует перечислимые свойства объекта "from" в объект "to",
// но только те, которые еще не определены в объекте "to".
// Это может оказаться необходимым, например, когда объект "from" содержит
// значения по умолчанию, которые необходимо скопировать в свойства,
// если они еще не были определены в объекте "to".
function copyUndefinedProperties(/* объект */ from, /* объект */ to) {
    for(p in from) {
        if (!p in to) to[p] = from[p];
    }
}
```

В следующем примере 8.4 приводятся функции для работы с массивами.

Пример 8.4. Функции для работы с массивами

```
// Передать каждый элемент массива "a" заданной функции проверки.
// Вернуть массив, хранящий только те элементы, для которых
```

```
// функция проверки вернула значение true
function filterArray(/* массив */ a, /* функция проверки */ predicate) {
    var results = [];
    var length = a.length; // На случай, если функция проверки изменит свойство length!
    for(var i = 0; i < length; i++) {
        var element = a[i];
        if (predicate(element)) results.push(element);
    }
    return results;
}

// Возвращает массив значений, которые являются результатом передачи
// каждого элемента массива функции "f"
function mapArray(/* массив */a, /* функция */ f) {
    var r = [];             // результаты
    var length = a.length; // На случай, если f изменит свойство length!
    for(var i = 0; i < length; i++) r[i] = f(a[i]);
    return r;
}
```

Наконец, функции из примера 8.5 предназначены для работы с функциями. Фактически они используют и возвращают вложенные функции. Вложенные функции возвращаются способом, получившим некогда название «замыкание». Замыкания, которые могут оказаться сложными для понимания, рассматриваются в следующем разделе.

Пример 8.5. Функции для работы с функциями

```
// Возвращает самостоятельную функцию, которая в свою очередь вызывает
// функцию "f" как метод объекта "o". Эта функция может использоваться,
// когда возникает необходимость передать в функцию метод.
// Если не связать метод с объектом, ассоциация будет утрачена, и метод,
// переданный функции, будет вызван как обычная функция.
function bindMethod(/* объект */ o, /* функция */ f) {
    return function() { return f.apply(o, arguments) }
}

// Возвращает самостоятельную функцию, которая в свою очередь вызывает
// функцию "f" с заданными аргументами и добавляет дополнительные
// аргументы, передаваемые возвращаемой функции.
// (Этот прием иногда называется "currying".)
function bindArguments(/* функция */ f /*, начальные аргументы... */) {
    var boundArgs = arguments;
    return function() {
        // Создать массив аргументов. Он будет начинаться с аргументов,
        // определенных ранее, и заканчиваться аргументами, переданными сейчас
        var args = [];
        for(var i = 1; i < boundArgs.length; i++) args.push(boundArgs[i]);
        for(var i = 0; i < arguments.length; i++) args.push(arguments[i]);

        // Теперь вызвать функцию с новым списком аргументов
        return f.apply(this, args);
    }
}
```

8.8. Область видимости функций и замыкания

Как говорилось в главе 4, в JavaScript тело функции исполняется в локальной области видимости, которая отличается от глобальной. В этом разделе рассматриваются вопросы, связанные с областью видимости, включая замыкания.[1]

8.8.1. Лексическая область видимости

Функции в JavaScript имеют не динамическую, а лексическую область видимости. Это означает, что они исполняются в области видимости, которая была создана на момент определения функции, а не на момент ее исполнения. В момент определения функции текущая цепочка областей видимости сохраняется и становится частью внутреннего состояния функции. На верхнем уровне область видимости просто состоит из глобального объекта, и о лексической области видимости говорить не приходится. Однако когда объявляется вложенная функция, ее цепочка областей видимости включает объемлющую функцию. Это означает, что вложенная функция обладает возможностью доступа ко всем аргументам и локальным переменным объемлющей функции.

Обратите внимание: несмотря на то, что цепочка областей видимости фиксируется в момент определения функции, перечень свойств, определенных в этой цепочке, не фиксируется. Цепочка областей видимости подвержена изменениям, и функция может обращаться ко всем элементам, существующим на момент исполнения.

8.8.2. Объект вызова

Когда интерпретатор JavaScript вызывает функцию, в первую очередь он устанавливает область видимости в соответствии с цепочкой областей видимости, которая действовала на момент определения функции. Затем он добавляет в начало цепочки новый объект, известный как *объект вызова* – в спецификации ECMAScript используется термин *объект активации* (*activation object*). В объект вызова добавляется свойство `arguments`, которое ссылается на объект `Arguments` функции. После этого в объект вызова добавляются именованные аргументы функции. Любые локальные переменные, объявленные с помощью инструкции `var`, также определяются внутри объекта. Поскольку данный объект вызова располагается в начале цепочки областей видимости, все локальные переменные, аргументы функции и объект `Arguments` становятся видимыми из тела функции. Помимо всего прочего это означает, что все одноименные свойства оказываются за пределами области видимости.

Обратите внимание: `this`, в отличие от `arguments`, – это не свойство объекта вызова, а ключевое слово.

8.8.3. Объект вызова как пространство имен

Иногда бывает удобно создать функцию только ради того, чтобы получить объект вызова, который действует как временное пространство имен, где можно оп-

[1] Этот раздел содержит материал повышенной сложности, который при первом прочтении можно пропустить.

ределять переменные и свойства, не беспокоясь о возможных конфликтах с глобальным пространством имен. Предположим, для примера, что имеется файл с программным кодом на языке JavaScript, который необходимо использовать в разных JavaScript-программах (или, если дело касается клиентского языка JavaScript, на разных веб-страницах). Допустим, что в этом коде, как и любом другом, определяются переменные, предназначенные для хранения промежуточных результатов вычислений. Проблема заключается в следующем: поскольку этот код будет использоваться в разных программах, в нем могут определяться переменные с именами, конфликтующими с именами, определяемыми в самих программах.

Чтобы избежать подобных конфликтов, импортируемый код можно поместить внутрь функции и затем обращаться к ней. Благодаря этому переменные будут определяться внутри объекта вызова функции:

```
function init( ) {
// Здесь располагается импортируемый программный код.
// Любые объявленные переменные станут свойствами объекта вызова,
// тем самым будет исключена вероятность конфликтов
// с глобальным пространством имен.
}
init( ); // Не забудьте вызвать функцию!
```

Этот фрагмент добавляет единственное свойство в глобальное пространство имен – свойство `init`, которое ссылается на функцию. Если даже добавление единственного свойства покажется вам излишним, можно определить и вызвать анонимную функцию в одном выражении. Вот фрагмент, который работает именно таким образом:

```
(function( ) { // Это безымянная функция.
// Здесь располагается импортируемый программный код. Любые
// объявленные переменные станут свойствами объекта вызова, тем самым
// исключается вероятность конфликтов с глобальным пространством имен.
})(); // конец функционального литерала и его вызов.
```

Обратите внимание на круглые скобки, окружающие функциональный литерал, – этого требует синтаксис JavaScript.

8.8.4. Вложенные функции в качестве замыканий

Тот факт, что JavaScript допускает объявление вложенных функций, позволяет использовать функции как обычные данные и способствует организации взаимодействий между цепочками областей видимости, что позволяет получать интересные и мощные эффекты. Прежде чем приступить к описанию, рассмотрим функцию `g`, которая определяется внутри функции `f`. Когда вызывается функция `f`, ее цепочка областей видимости содержит объект вызова, за которым следует глобальный объект. Функция `g` определяется внутри функции `f`, таким образом, цепочка областей видимости этой функции сохраняется как часть определения функции `g`. Когда вызывается функция `g`, ее цепочка областей видимости содержит уже три объекта: собственный объект вызова, объект вызова функции `f` и глобальный объект.

Порядок работы вложенных функций совершенно понятен, когда они вызываются в той же лексической области видимости, в которой определены. Например, следующий фрагмент не содержит ничего необычного:

```
var x = "глобальная";
function f( ) {
    var x = "локальная";
    function g() { alert(x); }
    g();
}
f(); // При обращении к этой функции будет выведено слово "локальная"
```

Однако в JavaScript функции рассматриваются как обычные данные, поэтому их можно возвращать из других функций, присваивать свойствам объектов, сохранять в массивах и т. д. В этом нет ничего необычного до тех пор, пока на арену не выходят вложенные функции. Рассмотрим следующий фрагмент, где определена функция, которая возвращает вложенную функцию. При каждом обращении к ней она возвращает функцию. Сам JavaScript-код при этом не меняется, но от вызова к вызову может изменяться область видимости, поскольку при каждом обращении к объемлющей функции могут изменяться ее аргументы. (То есть в цепочке областей видимости будет изменяться объект вызова объемлющей функции.) Если попробовать сохранить возвращаемые функции в массиве и затем вызвать каждую из них, можно заметить, что они будут возвращать разные значения. Поскольку программный код функции при этом не изменяется и каждая из них вызывается в той же самой области видимости, единственное, чем можно объяснить разницу, – это различия между областями видимости, в которых функции были определены:

```
// Эта функция возвращает другую функцию
// От вызова к вызову изменяется область видимости,
// в которой была определена вложенная функция
function makefunc(x) {
    return function() { return x; }
}

// Вызвать makefunc() несколько раз и сохранить результаты в массиве:
var a = [makefunc(0), makefunc(1), makefunc(2)];

// Теперь вызвать функции и вывести полученные от них значения.
// Хотя тело каждой функции остается неизменным, их области видимости
// изменяются, и при каждом вызове они возвращают разные значения:
alert(a[0]( )); // Выведет 0
alert(a[1]( )); // Выведет 1
alert(a[2]( )); // Выведет 2
```

Результаты работы этого фрагмента в точности соответствуют ожиданиям, если строго следовать правилу лексической области видимости: функция исполняется в той области видимости, в которой она была определена. Однако самое интересное состоит в том, что области видимости продолжают существовать и после выхода из объемлющей функции. В обычной ситуации этого не происходит. Когда вызывается функция, создается объект вызова и размещается в ее области видимости. Когда функция завершает работу, объект вызова удаляется из цепочки вызова. Пока дело не касается вложенных функций, цепочка видимости

является единственной ссылкой на объект вызова. Когда ссылка на объект удаляется из цепочки, в дело вступает сборщик мусора.

Однако ситуация меняется с появлением вложенных функций. Когда создается определение вложенной функции, оно содержит ссылку на объект вызова, поскольку этот объект находится на вершине цепочки областей видимости, в которой определяется функция. Если вложенная функция используется только внутри объемлющей функции, единственная ссылка на вложенную функцию – это объект вызова. Когда внешняя функция возвращает управление, вложенная функция ссылается на объект вызова, а объект вызова – на вложенную функцию, и никаких других ссылок на них не существует, благодаря этому они становятся доступными для механизма сборки мусора.

Все меняется, если ссылка на вложенную функцию сохраняется в глобальной области видимости. Это происходит, когда вложенная функция передается в виде возвращаемого значения объемлющей функции или сохраняется в виде свойства какого-либо другого объекта. В этом случае появляется внешняя ссылка на вложенную функцию, при этом вложенная функция продолжает ссылаться на объект вызова объемлющей функции. В результате все объекты вызова, созданные при каждом таком обращении к объемлющей функции, продолжают свое существование, а вместе с ними продолжают существование имена и значения аргументов функции и локальных переменных. JavaScript-программы не имеют возможности напрямую воздействовать на объект вызова, но его свойства являются частью цепочки областей видимости, создаваемой при любом обращении к вложенной функции. (Примечательно, что если объемлющая функция сохранит глобальные ссылки на две вложенные функции, эти вложенные функции будут совместно использовать один и тот же объект вызова, а изменения, появившиеся в результате обращения к одной из функций, будут видимы в другой.)

Функции в JavaScript представляют собой комбинацию исполняемого программного кода и области видимости, в которой этот код исполняется. Такая комбинация программного кода и области видимости в литературе по компьютерной тематике называется *замыканием* (*closure*). Все JavaScript-функции являются замыканиями. Однако все эти замыкания представляют интерес лишь в только что рассмотренной ситуации, когда вложенная функция экспортируется за пределы области видимости, в которой она была определена. Вложенные функции, используемые таким образом, нередко явно называют замыканиями.

Замыкания – это очень интересная и мощная техника программирования. Хотя замыкания используются довольно редко, они достойны того, чтобы изучить их. Если вы поймете механизм замыканий, вы без труда разберетесь в областях видимости и без ложной скромности сможете назвать себя опытным программистом на JavaScript.

8.8.4.1. Примеры замыканий

Иногда возникает необходимость, чтобы функция запоминала некоторое значение между вызовами. Значение не может сохраняться в локальной переменной, поскольку между обращениями к функции не сохраняется сам объект вызова. С ситуацией поможет справиться глобальная переменная, но это приводит к захламлению пространства имен. В разделе 8.6.3 была представлена функция `uniqueInteger()`, которая задействует для этих целей собственное свойство. Одна-

ко можно пойти дальше и для создания *частной* (*private*) неисчезающей переменной использовать замыкание. Вот пример такой функции, для начала без замыкания:

```
// При каждом вызове возвращает разные значения
uniqueID = function() {
    if (!arguments.callee.id) arguments.callee.id = 0;
    return arguments.callee.id++;
};
```

Проблема заключается в том, что свойство `uniqueID.id` доступно за пределами функции и может быть установлено в значение 0, вследствие чего будет нарушено соглашение, по которому функция обязуется никогда не возвращать одно и то же значение дважды. Для решения этой проблемы можно сохранять значение в замыкании, доступ к которому будет иметь только эта функция:

```
uniqueID = (function() { // Значение сохраняется в объекте вызова функции
    var id = 0;          // Это частная переменная, сохраняющая свое
                         // значение между вызовами функции
    // Внешняя функция возвращает вложенную функцию, которая имеет доступ
    // к этому значению. Эта вложенная функция сохраняется
    // в переменной uniqueID выше.
    return function() { return id++; }; // Вернуть и увеличить
})(); // Вызов внешней функции после ее определения.
```

Пример 8.6 – это еще один пример замыкания. В нем демонстрируется, как частные переменные, подобные той, что была показана ранее, могут совместно использоваться несколькими функциями.

Пример 8.6. Создание частных свойств с помощью замыканий

```
// Эта функция добавляет методы доступа к свойству объекта "o"
// с заданными именами. Методы получают имена get<name>
// и set<name>. Если дополнительно предоставляется
// функция проверки, метод записи будет использовать ее
// для проверки значения перед сохранением. Если функция проверки
// возвращает false, метод записи генерирует исключение.
//
// Необычность такого подхода заключается в том, что значение
// свойства, доступного методам, сохраняется не в виде свойства
// объекта "o", а в виде локальной переменной этой функции.
// Кроме того, методы доступа определены локально, в этой функции
// и обеспечивают доступ к этой локальной переменной.
// Примечательно, что значение доступно только для этих двух методов
// и не может быть установлено или изменено иначе, как методом записи.
function makeProperty(o, name, predicate) {
    var value; // This is the property value

    // Метод чтения просто возвращает значение.
    o["get" + name] = function() { return value; };

    // Метод записи сохраняет значение или генерирует исключение,
    // если функция проверки отвергает это значение.
    o["set" + name] = function(v) {
        if (predicate && !predicate(v))
            throw "set" + name + ": неверное значение " + v;
```

```
        else
            value = v;
    };
}

// Следующий фрагмент демонстрирует работу метода makeProperty().
var o = {}; // Пустой объект

// Добавить методы доступа к свойству с именами getName() и setName()
// Обеспечить допустимость только строковых значений
makeProperty(o, "Name", function(x) { return typeof x == "string"; });

o.setName("Frank");  // Установить значение свойства
print(o.getName( )); // Получить значение свойства
o.setName(0);        // Попытаться установить значение ошибочного типа
```

Самый практичный и наименее искусственный пример использования замыканий, который мне известен, – это механизм точек останова, разработанный Стивом Йеном (Steve Yen) и опубликованный на сайте *http://trimpath.com* как часть клиентской платформы TrimPath. *Точка останова* – это точка внутри функции, где останавливается исполнение программы, и разработчик получает возможность просмотреть значения переменных, вычислить выражения, вызвать функции и тому подобное. В механизме точек останова, придуманном Стивом, замыкания служат для хранения контекста исполнения текущей функции (включая локальные переменные и входные аргументы) и с помощью глобальной функции `eval()` позволяют просмотреть содержимое этого контекста. Функция `eval()` исполняет строки на языке JavaScript и возвращает полученные значения (подробнее об этой функции можно прочитать в третьей части книги). Вот пример вложенной функции, которая работает как замыкание, выполняющее проверку своего контекста исполнения:

```
// Запомнить текущий контекст и позволить проверить его
// с помощью функции eval( )
var inspector = function($) { return eval($); }
```

В качестве имени аргумента эта функция использует малораспространенный идентификатор `$`, чем снижается вероятность конфликта имен в инспектируемой области видимости.

Создать точку останова можно, передав это замыкание в функцию, как показано в примере 8.7.

Пример 8.7. Точки останова на основе замыканий

```
// Эта функция является реализацией точки останова. Она предлагает
// пользователю ввести выражение, вычисляет его с использованием
// замыкания и выводит результат. Используемое замыкание предоставляет
// доступ к проверяемой области видимости, таким образом любая функция
// будет создавать собственное замыкание.
//
// Реализовано по образу и подобию функции breakpoint() Стива Йена
// http://trimpath.com/project/wiki/TrimBreakpoint
function inspect(inspector, title) {
    var expression, result;

    // Существует возможность отключать точки останова
```

```
    // за счет создания свойства "ignore" у этой функции.
    if ("ignore" in arguments.callee) return;

    while(true) {
        // Определить, как вывести запрос перед пользователем
        var message = "";
        // Если задан аргумент title, вывести его первым
        if (title) message = title + "\n";
        // Если выражение уже вычислено, вывести его вместе с его значением
        if (expression) message += "\n"+expression+" ==> "+result+"\n";
        else expression = "";
        // Типовое приглашение к вводу всегда должно выводиться:
        message += "Введите выражение, которое следует вычислить:";

        // Получить ввод пользователя, вывести приглашение и использовать
        // последнее выражение как значение по умолчанию.
        expression = prompt(message, expression);

        // Если пользователь ничего не ввел (или щелкнул на кнопке Отменить),
        // работу в точке останова можно считать оконченной
        // и вернуть управление.
        if (!expression) return;

        // В противном случае вычислить выражение с использованием
        // замыкания в инспектируемом контексте исполнения.
        // Результаты будут выведены на следующей итерации.
        result = inspector(expression);
    }
}
```

Обратите внимание: для вывода информации и ввода строки пользователя функция `inspect()` **из примера 8.7 задействует метод** `Window.prompt()` **(подробнее об этом методе рассказывается в четвертой части книги).**

Рассмотрим пример функции, вычисляющей факториал числа и использующей механизм точек останова:

```
function factorial(n) {
    // Создать замыкание для этой функции
    var inspector = function($) { return eval($); }
    inspect(inspector, "Вход в функцию factorial()");

    var result = 1;
    while(n > 1) {
        result = result * n;
        n--;
        inspect(inspector, "factorial( ) loop");
    }

    inspect(inspector, "Выход из функции factorial()");
    return result;
}
```

8.8.4.2. Замыкания и утечки памяти в Internet Explorer

В веб-броузере Microsoft Internet Explorer используется достаточно слабая разновидность механизма сборки мусора для объектов ActiveX и DOM-элементов на

стороне клиента. Для этих элементов на стороне клиента выполняется подсчет ссылок, и они утилизируются интерпретатором, только когда значение счетчика ссылок достигает нуля. Однако такая схема оказывается неработоспособной в случае циклических ссылок, например, когда базовый JavaScript-объект ссылается на элемент документа, а этот элемент имеет свойство (например, обработчик события), которое, в свою очередь, хранит ссылку на базовый JavaScript-объект.

Такого рода циклические ссылки часто возникают при работе с замыканиями. При использовании техники замыканий не забывайте, что объект вызова замкнутой функции, включающий в себя все аргументы функции и локальные переменные, будет продолжать существовать до тех пор, пока существует само замыкание. Если какие-либо аргументы функции или локальные переменные ссылаются на объект, может возникнуть утечка памяти.

Обсуждение этой проблемы выходит за рамки темы книги. За дополнительной информацией обращайтесь по адресу: *http://msdn.microsoft.com/library/en-us/IETechCol/dnwebgen/ie_leak_patterns.asp*.

8.9. Конструктор Function()

Как уже говорилось ранее, функции обычно определяются с помощью ключевого слова `function` либо в форме определения функции, либо посредством функционального литерала. Однако помимо этого существует возможность создания функций с помощью конструктора `Function()`. Создание функций с помощью конструктора `Function()` обычно сложнее, чем с помощью функционального литерала, поэтому такая методика распространена не так широко. Вот пример создания функции подобным образом:

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

Эта строка создает новую функцию, более или менее эквивалентную функции, определенной с помощью более привычного синтаксиса:

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

Конструктор `Function()` принимает любое количество строковых аргументов. Последний аргумент – это тело функции. Он может содержать произвольные JavaScript-инструкции, отделенные друг от друга точками с запятой. Все остальные аргументы конструктора представляют собой строки, задающие имена параметров определяемой функции. Если вы определяете функцию без аргументов, конструктору передается только одна строка – тело функции.

Обратите внимание: конструктору `Function()` не передается аргумент, задающий имя создаваемой им функции. Неименованные функции, созданные с помощью конструктора `Function()`, иногда называются анонимными функциями.

Есть несколько моментов, связанных с конструктором `Function()`, о которых следует упомянуть особо:

- Конструктор `Function()` позволяет динамически создавать и компилировать функции во время исполнения программы. В чем-то он напоминает функцию `eval()` (за информацией обращайтесь к третьей части книги).

- Конструктор `Function()` компилирует и создает новую функцию при каждом вызове. Если вызов конструктора производится в теле цикла или часто вызываемой функции, это может отрицательно сказаться на производительности программы. В противовес этому функциональные литералы или вложенные функции, находящиеся внутри цикла, не перекомпилируются на каждой итерации, а кроме того, в случае литералов не создается новый объект функции. (Хотя, как уже отмечалось ранее, может создаваться новое замыкание, хранящее лексический контекст, в котором была определена функция.)
- И последний очень важный момент: когда функция создается с помощью конструктора `Function()`, не учитывается текущая лексическая область видимости — функции, созданные таким способом, всегда компилируются как функции верхнего уровня, что наглядно демонстрируется в следующем фрагменте:

```
var y = "глобальная";
function constructFunction() {
    var y = "локальная";
    return new Function("return y"); // Не сохраняет локальный контекст!
}
// Следующая строка выведет слово "глобальная", потому что функция,
// созданная конструктором Function(), не использует локальный контекст.
// Если функция была определена как литерал,
// эта строка вывела бы слово "локальная".
alert(constructFunction()()); // Выводит слово "глобальная"
```

9

Классы, конструкторы и прототипы

Введение в JavaScript-объекты было дано в главе 7, в которой каждый объект трактовался как уникальный набор свойств, отличающих его от любых других объектов. В большинстве объектно-ориентированных языков программирования существует возможность определять *классы* объектов и затем создавать отдельные объекты как *экземпляры* этих классов. Например, можно объявить класс `Complex`, призванный представлять комплексные числа и выполнять арифметические действия с этими числами, тогда объект `Complex` представлял бы единственное комплексное число и мог бы создаваться как экземпляр этого класса.

Язык JavaScript не обладает полноценной поддержкой классов, как другие языки, например Java, C++ или C#.[1] Тем не менее в JavaScript существует возможность определять псевдоклассы с помощью таких инструментальных средств, как функции-конструкторы и прототипы объектов. В этой главе рассказывается о конструкторах и прототипах и приводится ряд примеров некоторых псевдоклассов и даже псевдоподклассов JavaScript.

За неимением лучшего термина в этой главе неофициально я буду пользоваться словом «класс», поэтому будьте внимательны и не путайте эти неформальные «классы» с настоящими классами, которые поддерживаются в JavaScript 2.0 и в других языках программирования.

9.1. Конструкторы

В главе 7 демонстрировался порядок создания новых пустых объектов как с помощью литерала `{}`, так и с помощью следующего выражения:

```
new Object()
```

Кроме того, была продемонстрирована возможность создания объектов других типов примерно следующим образом:

---

[1] Полноценную поддержку классов планируется реализовать в JavaScript 2.0.

```
var array = new Array(10);
var today = new Date( );
```

За оператором new должно быть указано имя функции-конструктора. Оператор создает новый пустой объект без каких-либо свойств, а затем вызывает функцию, передавая ей только что созданный объект в виде значения ключевого слова this. Функция, применяемая совместно с оператором new, называется *функцией-конструктором*, или просто *конструктором*. Главная задача конструктора заключается в инициализации вновь созданного объекта – установке всех его свойств, которые необходимо инициализировать до того, как объект сможет использоваться программой. Чтобы определить собственный конструктор, достаточно написать функцию, добавляющую новые свойства к объекту, на который ссылается ключевое слово this. В следующем фрагменте приводится определение конструктора, с помощью которого затем создаются два новых объекта:

```
// Определяем конструктор.
// Обратите внимание, как инициализируется объект с помощью "this".
function Rectangle(w, h) {
    this.width = w;
    this.height = h;
}

// Вызываем конструктор для создания двух объектов Rectangle. Мы передаем ширину и высоту
// конструктору, чтобы можно было правильно проинициализировать оба новых объекта.
var rect1 = new Rectangle(2, 4);    // rect1 = { width:2, height:4 };
var rect2 = new Rectangle(8.5, 11); // rect2 = { width:8.5, height:11 };
```

Обратите внимание на то, как конструктор использует свои аргументы для инициализации свойств объекта, на который ссылается ключевое слово this. Здесь мы определили класс объектов, просто создав соответствующую функцию-конструктор – все объекты, созданные с помощью конструктора Rectangle(), гарантированно будут иметь инициализированные свойства width и height. Это означает, что учитывая данное обстоятельство, можно организовать единообразную работу со всеми объектами класса Rectangle. Поскольку каждый конструктор определяет отдельный класс объектов, стилистически очень важно присвоить такое имя функции-конструктору, которое будет явно отражать класс объектов, создаваемых с ее помощью. Например, строка new Rectangle(1, 2), создающая объект прямоугольника, выглядит гораздо более понятно, нежели new init_rect(1, 2).

Обычно функции-конструкторы ничего не возвращают, они лишь инициализируют объект, полученный в качестве значения ключевого слова this. Однако для конструкторов допускается возможность возвращать объект, в этом случае возвращаемый объект становится значением выражения new. При этом объект, переданный конструктору в виде значения ключевого слова this, просто уничтожается.

9.2. Прототипы и наследование

В главе 8 говорилось, что *метод* – это функция, которая вызывается как свойство объекта. Когда функция вызывается таким способом, объект, посредством которого производится вызов, становится значением ключевого слова this. Предположим, что необходимо рассчитать площадь прямоугольника, представленного объектом Rectangle. Вот один из возможных способов:

```
function computeAreaOfRectangle(r) { return r.width * r.height; }
```

Эта функция прекрасно справляется с возложенными на нее задачами, но она не является объектно-ориентированной. Работая с объектом, лучше всего вызывать методы этого объекта, а не передавать объекты посторонним функциям в качестве аргументов. Этот подход демонстрируется в следующем фрагменте:

```
// Создать объект Rectangle
var r = new Rectangle(8.5, 11);
// Добавить к объекту метод
r.area = function() { return this.width * this.height; }
// Теперь рассчитать площадь, вызвав метод объекта
var a = r.area();
```

Конечно же, не совсем удобно добавлять новый метод к объекту перед его использованием. Однако ситуацию можно улучшить, если инициализировать свойство `area` в функции-конструкторе. Вот как выглядит улучшенная реализация конструктора `Rectangle()`:

```
function Rectangle(w, h) {
    this.width = w;
    this.height = h;
    this.area = function( ) { return this.width * this.height; }
}
```

С новой версией конструктора тот же самый алгоритм можно реализовать по-другому:

```
// Найти площадь листа бумаги формата U.S. Letter в квадратных дюймах
var r = new Rectangle(8.5, 11);
var a = r.area();
```

Такое решение выглядит гораздо лучше, но оно по-прежнему не является оптимальным. Каждый созданный прямоугольник будет иметь три свойства. Свойства `width` и `height` могут иметь уникальные значения для каждого прямоугольника, но свойство `area` каждого отдельно взятого объекта `Rectangle` всегда будет ссылаться на одну и ту же функцию (разумеется, это свойство можно изменить в процессе работы, но, как правило, предполагается, что методы объекта не должны меняться). Применение отдельных свойств для хранения методов объектов, которые могли бы совместно использоваться всеми экземплярами одного и того же класса, – это достаточно неэффективное решение.

Однако и эту проблему можно решить. Оказывается, все объекты в JavaScript содержат внутреннюю ссылку на объект, известный как прототип. Любые свойства прототипа становятся свойствами другого объекта, для которого он является прототипом. То есть, говоря другими словами, любой объект в JavaScript *наследует* свойства своего прототипа.

В предыдущем разделе было показано, как оператор `new` создает пустой объект и затем вызывает функцию-конструктор. Но история на этом не заканчивается. После создания пустого объекта оператор `new` устанавливает в этом объекте ссылку на прототип. Прототипом объекта является значение свойства `prototype` функции-конструктора. Все функции имеют свойство `prototype`, которое инициализируется в момент определения функции. Начальным значением этого свойства является объект с единственным свойством. Это свойство называется `constructor` и значением его является ссылка на саму функцию-конструктор,

с которой этот прототип ассоциируется. (Описание свойства `constructor` приводилось в главе 7, здесь же объясняется, почему каждый объект обладает свойством `constructor`.) Любые свойства, добавленные к прототипу, автоматически становятся свойствами объектов, инициализируемых конструктором.

Более понятно это можно объяснить на примере. Вот новая версия конструктора `Rectangle()`:

```
// Функция-конструктор инициализирует те свойства, которые могут
// иметь уникальные значения для каждого отдельного экземпляра.
function Rectangle(w, h) {
    this.width = w;
    this.height = h;
}

// Прототип объекта содержит методы и другие свойства, которые должны
// совместно использоваться всеми экземплярами этого класса.
Rectangle.prototype.area = function() { return this.width * this.height; }
```

Конструктор определяет «класс» объектов и инициализирует свойства, такие как `width` и `height`, которые могут отличаться для каждого экземпляра класса. Объект-прототип связан с конструктором, и каждый объект, инициализируемый конструктором, наследует тот набор свойств, который имеется в прототипе. Это значит, что объект-прототип – идеальное место для методов и других свойств-констант.

Обратите внимание, что наследование осуществляется автоматически как часть процесса поиска значения свойства. Свойства *не* копируются из объекта-прототипа в новый объект; они просто присутствуют, как если бы были свойствами этих объектов. Это имеет два важных следствия. Во-первых, использование объектов-прототипов может в значительной степени уменьшить объем памяти, требуемый для каждого объекта, т. к. объекты могут наследовать многие из своих свойств. Во-вторых, объект наследует свойства, даже если они были добавлены в прототип *после* создания объекта. Это означает наличие возможности добавлять новые методы к существующим классам (хотя это не совсем правильно).

Унаследованные свойства ничем не отличаются от обычных свойств объекта. Они поддаются перечислению в цикле `for/in` и могут проверяться с помощью оператора `in`. Отличить их можно только с помощью метода `Object.hasOwnProperty()`:

```
var r = new Rectangle(2, 3);
r.hasOwnProperty("width"); // true: width - непосредственное свойство "r"
r.hasOwnProperty("area");  // false: area - унаследованное свойство "r"
"area" in r;               // true: area - свойство объекта "r"
```

9.2.1. Чтение и запись унаследованных свойств

У каждого класса имеется один объект-прототип с одним наборов свойств, но потенциально может существовать множество экземпляров класса, каждый из которых наследует свойства прототипа. Свойство прототипа может наследоваться многими объектами, поэтому интерпретатор JavaScript должен обеспечить фундаментальную асимметричность между чтением и записью значений свойств. Когда вы читаете свойство `p` объекта `o`, JavaScript сначала проверяет, есть ли у объекта `o` свойство с именем `p`. Если такого свойства нет, то проверяется, есть ли свойство с именем `p` в объекте-прототипе. Так работает наследование на базе прототипов.

Однако когда свойству присваивается значение, JavaScript не использует объект-прототип. Чтобы понять почему, подумайте, что произошло бы в этом случае: предположим, вы пытаетесь установить значение свойства `o.p`, а у объекта `o` нет свойства с именем `p`. Предположим теперь, что JavaScript идет дальше и ищет свойство `p` в объекте-прототипе объекта `o` и позволяет вам изменить значение свойства прототипа. В результате вы изменяете значение `p` для всего класса объектов, а это вовсе не то, что требовалось.

Поэтому наследование свойств происходит только при чтении значений свойств, но не при их записи. Если вы устанавливаете свойство `p` в объекте `o`, который наследует это свойство от своего прототипа, происходит создание нового свойства непосредственно в объекте `p`. Теперь, когда объект `o` имеет собственное свойство с именем `p`, он больше не наследует значение `p` от прототипа. Когда вы читаете значение `p`, JavaScript сначала ищет его в свойствах объекта `o`. Так как он находит свойство `p`, определенное в `o`, ему не требуется искать его в объекте-прототипе, и JavaScript никогда не будет искать определенное в нем значение `p`. Мы иногда говорим, что свойство `p` «затеняет» (скрывает) свойство `p` объекта-прототипа. Наследование прототипов может показаться запутанным, но все вышеизложенное хорошо иллюстрирует рис. 9.1.

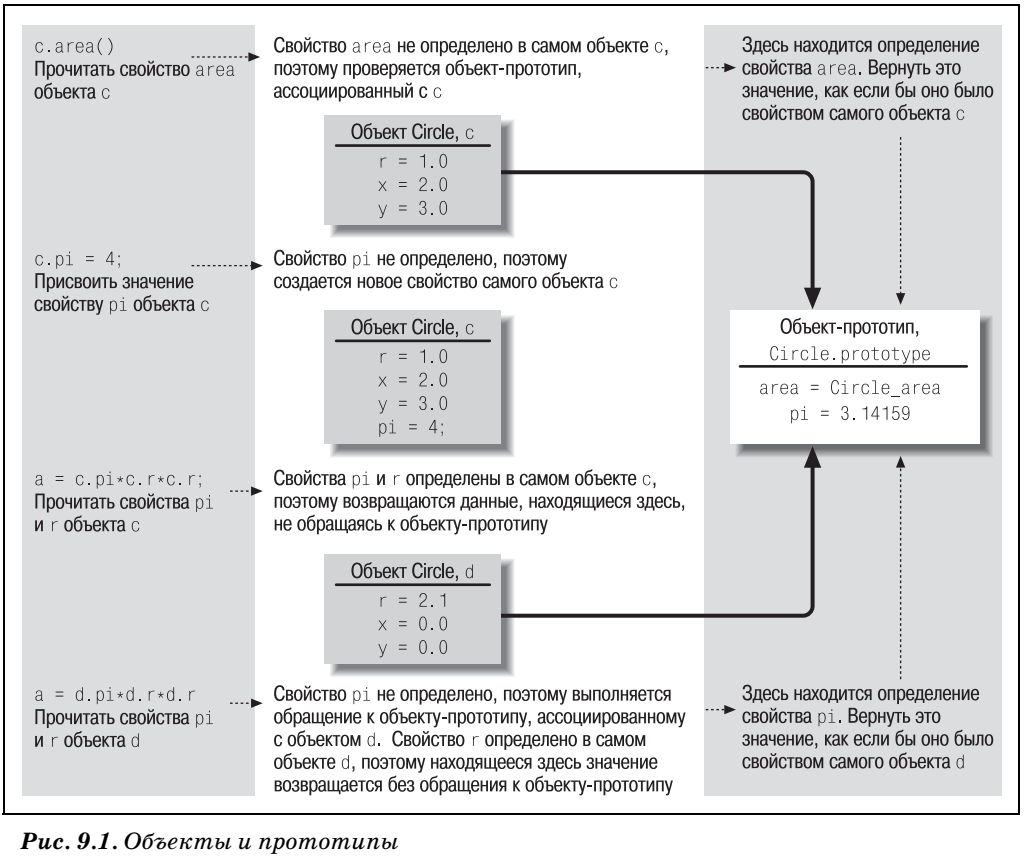

Рис. 9.1. *Объекты и прототипы*

Свойства прототипа совместно используются всеми объектами класса, поэтому, как правило, их имеет смысл применять только для определения свойств, совпадающих для всех объектов класса. Это делает прототипы идеальными для определения методов. Другие свойства с постоянными значениями (такими как математические константы) также подходят для определения в качестве свойств прототипа. Если класс определяет свойство с очень часто используемым значением по умолчанию, то можно определить это свойство и его значение по умолчанию в объекте-прототипе. Тогда те немногие объекты, которые хотят изменить значение по умолчанию, могут создавать свои частные копии свойства и определять собственные значения, отличные от значения по умолчанию.

9.2.2. Расширение встроенных типов

Не только классы, определенные пользователем, имеют объекты-прототипы. Встроенные классы, такие как `String` и `Date`, также имеют объекты-прототипы, и вы можете присваивать им значения. Например, следующий фрагмент определяет новый метод, доступный всем объектам `String`:

```
// Возвращает true, если последним символом является значение аргумента c
String.prototype.endsWith = function(c) {
    return (c == this.charAt(this.length-1))
}
```

Определив новый метод `endsWith()` в объекте-прототипе `String`, мы сможем обратиться к нему следующим образом:

```
var message = "hello world";
message.endsWith('h') // Возвращает false
message.endsWith('d') // Возвращает true
```

Против такого расширения возможностей встроенных типов можно привести достаточно сильные аргументы: в случае расширения некоторого встроенного типа, по сути, создается самостоятельная версия базового прикладного JavaScript-интерфейса. Любые другие программисты, которые будут читать или сопровождать ваш код, могут прийти в недоумение, встретив методы, о которых они ранее не слышали. Если только вы не собираетесь создавать низкоуровневую JavaScript-платформу, которая будет воспринята многими другими программистами, лучше оставить прототипы встроенных объектов в покое.

Обратите внимание: *никогда* не следует добавлять свойства к объекту `Object.prototype`. Любые добавляемые свойства и методы становятся перечислимыми для цикла `for/in`, поэтому добавив их к объекту `Object.prototype`, вы сделаете их доступными во всех JavaScript-объектах. Пустой объект `{}`, как предполагается, не имеет перечислимых свойств. Любое расширение `Object.prototype` превратится в перечислимое свойство пустого объекта, что, скорее всего, приведет к нарушениям в функционировании программного кода, который работает с объектами как с ассоциативными массивами.

Техника расширения встроенных объектов, о которой сейчас идет речь, гарантированно работает только в случае применения к «родным» объектам базового языка JavaScript. Когда JavaScript встраивается в некоторый контекст, например в веб-броузер или в Java-приложение, он получает доступ к дополнительным «платформозависимым» объектам, таким как объекты веб-броузера, пред-

ставляющие содержимое документа. Эти объекты, как правило, не имеют ни конструктора, ни прототипа и потому недоступны для расширения.

Один из случаев, когда можно расширять прототипы встроенных объектов достаточно безопасно и даже желательно, – это добавление стандартных методов прототипов в старых несовместимых реализациях JavaScript, где эти свойства и методы отсутствуют. Например, метод `Function.apply()` в Microsoft Internet Explorer версий 4 и 5 не поддерживается. Это достаточно важная функция, поэтому иногда вам может встретиться код, который добавляет эту функцию:

```
// Если функция Function.apply() не реализована, можно добавить
// этот фрагмент, основанный на разработках Аарона Будмана (Aaron Boodman).
if (!Function.prototype.apply) {
    // Вызвать эту функцию как метод заданного объекта с указанными
    // параметрами. Для этих целей здесь используется функция eval()
    Function.prototype.apply = function(object, parameters) {
        var f = this;                // Вызываемая функция
        var o = object || window;    // Объект, через который выполняется вызов
        var args = parameters || []; // Передаваемые аргументы

        // Временно превратить функцию в метод объекта o.
        // Для этого выбирается имя метода, которое скорее всего отсутствует
        o._$_apply_$_ = f;

        // Вызов метода выполняется с помощью eval().
        // Для этого необходимо сконструировать строку вызова.
        // В первую очередь собирается список аргументов.
        var stringArgs = [];
        for(var i = 0; i < args.length; i++)
            stringArgs[i] = "args[" + i + "]";

        // Объединить строки с аргументами в единый список,
        // разделив аргументы запятыми.
        var arglist = stringArgs.join(",");

        // Теперь собрать всю строку вызова метода
        var methodcall = "o._$_apply_$_(" + arglist + ");";

        // С помощью функции eval() вызвать метод
        var result = eval(methodcall);

        // Удалить метод из объекта
        delete o._$_apply_$_;

        // И вернуть результат
        return result;
    };
}
```

В качестве еще одного примера рассмотрим новые методы массивов, реализованные в Firefox 1.5 (см. раздел 7.7.10). Если необходимо задействовать метод `Array.map()` и при этом желательно сохранить совместимость с платформами, где этот метод не поддерживается, можно воспользоваться следующим фрагментом:

```
// Array.map() вызывает функцию f для каждого элемента массива
// и возвращает новый массив, содержащий результаты каждого вызова функции.
// Если map() вызывается с двумя аргументами, функция f вызывается как метод
```

```
// второго аргумента. Функции f() передается 3 аргумента. Первый представляет
// значение элемента массива, второй – индекс элемента, третий – сам массив.
// В большинстве случаев достаточно передать только первый аргумент.
if (!Array.prototype.map) {
    Array.prototype.map = function(f, thisObject) {
        var results = [];
        for(var len = this.length, i = 0; i < len; i++) {
            results.push(f.call(thisObject, this[i], i, this));
        }
        return results;
    }
}
```

9.3. Объектно-ориентированный язык JavaScript

Хотя JavaScript поддерживает тип данных, который мы называем объектом, в нем нет формального понятия класса. Это в значительной степени отличает его от классических объектно-ориентированных языков программирования, таких как C++ и Java. Общая черта объектно-ориентированных языков – это их строгая типизация и поддержка механизма наследования на базе классов. По этому критерию JavaScript легко исключить из числа истинно объектно-ориентированных языков. С другой стороны, мы видели, что JavaScript активно использует объекты и имеет особый тип наследования на базе прототипов. JavaScript – это истинно объектно-ориентированный язык. Он был реализован под влиянием некоторых других (относительно малоизвестных) объектно-ориентированных языков, в которых вместо наследования на основе классов реализовано наследование на базе прототипов.

Несмотря на то что JavaScript – это объектно-ориентированный язык, не базирующийся на классах, он неплохо имитирует возможности языков на базе классов, таких как Java и C++. Я употребил термин «класс» в этой главе неформально. В данном разделе проводятся более формальные параллели между JavaScript и истинным наследованием на базе классов в таких языках, как Java и C++.[1]

Начнем с того, что определим некоторые базовые термины. *Объект*, как мы уже видели, – это структура данных, которая содержит различные фрагменты именованных данных, а также может содержать методы для работы с этими фрагментами данных. Объект группирует связанные значения и методы в единый удобный набор, который, как правило, облегчает процесс программирования, увеличивая степень модульности и возможности для многократного использования кода. Объекты в JavaScript могут иметь произвольное число свойств, и свойства могут добавляться в объект динамически. В строго типизированных языках, таких как Java и C++, это не так. В них любой объект имеет предопределенный набор свойств[2], а каждое свойство имеет предопределенный тип. Имитируя объектно-ориентированные приемы программирования при помощи JavaScript-

---

[1] Этот раздел рекомендуется прочитать даже тем, кто незнаком с этими языками и упомянутым стилем объектно-ориентированного программирования.

[2] Обычно в Java и C++ они называются «полями», но здесь мы будем называть их свойствами, поскольку такая терминология принята в JavaScript.

объектов, мы, как правило, заранее определяем набор свойств для каждого объекта и тип данных, содержащихся в каждом свойстве.

В Java и C++ *класс* определяет структуру объекта. Класс точно задает поля, которые содержатся в объекте, и типы данных этих полей. Он также определяет методы для работы с объектом. В JavaScript нет формального понятия класса, но, как мы видели, в этом языке приближение к возможностям классов реализуется с помощью конструкторов и объектов-прототипов.

И JavaScript, и объектно-ориентированные языки, основывающиеся на классах, допускают наличие множества объектов одного класса. Мы часто говорим, что объект – это *экземпляр* класса. Таким образом, одновременно может существовать множество экземпляров любого класса. Иногда для описания процесса создания объекта (т. е. экземпляра класса) используется термин *создание экземпляра*.

В Java обычная практика программирования заключается в присвоении классам имен с первой прописной буквой, а объектам – со всеми строчными. Это соглашение помогает отличать классы и объекты в исходных текстах. Этому же соглашению желательно следовать и при написании программ на языке JavaScript. Например, в предыдущих разделах мы определили класс `Rectangle` и создавали экземпляры этого класса с именами, такими как `rect`.

Члены Java-класса могут принадлежать одному из четырех основных типов: свойства экземпляра, методы экземпляра, свойства класса и методы класса. В следующих разделах мы рассмотрим различия между этими типами и поговорим о том, как JavaScript имитирует эти типы.

9.3.1. Свойства экземпляра

Каждый объект имеет собственные копии *свойств экземпляра*. Другими словами, если имеется 10 объектов данного класса, то имеется и 10 копий каждого свойства экземпляра. Например, в нашем классе `Rectangle` любой объект `Rectangle` имеет свойство `width`, определяющее ширину прямоугольника. В данном случае `width` представляет собой свойство экземпляра. А поскольку каждый объект имеет собственную копию свойства экземпляра, доступ к этим свойствам можно получить через отдельные объекты. Если, например, `r` – это объект, представляющий собою экземпляр класса `Rectangle`, мы можем получить его ширину следующим образом:

```
r.width
```

По умолчанию любое свойство объекта в JavaScript является свойством экземпляра. Однако чтобы по-настоящему имитировать объектно-ориентированное программирование, мы будем говорить, что свойства экземпляра в JavaScript – это те свойства, которые создаются и/или инициализируются функцией-конструктором.

9.3.2. Методы экземпляра

Метод экземпляра во многом похож на свойство экземпляра, за исключением того, что это метод, а не значение. (В Java функции и методы не являются данными, как это имеет место в JavaScript, поэтому в Java данное различие выражено более четко.) Методы экземпляра вызываются по отношению к определен-

ному объекту, или экземпляру. Метод `area()` нашего класса `Rectangle` представляет собой метод экземпляра. Он вызывается для объекта `Rectangle` следующим образом:

```
a = r.area( );
```

Методы экземпляра ссылаются на объект, или экземпляр, с которым они работают, при помощи ключевого слова `this`. Метод экземпляра может быть вызван для любого экземпляра класса, но это не значит, что каждый объект содержит собственную копию метода, как в случае свойства экземпляра. Вместо этого каждый метод экземпляра совместно используется всеми экземплярами класса. В JavaScript мы определяем метод экземпляра класса путем присваивания функции свойству объекта-прототипа в конструкторе. Так, все объекты, созданные данным конструктором, совместно используют унаследованную ссылку на функцию и могут вызывать ее с помощью приведенного синтаксиса вызова методов.

9.3.2.1. Методы экземпляра и ключевое слово this

Если у вас есть опыт работы с такими языками, как Java или C++, вы наверняка заметили одно важное отличие между методами экземпляров в этих языках и методами экземпляров в JavaScript. В Java и C++ область видимости методов экземпляров включает объект `this`. Так, например, метод `area` в Java может быть реализован проще:

```
return width * height;
```

Однако в JavaScript приходится явно вставлять ключевое слово `this` перед именами свойств:

```
return this.width * this.height;
```

Если вам покажется неудобным вставлять `this` перед каждым именем свойства экземпляра, можно воспользоваться инструкцией `with` (описываемой в разделе 6.18), например:

```
Rectangle.prototype.area = function( ) {
    with(this) {
        return width*height;
    }
}
```

9.3.3. Свойства класса

Свойство класса в Java – это свойство, связанное с самим классом, а не с каждым экземпляром этого класса. Независимо от того, сколько создано экземпляров класса, есть только одна копия каждого свойства класса. Так же, как свойства экземпляра доступны через экземпляр класса, доступ к свойствам класса можно получить через сам класс. Запись `Number.MAX_VALUE` – это пример обращения к свойству класса в JavaScript, означающая, что свойство `MAX_VALUE` доступно через класс `Number`. Так как имеется только одна копия каждого свойства класса, свойства класса по существу являются глобальными. Однако их достоинство состоит в том, что они связаны с классом и имеют логичную нишу, позицию в пространстве имен JavaScript, где они вряд ли будут перекрыты другими свойствами с тем же именем. Очевидно, что свойства класса имитируются в JavaScript

простым определением свойства самой функции-конструктора. Например, свойство класса `Rectangle.UNIT` для хранения единичного прямоугольника с размерами 1x1 можно создать так:

```
Rectangle.UNIT = new Rectangle(1,1);
```

Здесь `Rectangle` – это функция-конструктор, но поскольку функции в JavaScript представляют собой объекты, мы можем создать свойство функции точно так же, как свойства любого другого объекта.

9.3.4. Методы класса

Метод класса – это метод, связанный с классом, а не с экземпляром класса; он вызывается через сам класс, а не через конкретный экземпляр класса. Метод `Date.parse()` (описываемый в третьей части книги) – это метод класса. Он всегда вызывается через объект конструктора `Date`, а не через конкретный экземпляр класса `Date`.

Поскольку методы класса вызываются через функцию-конструктор, они не могут использовать ключевое слово `this` для ссылки на какой-либо конкретный экземпляр класса, поскольку в данном случае `this` ссылается на саму функцию-конструктор. (Обычно ключевое слово `this` в методах классов вообще не используется.)

Как и свойства класса, методы класса являются глобальными. Методы класса не работают с конкретным экземпляром, поэтому их, как правило, проще рассматривать в качестве функций, вызываемых через класс. Как и в случае со свойствами класса, связь этих функций с классом дает им в пространстве имен JavaScript удобную нишу и предотвращает возникновение конфликтов имен. Для того чтобы определить метод класса в JavaScript, требуется сделать соответствующую функцию свойством конструктора.

9.3.5. Пример: класс Circle

В примере 9.1 приводится программный код функции-конструктора и объекта-прототипа, используемых для создания объектов, представляющих круг. Здесь можно найти примеры свойств экземпляра, методов экземпляра, свойств класса и методов класса.

Пример 9.1. Класс Circle

```
// Начнем с конструктора.
function Circle(radius) {
    // r - свойство экземпляра, оно определяется
    // и инициализируется конструктором.
    this.r = radius;
}

// Circle.PI - свойство класса, т. е. свойство функции-конструктора.
Circle.PI = 3.14159;

// Метод экземпляра, который рассчитывает площадь круга.
Circle.prototype.area = function( ) { return Circle.PI * this.r * this.r; }

// Метод класса - принимает два объекта Circle и возвращает объект с большим радиусом.
Circle.max = function(a,b) {
```

```
    if (a.r > b.r) return a;
    else return b;
}

// Примеры использования каждого из этих полей:
var c = new Circle(1.0);      // Создание экземпляра класса Circle
c.r = 2.2;                    // Установка свойства экземпляра r
var a = c.area();             // Вызов метода экземпляра area()
var x = Math.exp(Circle.PI);  // Обращение к свойству PI класса для выполнения расчетов
var d = new Circle(1.2);      // Создание другого экземпляра класса Circle
var bigger = Circle.max(c,d); // Вызов метода класса max()
```

9.3.6. Пример: комплексные числа

В примере 9.2 представлен еще один способ определения класса объектов в JavaScript, но несколько более формальный, чем предыдущий. Код и комментарии достойны тщательного изучения.

Пример 9.2. Класс комплексных чисел

```
/*
 * Complex.js:
 * В этом файле определяется класс Complex для представления комплексных чисел.
 * Вспомним, что комплексное число - это сумма вещественной и мнимой
 * частей числа, и что мнимое число i - это квадратный корень из -1.
 */

/*
 * Первый шаг в определении класса - это определение функции-конструктора
 * класса. Этот конструктор должен инициализировать все свойства
 * экземпляра объекта. Это неотъемлемые "переменные состояния",
 * делающие все экземпляры класса разными.
 */
function Complex(real, imaginary) {
    this.x = real;      // Вещественная часть числа
    this.y = imaginary; // Мнимая часть числа
}

/*
 * Второй шаг в определении класса - это определение методов экземпляра
 * (и возможно других свойств) в объекте-прототипе конструктора.
 * Любые свойства, определенные в этом объекте, будут унаследованы всеми
 * экземплярами класса. Обратите внимание, что методы экземпляра
 * неявно работают с ключевым словом this. Для многих методов никаких
 * других аргументов не требуется.
 */

// Возвращает модуль комплексного числа. Он определяется как расстояние
// на комплексной плоскости до числа от начала координат (0,0).
Complex.prototype.magnitude = function() {
    return Math.sqrt(this.x*this.x + this.y*this.y);

};

// Возвращает комплексное число с противоположным знаком.
Complex.prototype.negative = function() {
```

```
    return new Complex(-this.x, -this.y);
};

// Складывает данное комплексное число с заданным и возвращает
// сумму в виде нового объекта.
Complex.prototype.add = function(that) {
    return new Complex(this.x + that.x, this.y + that.y);
}

// Умножает данное комплексное число на заданное и возвращает
// произведение в виде нового объекта.
Complex.prototype.multiply = function(that) {
    return new Complex(this.x * that.x - this.y * that.y,
                       this.x * that.y + this.y * that.x);
}

// Преобразует объект Complex в строку в понятном формате.
// Вызывается, когда объект Complex используется как строка.
Complex.prototype.toString = function() {
    return "{" + this.x + "," + this.y + "}";
};

// Проверяет равенство данного комплексного числа с заданным.
Complex.prototype.equals = function(that) {
    return this.x == that.x && this.y == that.y;
}

// Возвращает вещественную часть комплексного числа.
// Эта функция вызывается, когда объект Complex рассматривается
// как числовое значение.
Complex.prototype.valueOf = function() { return this.x; }

/*
 * Третий шаг в определении класса – это определение методов класса,
 * констант и других необходимых свойств класса как свойств самой
 * функции-конструктора (а не как свойств объекта-прототипа
 * конструктора). Обратите внимание, что методы класса не используют
 * ключевое слово this, они работают только со своими аргументами.
 */

// Складывает два комплексных числа и возвращает результат.
Complex.add = function (a, b) {
    return new Complex(a.x + b.x, a.y + b.y);
};

// Умножает два комплексных числа и возвращает полученное произведение.
Complex.multiply = function(a, b) {
    return new Complex(a.x * b.x - a.y * b.y,
                       a.x * b.y + a.y * b.x);
};

// Несколько предопределенных комплексных чисел.
// Они определяются как свойства класса, в котором могут использоваться как "константы".
// (Хотя в JavaScript невозможно определить свойства, доступные только для чтения.)
Complex.ZERO = new Complex(0,0);
Complex.ONE = new Complex(1,0);
Complex.I = new Complex(0,1);
```

9.3.7. Частные члены

Одна из наиболее общих характеристик традиционных объектно-ориентированных языков программирования, таких как C++, заключается в возможности объявления частных (private) свойств класса, обращаться к которым можно только из методов этого класса и недоступных за пределами класса. Распространенная техника программирования, называемая *инкапсуляцией данных*, заключается в создании частных свойств и организации доступа к этим свойствам только через специальные методы чтения/записи. JavaScript позволяет имитировать такое поведение посредством замыканий (эта тема обсуждается в разделе 8.8), но для этого необходимо, чтобы методы доступа хранились в каждом экземпляре класса и по этой причине не могли наследоваться от объекта-прототипа.

Следующий фрагмент демонстрирует, как можно добиться этого. Он содержит реализацию объекта прямоугольника `Rectangle`, ширина и высота которого доступны и могут изменяться только путем обращения к специальным методам:

```
function ImmutableRectangle(w, h) {
    // Этот конструктор не создает свойства объекта, где может храниться
    // ширина и высота. Он просто определяет в объекте методы доступа
    // Эти методы являются замыканиями и хранят значения ширины и высоты
    // в своих цепочках областей видимости.
    this.getWidth = function() { return w; }
    this.getHeight = function() { return h; }
}

// Обратите внимание: класс может иметь обычные методы в объекте-прототипе.
ImmutableRectangle.prototype.area = function( ) {
    return this.getWidth( ) * this.getHeight( );
};
```

Первенство открытия этой методики (или, по крайней мере, первенство публикации), вообще говоря, принадлежит Дугласу Крокфорду (Douglas Crockford). Его обсуждение этой темы можно найти на странице *http://www.crockford.com/javascript/private.html*.

9.4. Общие методы класса Object

Когда в JavaScript определяется новый класс, некоторые из его методов следует считать предопределенными. Эти методы подробно описываются в следующих подразделах.

9.4.1. Метод toString()

Идея метода `toString()` состоит в том, что каждый класс объектов должен иметь собственное особое строковое представление и поэтому определять соответствующий метод `toString()` для преобразования объектов в строковую форму. То есть определяя класс, необходимо определить для него специальный метод `toString()`, чтобы экземпляры класса могли быть преобразованы в осмысленные строки. Строка должна содержать информацию о преобразуемом объекте, т. к. это может потребоваться для нужд отладки. Если способ преобразования в строку выбран правильно, он также может быть полезным в самих программах. Кроме того,

можно создать собственную реализацию статического метода parse() для преобразования строки, возвращаемой методом toString(), обратно в форму объекта.

Класс Complex из примера 9.2 уже содержит реализацию метода toString(), а в следующем фрагменте приводится возможная реализация метода toString() для класса Circle:

```
Circle.prototype.toString = function () {
    return "[Круг радиуса " + this.r + " с центром в точке ("
        + this.x + ", " + this.y + ").]";
}
```

После определения такого метода toString() типичный объект Circle может быть преобразован в следующую строку:

```
"Круг радиуса 1 с центром в точке (0, 0)."
```

9.4.2. Метод valueOf()

Метод valueOf() во многом похож на метод toString(), но вызывается, когда JavaScript требуется преобразовать объект в значение какого-либо элементарного типа, отличного от строкового – обычно в число. Когда это возможно, функция должна возвращать элементарное значение, каким-либо образом представляющее значение объекта, на который ссылается ключевое слово this.

По определению объекты не являются элементарными значениями, поэтому большинство объектов не имеют эквивалентного элементарного типа. Вследствие этого метод valueOf(), определяемый по умолчанию классом Object, не выполняет преобразования, а просто возвращает объект, с которым он был вызван. Такие классы, как Number и Boolean, имеют очевидные элементарные эквиваленты, поэтому они переопределяют метод valueOf() так, чтобы он возвращал соответствующие значения. Именно поэтому объекты Number и Boolean могут вести себя во многом так же, как эквивалентные им элементарные значения.

Иногда можно определить класс, имеющий какой-то разумный элементарный эквивалент. В этом случае может потребоваться определить для этого класса специальный метод valueOf(). Если мы вернемся к примеру 9.2, то увидим, что метод valueOf() определен для класса Complex. Этот метод просто возвращает вещественную часть комплексного числа. Поэтому в числовом контексте объект Complex ведет себя так, как будто является вещественным числом без мнимой составляющей. Рассмотрим, например, следующий фрагмент:

```
var a = new Complex(5,4);
var b = new Complex(2,1);
var c = Complex.sum(a,b); // c это комплексное число {7,5}
var d = a + b;            // d это число 7
```

При наличии метода valueOf() следует соблюдать одну осторожность: в случае преобразования объекта в строку метод valueOf() иногда имеет приоритет перед методом toString(). Поэтому, когда для класса определен метод valueOf() и надо, чтобы объект этого класса был преобразован в строку, может потребоваться явно указать на это, вызвав метод toString(). Продолжим пример с классом Complex:

```
alert("c = " + c);            // Используется valueOf(); выводит "c = 7"
alert("c = " + c.toString()); // Выводит "c = {7,5}"
```

9.4.3. Методы сравнения

Операторы сравнения в JavaScript сравнивают объекты по ссылке, а не по значению. Так, если имеются две ссылки на объекты, то выясняется, ссылаются они на один и тот же объект или нет, но не выясняется, обладают ли разные объекты одинаковыми свойствами с одинаковыми значениями.[1] Часто бывает удобным иметь возможность выяснить эквивалентность объектов или даже определить порядок их следования (например, с помощью операторов отношения < и >). Если вы определяете класс и хотите иметь возможность сравнивать экземпляры этого класса, вам придется определить соответствующие методы, выполняющие сравнение.

В языке программирования Java сравнение объектов производится с помощью методов, и подобный подход можно с успехом использовать в JavaScript. Чтобы иметь возможность сравнивать экземпляры класса, можно определить метод экземпляра с именем `equals()`. Этот метод должен принимать единственный аргумент и возвращать `true`, если аргумент эквивалентен объекту, метод которого был вызван. Разумеется, вам решать, что следует понимать под словом «эквивалентен» в контексте вашего класса. Обычно для определения того, равны ли объекты, сравниваются значения свойств экземпляров двух объектов. Класс `Complex` из примера 9.2 как раз обладает таким методом `equals()`.

Иногда возникает необходимость реализовать операции сравнения, чтобы выяснить порядок следования объектов. Так, для некоторых классов вполне можно сказать, что один экземпляр «меньше» или «больше» другого. Например, порядок следования объектов класса `Complex` определяется на основе значения, возвращаемого методом `magnitude()`. В то же время для объектов класса `Circle` сложно определить смысл слов «меньше» и «больше» – следует ли сравнивать величину радиуса или нужно сравнивать координаты X и Y? А может быть, следует учитывать величины всех трех параметров?

При попытке сравнения JavaScript-объектов с помощью операторов отношения, таких как < и <=, интерпретатор сначала вызовет методы `valueOf()` объектов, и если методы вернут значения элементарных типов, сравнит эти значения. Поскольку класс `Complex` имеет метод `valueOf()`, который возвращает вещественную часть комплексного числа, экземпляры класса `Complex` можно сравнивать как обычные вещественные числа, не имеющие мнимой части.[2] Это может совпадать или не совпадать с вашими намерениями. Чтобы сравнивать объекты для выяснения порядка их следования по вашему выбору, вам необходимо (опять же, следуя соглашениям, принятым в языке программирования Java) реализовать метод с именем `compareTo()`.

Метод `compareTo()` должен принимать единственный аргумент и сравнивать его с объектом, метод которого был вызван. Если объект `this` меньше, чем объект,

---

[1] То есть являются эквивалентными копиями-экземплярами одного класса. – *Примеч. науч. ред.*

[2] При этом получается результат, очень странный для всякого, кто работает в областях применения комплексной математики. Это хороший пример того, как скоропалительное определение метода `valueOf()` (да и любого метода, особенно из числа базовых) в дальнейшем может преподносить пользователю большие сюрпризы, не согласующиеся с его логикой восприятия. – *Примеч. науч. ред.*

представленный аргументом, метод compareTo() должен возвращать значение меньше нуля. Если объект this больше, чем объект, представленный аргументом, метод должен возвращать значение больше нуля. И если оба объекта равны, метод должен возвращать значение, равное нулю. Эти соглашения о возвращаемом значении весьма важны, потому что позволяют выполнять замену операторов отношения следующими выражениями:

Выражение отношения	Выражение замены
a < b	a.compareTo(b) < 0
a <= b	a.compareTo(b) <= 0
a > b	a.compareTo(b) > 0
a >= b	a.compareTo(b) >= 0
a == b	a.compareTo(b) == 0
a != b	a.compareTo(b) != 0

Вот одна из возможных реализаций метода compareTo() для класса Complex из примера 9.2, в которой сравниваются комплексные числа по их модулям:

```
Complex.prototype.compareTo = function(that) {
    // Если аргумент не был передан или он не имеет метода
    // magnitude(), необходимо сгенерировать исключение.
    // Как вариант - можно было бы вернуть значение -1 или 1,
    // чтобы как-то обозначить, что комплексное число всегда меньше
    // или больше, чем любое другое значение.
    if (!that || !that.magnitude || typeof that.magnitude != "function")
        throw new Error("неверный аргумент в Complex.compareTo()");

    // Здесь используется свойство операции вычитания, которая
    // возвращает значение меньшее, большее или равное нулю.
    // Этот прием можно использовать во многих реализациях метода compareTo().
    return this.magnitude() - that.magnitude();
}
```

Одна из причин, по которым может потребоваться сравнивать экземпляры класса, – возможность сортировки массива этих экземпляров в некотором порядке. Метод Array.sort() может принимать в виде необязательного аргумента функцию сравнения, которая должна следовать тем же соглашениям о возвращаемом значении, что и метод compareTo(). При наличии метода compareTo() достаточно просто организовать сортировку массива комплексных чисел примерно следующим образом:

```
complexNumbers.sort(new function(a,b) { return a.compareTo(b); });
```

Сортировка имеет большое значение, и потому следует рассмотреть возможность реализации статического метода compare() в любом классе, где определен метод экземпляра compareTo(). Особенно если учесть, что первый может быть легко реализован в терминах второго, например:

```
Complex.compare = function(a,b) { return a.compareTo(b); };
```

При наличии этого метода сортировка массива может быть реализована еще проще:

```
complexNumbers.sort(Complex.compare);
```

Обратите внимание: реализации методов `compare()` и `compareTo()` не были включены в определение класса `Complex` из примера 9.2. Дело в том, что они не согласуются с методом `equals()`, который был определен в этом примере. Метод `equals()` утверждает, что два объекта класса `Complex` эквивалентны, если их вещественные и мнимые части равны. Однако метод `compareTo()` возвращает нулевое значение для любых двух комплексных чисел, которые имеют равные модули. Числа 1+0i и 0+1i имеют одинаковые модули и эти два числа будут объявлены равными при вызове метода `compareTo()`, но метод `equals()` утверждает, что они не равны. Таким образом, если вы собираетесь реализовать методы `equals()` и `compareTo()` в одном и том же классе, будет совсем нелишним их как-то согласовать. Несогласованность в понимании термина «равенство» может стать источником пагубных ошибок. Рассмотрим реализацию метода `compareTo()`, который согласуется с существующим методом `equals()`:[1]

```
// При сравнении комплексных чисел в первую очередь сравниваются
// их вещественные части. Если они равны, сравниваются мнимые части
Complex.prototype.compareTo = function(that) {
    var result = this.x - that.x; // Сравнить вещественные части
                                  // с помощью операции вычитания
    if (result == 0)              // Если они равны...
        result = this.y - that.y; // тогда сравнить мнимые части

    // Теперь результат будет равен нулю только в том случае,
    // если равны и вещественные, и мнимые части
    return result;
};
```

9.5. Надклассы и подклассы

В Java, C++ и других объектно-ориентированных языках на базе классов имеется явная концепция *иерархии классов*. Каждый класс может иметь *надкласс*, от которого он наследует свойства и методы. Любой класс может быть расширен, т. е. иметь *подкласс*, наследующий его поведение. Как мы видели, JavaScript поддерживает наследование прототипов вместо наследования на базе классов. Тем не менее в JavaScript могут быть проведены аналогии с иерархией классов. В JavaScript класс `Object` – это наиболее общий класс, и все другие классы являются его специализированными версиями, или подклассами. Можно также сказать, что `Object` – это надкласс всех встроенных классов. Все классы наследуют несколько базовых методов класса `Object`.

Мы узнали, что объекты наследуют свойства от объекта-прототипа их конструктора. Как они могут наследовать свойства еще и от класса `Object`? Вспомните, что объект-прототип сам представляет собой объект; он создается с помощью конст-

[1] Но при таком определении достаточно «странную» семантику обретают операторы отношения < и >. – *Примеч. науч. ред.*

руктора Object(). Это значит, что объект-прототип наследует свойства от Object.prototype! Поэтому объект класса Complex наследует свойства от объекта Complex.prototype, который в свою очередь наследует свойства от Object.prototype. Когда выполняется поиск некоторого свойства в объекте Complex, сначала выполняется поиск в самом объекте. Если свойство не найдено, поиск продолжается в объекте Complex.prototype. И наконец, если свойство не найдено и в этом объекте, выполняется поиск в объекте Object.prototype.

Обратите внимание: поскольку в объекте-прототипе Complex поиск происходит раньше, чем в объекте-прототипе Object, свойства объекта Complex.prototype скрывают любые свойства с тем же именем из Object.prototype. Так, в классе, показанном в примере 9.2, мы определили в объекте Complex.prototype метод toString(). Object.prototype также определяет метод с этим именем, но объекты Complex никогда не увидят его, поскольку определение toString() в Complex.prototype будет найдено раньше.

Все классы, которые мы показали в этой главе, представляют собой непосредственные подклассы класса Object. Это типично для программирования на JavaScript; обычно в создании более сложной иерархии классов нет никакой необходимости. Однако когда это требуется, можно создать подкласс любого другого класса. Предположим, что мы хотим создать подкласс класса Rectangle, чтобы добавить в него свойства и методы, связанные с координатами прямоугольника. Для этого мы просто должны быть уверены, что объект-прототип нового класса сам является экземпляром Rectangle и потому наследует все свойства Rectangle.prototype. Пример 9.3 повторяет определение простого класса Rectanle и затем расширяет это определение за счет создания нового класса PositionedRectangle.

Пример 9.3. Создание подкласса в JavaScript

```
// Определение простого класса прямоугольников.
// Этот класс имеет ширину и высоту и может вычислять свою площадь
function Rectangle(w, h) {
    this.width = w;
    this.height = h;
}
Rectangle.prototype.area = function( ) { return this.width * this.height; }

// Далее идет определение подкласса
function PositionedRectangle(x, y, w, h) {
    // В первую очередь необходимо вызвать конструктор надкласса
    // для инициализации свойств width и height нового объекта.
    // Здесь используется метод call, чтобы конструктор был вызван
    // как метод инициализируемого объекта.
    // Это называется вызов конструктора по цепочке.
    Rectangle.call(this, w, h);

    // Далее сохраняются координаты верхнего левого угла прямоугольника
    this.x = x;
    this.y = y;
}

// Если мы будем использовать объект-прототип по умолчанию,
// который создается при определении конструктора PositionedRectangle(),
// был бы создан подкласс класса Object.
```

```
// Чтобы создать подкласс класса Rectangle, необходимо явно создать объект-прототип.
PositionedRectangle.prototype = new Rectangle();

// Мы создали объект-прототип с целью наследования, но мы не собираемся
// наследовать свойства width и height, которыми обладают все объекты
// класса Rectangle, поэтому удалим их из прототипа.
delete PositionedRectangle.prototype.width;
delete PositionedRectangle.prototype.height;

// Поскольку объект-прототип был создан с помощью конструктора
// Rectangle(), свойство constructor в нем ссылается на этот
// конструктор. Но нам нужно, чтобы объекты PositionedRectangle
// ссылались на другой конструктор, поэтому далее выполняется
// присваивание нового значения свойству constructor
PositionedRectangle.prototype.constructor = PositionedRectangle;

// Теперь у нас имеется правильно настроенный прототип для нашего
// подкласса, можно приступать к добавлению методов экземпляров.
PositionedRectangle.prototype.contains = function(x,y) {
    return (x > this.x && x < this.x + this.width &&
            y > this.y && y < this.y + this.height);
}
```

Как видно из примера 9.3, создание подклассов в JavaScript выглядит более сложным, чем наследование от класса `Object`. Первая проблема связана с необходимостью вызова конструктора надкласса из конструктора подкласса, причем конструктор надкласса приходится вызывать как метод вновь созданного объекта. Затем приходится хитрить и подменять конструктор объекта-прототипа подкласса. Нам потребовалось явно создать этот объект-прототип как экземпляр надкласса, после чего надо было явно изменить свойство `constructor` объекта-прототипа.[1] Может также появиться желание удалить любые свойства, которые создаются конструктором надкласса в объекте-прототипе, поскольку очень важно, чтобы свойства объекта-прототипа наследовались из *его* прототипа.

Имея такое определение класса `PositionedRectangle`, его можно использовать в своих программах примерно так:

```
var r = new PositionedRectangle(2,2,2,2);
print(r.contains(3,3)); // Вызывается метод экземпляра
print(r.area( ));       // Вызывается унаследованный метод экземпляра

// Работа с полями экземпляра класса:
print(r.x + ", " + r.y + ", " + r.width + ", " + r.height);

// Наш объект может рассматриваться как экземпляр всех 3 классов
```

[1] В версии Rhino 1.6r1 и более ранних (интерпретатор JavaScript, написанный на языке Java) имеется ошибка, которая делает свойство `constructor` неудаляемым и доступным только для чтения. В этих версиях Rhino в программном коде, выполняющем настройку свойства `constructor`, происходит сбой без вывода сообщений об ошибке. В результате экземпляры класса PositionedRectangle наследуют значение свойства `constructor`, которое ссылается на конструктор `Rectangle()`. На практике эта ошибка почти не проявляется, потому что свойства наследуются правильно и оператор `instanceof` корректно различает экземпляры классов `PositionedRectangle` и `Rectangle`.

```
print(r instanceof PositionedRectangle &&
      r instanceof Rectangle &&
      r instanceof Object);
```

9.5.1. Изменение конструктора

В только что продемонстрированном примере функция-конструктор `PositionedRectangle()` должна явно вызывать функцию-конструктор надкласса. Это называется *вызовом конструктора по цепочке* и является обычной практикой при создании подклассов. Вы можете упростить синтаксис конструктора, добавив свойство `superclass` в объект-прототип подкласса:

```
// Сохранить ссылку на конструктор надкласса.
PositionedRectangle.prototype.superclass = Rectangle;
```

Однако следует заметить, что такой прием можно использовать только при условии неглубокой иерархии наследования. Так, если класс `B` является наследником класса `A`, а класс `C` – наследником класса `B`, и в обоих классах `B` и `C` используется прием с обращением к свойству `superclass`, то при попытке создать экземпляр класса `C` ссылка `this.superclass` будет указывать на конструктор `B()`, что в результате приведет к бесконечному рекурсивному зацикливанию конструктора `B()`. Поэтому для всего, что не является простым подклассом, используйте методику вызова конструктора по цепочке, которая продемонстрирована в примере 9.3.

После того как свойство определено, синтаксис вызова конструктора по цепочке становится значительно проще:

```
function PositionedRectangle(x, y, w, h) {
    this.superclass(w,h);
    this.x = x;
    this.y = y;
}
```

Обратите внимание: функция-конструктор явно вызывается в контексте объекта `this`. Это означает, что можно отказаться от использования метода `call()` или `apply()` для вызова конструктора надкласса как метода данного объекта.

9.5.2. Вызов переопределенных методов

Когда в подклассе определяется метод, имеющий то же самое имя, что и метод надкласса, подкласс *переопределяет* (*overrides*) этот метод. Ситуация, когда подкласс порождается от существующего класса, встречается достаточно часто. Например, в любой момент можно определить метод `toString()` класса и тем самым переопределить метод `toString()` класса `Object`.

Зачастую переопределение методов производится не с целью полной замены, а лишь для того, чтобы расширить их функциональность. Для этого метод должен иметь возможность вызывать переопределенный метод. В определенном смысле такой прием по аналогии с конструкторами можно назвать вызовом методов по цепочке. Однако вызвать переопределенный метод гораздо менее удобно, чем конструктор надкласса.

Рассмотрим следующий пример. Предположим, что класс `Rectangle` определяет метод `toString()` (что должно быть сделано чуть ли не в первую очередь) следующим образом:

```
Rectangle.prototype.toString = function( ) {
    return "[" + this.width + "," + this.height + "]";
}
```

Если уж вы реализовали метод `toString()` в классе `Rectangle`, то тем более его необходимо переопределить в классе `PositionedRectangle`, чтобы экземпляры подкласса могли иметь строковое представление, отражающее значения не только ширины и высоты, но и всех остальных их свойств. `PositionedRectangle` – очень простой класс и для него достаточно, чтобы метод `toString()` просто возвращал значения всех его свойств. Однако ради примера будем обрабатывать значения свойств координат в самом классе, а обработку свойств `width` и `height` делегируем надклассу. Сделать это можно примерно следующим образом:

```
PositionedRectangle.prototype.toString = function() {
    return "(" + this.x + "," + this.y + ") " + // поля этого класса
    Rectangle.prototype.toString.apply(this);   // вызов надкласса по цепочке
}
```

Реализация метода `toString()` надкласса доступна как свойство объекта-прототипа надкласса. Обратите внимание: мы не можем вызвать метод напрямую – нам пришлось воспользоваться методом `apply()`, чтобы указать, для какого объекта вызывается метод.

Однако если в `PositionedRectangle.prototype` добавить свойство `superclass`, можно сделать так, чтобы этот код не зависел от типа надкласса:

```
PositionedRectangle.prototype.toString = function( ) {
    return "(" + this.x + "," + this.y + ") " + // поля этого класса
        this.superclass.prototype.toString.apply(this);
}
```

Еще раз обратите внимание, что свойство `superclass` может использоваться в иерархии наследования только один раз. Если оно будет задействовано классом и его подклассом, это приведет к бесконечной рекурсии.

9.6. Расширение без наследования

Предыдущее обсуждение проблемы создания подклассов описывает порядок создания новых классов, наследующих методы других классов. Язык JavaScript настолько гибкий, что создание подклассов и использование механизма наследования – это не единственный способ расширения функциональных возможностей классов. Поскольку функции в JavaScript – это просто значения данных, они могут легко копироваться (или «заимствоваться») из одного класса в другой. В примере 9.4 демонстрируется функция, которая заимствует все методы одного класса и создает их копии в объекте-прототипе другого класса.

Пример 9.4. Заимствование методов одного класса для использования в другом

```
// Заимствование методов одного класса для использования в другом.
// Аргументы должны быть функциями-конструкторами классов.
// Методы встроенных типов, таких как Object, Array, Date и RegExp
// не являются перечислимыми и потому не заимствуются этой функцией.
function borrowMethods(borrowFrom, addTo) {
    var from = borrowFrom.prototype; // прототип-источник
```

```
    var to = addTo.prototype;        // прототип-приемник

    for(m in from) { // Цикл по всем свойствам прототипа-источника
        if (typeof from[m] != "function") continue; // Игнорировать все,
                                                    // что не является функциями
            to[m] = from[m];                        // Заимствовать метод
    }
}
```

Многие методы настолько тесно связаны с классом, в котором они определены, что нет смысла пытаться использовать их в другом классе. Однако некоторые методы могут быть достаточно универсальными и пригодяться в любом классе. В примере 9.5 приводятся определения двух классов, ничего особо полезного не делающих, зато реализующих методы, которые могут быть заимствованы другими классами. Подобные классы, разрабатываемые специально с целью заимствования, называются *классами-смесями*, или просто *смесями*.

Пример 9.5. Классы-смеси с универсальными методами, предназначенными для заимствования

```
// Сам по себе этот класс не очень хорош. Но он определяет универсальный
// метод toString(), который может представлять интерес для других классов.
function GenericToString() {}
GenericToString.prototype.toString = function( ) {
    var props = [];
    for(var name in this) {
        if (!this.hasOwnProperty(name)) continue;
        var value = this[name];
        var s = name + ":"
        switch(typeof value) {
        case 'function':
            s += "function";
            break;
        case 'object':
            if (value instanceof Array) s += "array"
            else s += value.toString( );
            break;
        default:
            s += String(value);
            break;
        }
        props.push(s);
    }
    return "{" + props.join(", ") + "}";
}

// Следующий класс определяет метод equals(), который сравнивает простые объекты.
function GenericEquals() {}
GenericEquals.prototype.equals = function(that) {
    if (this == that) return true;

    // объекты равны, только если объект this имеет те же свойства,
    // что и объект that, и не имеет никаких других свойств
    // Обратите внимание: нам не требуется глубокое сравнение.
    // Значения просто должны быть === друг другу. Из этого следует,
```

```
    // если есть свойства, ссылающиеся на другие объекты, они должны ссылаться
    // на те же самые объекты, а не на объекты, для которых equals() возвращает true
    var propsInThat = 0;
    for(var name in that) {
        propsInThat++;
        if (this[name] !== that[name]) return false;
    }

    // Теперь необходимо убедиться, что объект this не имеет дополнительных свойств
    var propsInThis = 0;
    for(name in this) propsInThis++;

    // Если объект this обладает дополнительными свойствами,
    // следовательно, объекты не равны
    if (propsInThis != propsInThat) return false;

    // Два объекта выглядят равными.
    return true;
}
```

Вот как выглядит простой класс `Rectangle`**, который заимствует методы** `toString()` **и** `equals()`**, определенные в классах-смесях:**

```
// Простой класс Rectangle
function Rectangle(x, y, w, h) {
    this.x = x;
    this.y = y;
    this.width = w;
    this.height = h;
}
Rectangle.prototype.area = function( ) { return this.width * this.height; }

// Заимствование некоторых методов
borrowMethods(GenericEquals, Rectangle);
borrowMethods(GenericToString, Rectangle);
```

Ни один из представленных здесь классов-смесей не имеет собственного конструктора, однако это не значит, что конструкторы нельзя заимствовать. В следующем фрагменте приводится определение нового класса с именем `ColoredRectangle`**. Он наследует функциональность класса** `Rectangle` **и заимствует конструктор и метод из класса-смеси** `Colored`**:**

```
// Эта смесь содержит метод, зависящий от конструктора. Оба они,
// и конструктор, и метод должны быть заимствованы.
function Colored(c) { this.color = c; }
Colored.prototype.getColor = function() { return this.color; }

// Определение конструктора нового класса
function ColoredRectangle(x, y, w, h, c) {
    this.superclass(x, y, w, h); // Вызов конструктора надкласса
    Colored.call(this, c);       // и заимствование конструктора Colored
}

// Настройка объекта-прототипа на наследование методов от Rectangle
ColoredRectangle.prototype = new Rectangle();
ColoredRectangle.prototype.constructor = ColoredRectangle;
ColoredRectangle.prototype.superclass = Rectangle;
```

```
// Заимствовать методы класса Colored в новый класс
borrowMethods(Colored, ColoredRectangle);
```

Класс `ColoredRectangle` расширяет класс `Rectangle` (и наследует его методы), а также заимствует методы класса `Colored`. Сам класс `Rectangle` наследует класс `Object` и заимствует методы классов `GenericEquals` и `GenericToString`. Хотя подобные аналогии здесь неуместны, можно воспринимать это как своего рода множественное наследование. Так как класс `ColoredRectangle` заимствует методы класса `Colored`, экземпляры класса `ColoredRectangle` можно одновременно рассматривать как экземпляры класса `Colored`. Оператор `instanceof` не сможет сообщить об этом, но в разделе 9.7.3 мы создадим более универсальный метод, который позволит определять, наследует или заимствует некоторый объект методы заданного класса.

9.7. Определение типа объекта

Язык JavaScript – это слабо типизированный язык, а JavaScript-объекты еще менее типизированы. Тем не менее в JavaScript существует несколько приемов, которые могут служить для определения типа произвольного значения.

Конечно же, самый распространенный прием основан на использовании оператора `typeof` (подробности см. в разделе 5.10.2). В первую очередь `typeof` позволяет различать объекты и элементарные типы, однако он обладает некоторыми странностями. Во-первых, выражение `typeof null` дает в результате строку `"object"`, тогда как выражение `typeof undefined` возвращает строку `"undefined"`. Помимо этого в качестве типа любого массива возвращается строка `"object"`, поскольку все массивы – объекты, однако для любой функции возвращается строка `"function"`, хотя функции фактически также являются объектами.

9.7.1. Оператор instanceof и конструктор

После того как выяснится, что некоторое значение является объектом, а не элементарным значением и не функцией, его можно передать оператору `instanceof`, чтобы подробнее выяснить его природу. Например, если `x` является массивом, тогда следующее выражение вернет `true`:

```
x instanceof Array
```

Слева от оператора `instanceof` располагается проверяемое значение, справа – имя функции-конструктора, определяющей класс объектов. Обратите внимание: объект расценивается как экземпляр собственного класса и всех его надклассов. Таким образом, для любого объекта `o` выражение `o instanceof Object` всегда вернет `true`. Интересно, что оператор `instanceof` может работать и с функциями, так, все нижеследующие выражения возвращают значение `true`:

```
typeof f == "function"
f instanceof Function
f instanceof Object
```

В случае необходимости можно убедиться, что некоторый объект является экземпляром определенного класса, а не одного из подклассов – для этого достаточно проверить значение свойства `constructor`. В следующем фрагменте выполняется такая проверка:

```
var d = new Date();   // Объект Date; Date - подкласс Object
```

```
var isobject = d instanceof Object;     // Возвращает true
var realobject = d.constructor==Object; // Возвращает false
```

9.7.2. Определение типа объекта с помощью метода Object.toString()

Недостаток оператора `instanceof` и свойства `constructor` заключается в том, что они позволяют проверять объекты на принадлежность только известным вам классам, но не дают никакой полезной информации при исследовании неизвестных объектов, что может потребоваться, например, при отладке. В такой ситуации на помощь может прийти метод `Object.toString()`.

Как уже говорилось в главе 7, класс `Object` содержит определение метода `toString()` по умолчанию. Любой класс, который не определяет собственный метод, наследует реализацию по умолчанию. Интересная особенность метода по умолчанию `toString()` состоит в том, что он выводит некоторую внутреннюю информацию о типе встроенных объектов. Спецификация ECMAScript требует, чтобы метод по умолчанию `toString()` всегда возвращал строку в формате:

```
[object class]
```

Здесь *class* – это внутренний тип объекта, который обычно соответствует имени функции-конструктора этого объекта. Например, для массивов *class* – это `"Array"`, для функций – `"Function"`, и для объектов даты/времени – `"Date"`. Для встроенного класса `Math` возвращается `"Math"`, а для всех классов семейства `Error` – строка `"Error"`. Для объектов клиентского языка JavaScript и любых других объектов, определяемых реализацией JavaScript, в качестве строки *class* возвращается строка, определяемая реализацией (например, `"Window"`, `"Document"` или `"Form"`). Однако для типов объектов, определяемых пользователем, таких как `Circle` и `Complex`, описанных ранее в этой главе, в качестве строки *class* всегда возвращается строка `"Object"`. То есть метод `toString()` способен определять только встроенные типы объектов.

Поскольку в большинстве классов метод по умолчанию `toString()` переопределяется, не следует ожидать, что вызвав его непосредственно из объекта, вы получите имя класса. Поэтому необходимо обращаться к функции по умолчанию `Object.prototype` явно и использовать для этого метод `apply()` с указанием объекта, тип которого требуется узнать:

```
Object.prototype.toString.apply(o); // Всегда вызывается метод по умолчанию toString()
```

Этот прием используется в примере 9.6 в определении функции, реализующей расширенные возможности по выяснению типа. Как уже отмечалось ранее, метод `toString()` не работает с пользовательскими классами, в этом случае показанная далее функция проверяет строковое значение свойства `classname` и возвращает его значение, если оно определено.

Пример 9.6. Улучшенные возможности определения типа

```
function getType(x) {
    // Если значение x равно null, возвращается "null"
    if (x == null) return "null";

    // Попробовать определить тип с помощью оператора typeof
    var t = typeof x;
```

```
    // Если получен непонятный результат, вернуть его
    if (t != "object") return t;

    // В противном случае, x - это объект. Вызвать метод toString()
    // по умолчанию и извлечь подстроку с именем класса.
    var c = Object.prototype.toString.apply(x); // В формате "[object class]"
    c = c.substring(8, c.length-1);             // Удалить "[object" и "]"

    // Если имя класса - не Object, вернуть его.
    if (c != "Object") return c;

    // Если получен тип "Object", проверить, может быть x
    // действительно принадлежит этому классу.
    if (x.constructor == Object) return c;       // Тип действительно "Object"

    // Для пользовательских классов извлечь строковое значение свойства
    // classname, которое наследуется от объекта-прототипа
    if ("classname" in x.constructor.prototype &&              // наследуемое имя класса
        typeof x.constructor.prototype.classname == "string")  // это строка
        return x.constructor.prototype.classname;

    // Если определить тип так и не удалось, так и скажем об этом.
    return "<unknown type>";
}
```

9.7.3. Грубое определение типа

Существует старое высказывание: «Если оно ходит как утка и крякает как утка, значит, это утка!». Перевести этот афоризм на язык JavaScript довольно сложно, однако попробуем: «Если в этом объекте реализованы все методы некоторого класса, значит, это экземпляр данного класса». В гибких языках программирования со слабой типизацией, таких как JavaScript, это называется «грубым определением типа»: если объект обладает всеми свойствами класса `X`, его можно рассматривать как экземпляр класса `X`, даже если на самом деле этот объект не был создан с помощью функции-конструктора `X()`.[1]

Грубое определение типа особенно удобно использовать для классов, «заимствующих» методы у других классов. Ранее в этой главе демонстрировался класс `Rectangle`, заимствующий метод `equals()` у класса с именем `GenericEquals`. В результате любой экземпляр класса `Rectangle` можно рассматривать как экземпляр класса `GenericEquals`. Оператор `instanceof` не может определить этот факт, но в наших силах создать для этого собственный метод (пример 9.7).

Пример 9.7. Проверка факта заимствования объектом методов заданного класса

```
// Возвращает true, если каждый из методов c.prototype был
// заимствован объектом o. Если o - это функция, а не объект,
// вместо самого объекта o производится проверка его прототипа.
// Обратите внимание: для этой функции необходимо, чтобы методы были
// скопированы, а не реализованы повторно. Если класс заимствовал метод,
// а затем переопределил его, данная функция вернет значение false.
function borrows(o, c) {
```

[1] Термин «грубое определение типа» появился благодаря языку программирования Ruby. Точное его название – алломорфизм.

```
    // Если объект o уже является экземпляром класса c, можно вернуть true
    if (o instanceof c) return true;

    // Совершенно невозможно выполнить проверку факта заимствования методов
    // встроенного класса, поскольку методы встроенных типов неперечислимы.
    // В этом случае вместо того, чтобы генерировать, исключение возвращается
    // значение undefined, как своего рода ответ "Я не знаю".
    // Значение undefined ведет себя во многом похоже на false,
    // но может отличаться от false, если это потребуется вызывающей программе.
    if (c == Array || c == Boolean || c == Date || c == Error ||
        c == Function || c == Number || c == RegExp || c == String)
        return undefined;

    if (typeof o == "function") o = o.prototype;
    var proto = c.prototype;
    for(var p in proto) {
        // Игнорировать свойства, не являющиеся функциями
        if (typeof proto[p] != "function") continue;
        if (o[p] != proto[p]) return false;
    }
    return true;
}
```

Метод `borrows()` из примера 9.7 достаточно ограничен: он возвращает значение `true`, только если объект `o` имеет точные копии методов, определяемых классом `c`. В действительности грубое определение типа должно работать более гибко: объект `o` должен рассматриваться как экземпляр класса `c`, если содержит методы, напоминающие методы класса `c`. В JavaScript «напоминающие» означает «имеющие те же самые имена» и (возможно) «объявленные с тем же количеством аргументов». В примере 9.8 демонстрируется метод, реализующий такую проверку.

Пример 9.8. Проверка наличия одноименных методов

```
// Возвращает true, если объект o обладает методами с теми же именами
// и количеством аргументов, что и класс c.prototype. В противном случае
// возвращается false. Генерирует исключение, если класс c принадлежит
// встроенному типу с методами, не поддающимися перечислению.
function provides(o, c) {
    // Если o уже является экземпляром класса c, он и так будет "напоминать" класс c
    if (o instanceof c) return true;

    // Если вместо объекта был передан конструктор объекта, использовать объект-прототип
    if (typeof o == "function") o = o.prototype;

    // Методы встроенных классов не поддаются перечислению, поэтому
    // возвращается значение undefined. В противном случае любой объект
    // будет напоминать любой из встроенных типов.
    if (c == Array || c == Boolean || c == Date || c == Error ||
        c == Function || c == Number || c == RegExp || c == String)
        return undefined;

    var proto = c.prototype;
    for(var p in proto) { // Цикл по всем свойствам в c.prototype
        // Игнорировать свойства, не являющиеся функциями
        if (typeof proto[p] != "function") continue;
        // Если объект o не имеет одноименного свойства, вернуть false
        if (!(p in o)) return false;
```

```
        // Если это свойство, а не функция, вернуть false
        if (typeof o[p] != "function") return false;
        // Если две функции объявлены с разным числом аргументов, вернуть false.
        if (o[p].length != proto[p].length) return false;
    }
    // Если были проверены все методы, можно смело возвращать true.
    return true;
}
```

В качестве примера грубого определения типа и использования метода `provide()` рассмотрим метод `compareTo()`, описанный в разделе 9.4.3. Как правило, метод `compareTo()` не предназначен для заимствования, но иногда бывает желательно выяснить, обладают ли некоторые объекты возможностью сравнения с помощью метода `compareTo()`. С этой целью определим класс `Comparable`:

```
function Comparable( ) {}
Comparable.prototype.compareTo = function(that) {
    throw "Comparable.compareTo() - абстрактный метод. Не подлежит вызову!";
}
```

Класс `Comparable` является *абстрактным*: его методы не предназначены для вызова, он просто определяет прикладной интерфейс. Однако при наличии определения этого класса можно проверить, допускается ли сравнение двух объектов:

```
// Проверить, допускается ли сравнение объектов o и p
// Они должны принадлежать одному типу и иметь метод compareTo()
if (o.constructor == p.constructor && provides(o, Comparable)) {
    var order = o.compareTo(p);
}
```

Обратите внимание: обе функции, представленные в этом разделе, `borrows()` и `provides()`, возвращают значение `undefined`, если им передается объект одного из встроенных типов JavaScript, например `Array`. Сделано это по той простой причине, что свойства объектов-прототипов встроенных типов не поддаются перечислению в цикле `for/in`. Если бы функции не могли выполнять проверку на принадлежность встроенным типам и возвращать `undefined`, тогда обнаружилось бы, что встроенные типы не имеют методов, и для них всегда возвращалось бы значение `true`.

Однако на типе `Array` следует остановиться особо. Вспомним, что в разделе 7.8 приводилась масса алгоритмов (таких как обход элементов массива), которые прекрасно работают с объектами, не являющимися настоящими массивами, а лишь подобными им. Метод грубого определения типа можно использовать для выяснения, является ли некоторый экземпляр объектом, напоминающим массив. Один из вариантов решения этой задачи приводится в примере 9.9.

Пример 9.9. Проверка объектов, напоминающих массивы

```
function isArrayLike(x) {
    if (x instanceof Array) return true;           // Настоящий массив
    if (!("length" in x)) return false;            // Массивы имеют свойство length
    if (typeof x.length != "number") return false; // Свойство length должно быть число,
   if (x.length < 0) return false;                 // причем неотрицательным
   if (x.length > 0) {
        // Если массив непустой, в нем как минимум должно быть свойство с именем length-1
```

```
            if (!((x.length-1) in x)) return false;
        }
        return true;
    }
```

9.8. Пример: вспомогательный метод defineClass()

Данная глава заканчивается определением вспомогательного метода `defineClass()`**, воплощающего в себе обсуждавшиеся темы о конструкторах, прототипах, подклассах, заимствовании и предоставлении методов. Реализация метода приводится в примере 9.10.**

Пример 9.10. Вспомогательная функция для определения классов

```
/**
 * defineClass() - вспомогательная функция для определения JavaScript-классов.
 *
 * Эта функция ожидает получить объект в виде единственного аргумента.
 * Она определяет новый JavaScript-класс, основываясь на данных в этом
 * объекте, и возвращает функцию-конструктор нового класса. Эта функция
 * решает задачи, связанные с определением классов: корректно устанавливает
 * наследование в объекте-прототипе, копирует методы из других классов и пр.
 *
 * Объект, передаваемый в качестве аргумента, должен иметь все
 * или некоторые из следующих свойств:
 *
 * name:      Имя определяемого класса.
 *            Если определено, это имя сохранится в свойстве classname объекта-прототипа.
 *
 * extend:    Конструктор наследуемого класса. В случае отсутствия будет
 *            использован конструктор Object(). Это значение сохранится
 *            в свойстве superclass объекта-прототипа.
 *
 * construct: Функция-конструктор класса. В случае отсутствия будет использована новая
 *            пустая функция. Это значение станет возвращаемым значением функции,
 *            а также сохранится в свойстве constructor объекта-прототипа.
 *
 * methods:   Объект, который определяет методы (и другие свойства,
 *            совместно используемые разными экземплярами) экземпляра класса.
 *            Свойства этого объекта будут скопированы в объект-прототип класса.
 *            В случае отсутствия будет использован пустой объект.
 *            Свойства с именами "classname", "superclass" и "constructor"
 *            зарезервированы и не должны использоваться в этом объекте.
 *
 * statics:   Объект, определяющий статические методы (и другие статические
 *            свойства) класса. Свойства этого объекта станут свойствами
 *            функции-конструктора. В случае отсутствия будет использован пустой объект.
 *
 * borrows:   Функция-конструктор или массив функций-конструкторов.
 *            Методы экземпляров каждого из заданных классов будут
 *            скопированы в объект-прототип этого нового класса, таким образом
 *            новый класс будет заимствовать методы каждого из заданных классов.
```

```
 *            Конструкторы обрабатываются в порядке их следования, вследствие
 *            этого методы классов, стоящих в конце массива, могут переопределить
 *            методы классов, стоящих выше.
 *            Обратите внимание: заимствуемые методы сохраняются
 *            в объекте-прототипе до того, как будут скопированы свойства
 *            и методы вышеуказанных объектов.
 *            Поэтому методы, определяемые этими объектами, могут
 *            переопределить заимствуемые. При отсутствии этого свойства
 *            заимствование методов не производится.
 *
 * provides:  Функция-конструктор или массив функций-конструкторов.
 *            После того как объект-прототип будет инициализирован, данная функция
 *            проверит, что прототип включает методы с именами и количеством
 *            аргументов, совпадающими с методами экземпляров указанных классов.
 *            Ни один из методов не будет скопирован, она просто убедится,
 *            что данный класс "предоставляет" функциональность, обеспечиваемую
 *            указанным классом. Если проверка окажется неудачной, данный метод
 *            сгенерирует исключение. В противном случае любой экземпляр нового класса
 *            может рассматриваться (с использованием методики грубого определения типа)
 *            как экземпляр указанных типов. Если данное свойство не определено,
 *            проверка выполняться не будет.
 **/
function defineClass(data) {
    // Извлечь значения полей из объекта-аргумента.
    // Установить значения по умолчанию.
    var classname = data.name;
    var superclass = data.extend || Object;
    var constructor = data.construct || function( ) {};
    var methods = data.methods || {};
    var statics = data.statics || {};
    var borrows;
    var provides;

    // Заимствование может производиться как из единственного конструктора,
    // так и из массива конструкторов.
    if (!data.borrows) borrows = [];
    else if (data.borrows instanceof Array) borrows = data.borrows;
    else borrows = [ data.borrows ];

    // То же для предоставляемых свойств.
    if (!data.provides) provides = [];
    else if (data.provides instanceof Array) provides = data.provides;
    else provides = [ data.provides ];

    // Создать объект, который станет прототипом класса.
    var proto = new superclass();

    // Удалить все неунаследованные свойства из нового объекта-прототипа.
    for(var p in proto)
        if (proto.hasOwnProperty(p)) delete proto[p];

    // Заимствовать методы из классов-смесей, скопировав их в прототип.
    for(var i = 0; i < borrows.length; i++) {
        var c = data.borrows[i];
        borrows[i] = c;
        // Скопировать методы из прототипа объекта c в наш прототип
```

```
            for(var p in c.prototype) {
                if (typeof c.prototype[p] != "function") continue;
                proto[p] = c.prototype[p];
            }
        }

        // Скопировать методы экземпляра в объект-прототип
        // Эта операция может переопределить методы, скопированные из классов-смесей
        for(var p in methods) proto[p] = methods[p];

        // Установить значения зарезервированных свойств "constructor",
        // "superclass" и "classname" в прототипе
        proto.constructor = constructor;
        proto.superclass = superclass;

        // Свойство classname установить, только если оно действительно задано.
        if (classname) proto.classname = classname;

        // Убедиться, что прототип предоставляет все предполагаемые методы.
        for(var i = 0; i < provides.length; i++) { // для каждого класса
            var c = provides[i];
            for(var p in c.prototype) {            // для каждого свойства
                if (typeof c.prototype[p] != "function") continue;  // только методы
                if (p == "constructor" || p == "superclass") continue;
                // Проверить наличие метода с тем же именем и тем же количеством
                // объявленных аргументов. Если метод имеется, продолжить цикл
                if (p in proto &&
                    typeof proto[p] == "function" &&
                    proto[p].length == c.prototype[p].length) continue;
                // В противном случае возбудить исключение
                throw new Error("Класс " + classname + " не предоставляет метод "+
                                c.classname + "." + p);
            }
        }

        // Связать объект-прототип с функцией-конструктором
        constructor.prototype = proto;

        // Скопировать статические свойства в конструктор
        for(var p in statics) constructor[p] = data.statics[p];

        // И в заключение вернуть функцию-конструктор
        return constructor;
    }
```

В примере 9.11 приводится фрагмент, который демонстрирует использование метода `defineClass()`**.**

Пример 9.11. Использование метода defineClass( )

```
// Класс Comparable с абстрактным методом, благодаря которому
// можно определить классы, "предоставляющие" интерфейс Comparable.
var Comparable = defineClass({
    name: "Comparable",
    methods: { compareTo: function(that) { throw "abstract"; } }
});

// Класс-смесь с универсальным методом equals() для заимствования
var GenericEquals = defineClass({
```

```
    name: "GenericEquals",
    methods: {
        equals: function(that) {
            if (this == that) return true;
            var propsInThat = 0;
            for(var name in that) {
                propsInThat++;
                if (this[name] !== that[name]) return false;
            }

            // Убедиться, что объект this не имеет дополнительных свойств
            var propsInThis = 0;
            for(name in this) propsInThis++;

            // Если имеются дополнительные свойства, объекты равны не будут
            if (propsInThis != propsInThat) return false;

            // Похоже, что два объекта эквивалентны.
            return true;
        }
    }
});

// Очень простой класс Rectangle, который предоставляет интерфейс Comparable
var Rectangle = defineClass({
    name: "Rectangle",
    construct: function(w,h) { this.width = w; this.height = h; },
    methods: {
        area: function() { return this.width * this.height; },
        compareTo: function(that) { return this.area( ) - that.area( ); }
    },
    provides: Comparable
});

// Подкласс класса Rectangle, который вызывает по цепочке конструктор своего
// надкласса, наследует методы надкласса, определяет свои методы экземпляра
// и статические методы и заимствует метод equals().
var PositionedRectangle = defineClass({
    name: "PositionedRectangle",
    extend: Rectangle,
    construct: function(x,y,w,h) {
        this.superclass(w,h); // вызов по цепочке
        this.x = x;
        this.y = y;
    },
    methods: {
        isInside: function(x,y) {
            return x > this.x && x < this.x+this.width &&
                   y > this.y && y < this.y+this.height;
        }
    },
    statics: {
        comparator: function(a,b) { return a.compareTo(b); }
    },
    borrows: [GenericEquals]
});
```

10

Модули и пространства имен

В первые годы после появления язык JavaScript чаще всего использовался для создания маленьких и простых сценариев, встроенных прямо в веб-страницы. По мере становления веб-броузеров и веб-стандартов программы на языке JavaScript становились все больше и все сложнее. В настоящее время многие JavaScript-сценарии используют в своей работе внешние *модули*, или библиотеки программного JavaScript-кода.[1]

К моменту написания этих строк ведутся работы по созданию модулей многократного использования, распространяемых с открытыми исходными текстами на языке JavaScript. Сеть архивов JavaScript (JavaScript Archive Network, JSAN) реализуется по образу и подобию всемирной сети архивов Perl (Comprehensive Perl Archive Network, CPAN), причем предполагается, что она станет для JavaScript тем же, чем стала CPAN для языка программирования и сообщества Perl. Подробную информацию о JSAN и примеры программного кода можно найти на сайте *http://www.openjsan.org*.

Язык JavaScript не предусматривает синтаксических конструкций, предназначенных для создания и управления модулями, поэтому написание переносимых модулей многократного использования на языке JavaScript в значительной степени является вопросом следования некоторым основным соглашениям, описываемым в этой главе.

Наиболее важное соглашение связано с концепцией *пространства имен*. Основная цель этой концепции – предотвратить конфликты имен, которые могут возникнуть при одновременном использовании двух модулей, объявляющих гло-

---

[1] В базовом языке JavaScript отсутствуют какие-либо механизмы загрузки или подключения внешних модулей. Эту задачу берет на себя окружение, в которое встраивается интерпретатор JavaScript. В клиентском языке JavaScript задача решается с использованием тега `<script src=>` (см. главу 13). Некоторые встраиваемые реализации предоставляют простейшую функцию `load()`, с помощью которой производится загрузка модулей.

бальные свойства с одинаковыми именами: один модуль может перекрыть свойства другого, что может привести к нарушениям в работе последнего.

Другое соглашение связано с порядком инициализации модуля. Это имеет важное значение для клиентского языка JavaScript, потому что в модули, которые манипулируют содержимым документа в веб-броузере, часто требуется встраивать программный код, запускаемый по окончании загрузки документа.

В следующих разделах обсуждаются вопросы организации пространств имен и инициализации. В конце главы приводится расширенный пример модуля вспомогательных функций, предназначенного для работы с модулями.

10.1. Создание модулей и пространств имен

Если возникает необходимость написать JavaScript-модуль, предназначенный для использования в любом сценарии или любым другим модулем, очень важно следовать правилу, согласно которому необходимо избегать объявления глобальных переменных. Всякий раз, когда объявляется глобальная переменная, возникает риск, что эта переменная будет переопределена другим модулем или программистом, применяющим этот модуль. Решение проблемы заключается в создании специально для данного модуля пространства имен и определении всех свойств и методов внутри этого пространства.

Язык JavaScript не обладает встроенной поддержкой пространств имен[1], но для этих целей прекрасно подходят JavaScript-объекты. Рассмотрим вспомогательные методы `provides()` и `defineClass()`, представленные в примерах 9.8 и 9.10 соответственно. Имена обоих методов являются глобальными символами. Если предполагается создать модуль функций для работы с JavaScript-классами, эти методы не должны объявляться в глобальном пространстве имен. С целью соблюдения этого соглашения, реализацию методов можно записать так:

```
// Создать пустой объект, который будет выполнять функции пространства имен
// Это единственное глобальное имя будет вмещать все остальные имена
var Class = {};
// Определить функции в пространстве имен
Class.define = function(data)   { /* здесь находится реализация метода */ }
Class.provides = function(o, c) { /* здесь находится реализация метода */ }
```

Обратите внимание: здесь не объявляются методы экземпляра (и даже не объявляются статические методы) JavaScript-класса. Здесь объявляются обычные функции, ссылки на которые сохраняются в свойствах специально созданного объекта.

Этот фрагмент иллюстрирует первое правило разработки JavaScript-модулей: *модуль никогда не должен выставлять больше одного имени в глобальном пространстве имен*. Существуют также два дополнения к этому правилу:

- Если модуль добавляет имя в глобальное пространство имен, документация к модулю должна четко и ясно отражать назначение этого имени.

---

[1] Как, например, определение `namespace` в языке C++ или одноименная встроенная команда в интерпретирующем языке Tcl (а это уже совсем близкий JavaScript язык). – *Примеч. науч. ред.*

- Если модуль добавляет имя в глобальное пространство имен, это имя должно быть однозначно связано с именем файла, из которого загружен модуль.

Так, если модуль называется `Class`, необходимо поместить его в файл с именем *Class.js*, а сам файл должен начинаться с комментария, который может выглядеть примерно следующим образом:

```
/**
 * Class.js: Модуль вспомогательных функций для работы с классами.
 *
 * Данный модуль определяет единственное глобальное имя "Class".
 * Имя Class является пространством имен объекта, а все функции
 * сохраняются как ссылки в свойствах этого пространства имен.
 **/
```

Классы в JavaScript имеют чрезвычайно важное значение, поэтому для работы с ними может существовать множество модулей. Что же произойдет, если найдутся такие два модуля, которые будут использовать имя `Class` для определения своих пространств имен? В этом случае произойдет конфликт имен. С помощью пространств имен можно существенно снизить риск появления конфликтов, но полностью свести риск к нулю невозможно. В этом отношении существенную помощь может оказать следование правилу именования файлов. Если оба конфликтующих модуля будут иметь одинаковое имя *Class.js*, их не удастся сохранить в одном каталоге. Загрузить оба модуля сценарий сможет только из разных каталогов, например *utilities/Class.js* и *flanagan/Class.js*.

А если сценарии сохраняются в подкаталогах, тогда имена подкаталогов должны являться частью имени модуля. Это означает, что модуль `Class`, определяемый здесь, в действительности должен называться `flanagan.Class`. Вот как это может быть реализовано на практике:

```
/**
 * flanagan/Class.js: Модуль вспомогательных функций для работы с классами.
 *
 * Данный модуль определяет единственное глобальное имя "flanagan",
 * если оно еще не существует. Затем создается объект пространства имен,
 * который сохраняется в свойстве Class объекта flanagan. Все вспомогательные
 * функции размещаются в пространстве имен flanagan.Class.
 **/
var flanagan; // Объявление единственного глобального имени "flanagan"
if (!flanagan) flanagan = {}; // Создается объект, если он еще не определен
flanagan.Class = {}           // Создается пространство имен flanagan.Class
// Теперь пространство имен заполняется вспомогательными методами
flanagan.Class.define = function(data) { /* реализация метода */ };
flanagan.Class.provides = function(o, c) { /* реализация метода */ };
```

В данном фрагменте глобальный объект `flanagan` является пространством имен для других пространств имен. Если, к примеру, я напишу другой модуль вспомогательных функций для работы с датами, то сохраню эти функции в пространстве имен `flanagan.Date`. Примечательно, что этот фрагмент объявляет глобальное имя `flanagan` с помощью инструкции `var` и лишь затем проверяет его наличие. Сделано это потому, что попытка чтения из необъявленной глобальной переменной приводит к генерации исключения, тогда как попытка чтения из объявленной, но не определенной переменной просто возвращает значение `unde-`

fined. Такое поведение характерно только для глобальных элементов. Если попытаться прочитать значение несуществующего свойства объекта пространства имен, будет просто получено значение undefined.

При наличии двухуровневых пространств имен вероятность конфликтов имен снижается еще больше. Однако если некоторый разработчик, тоже имеющий фамилиею Flanagan, решит написать модуль вспомогательных функций для работы с классами, программист, пожелавший использовать оба модуля, окажется в тупиковой ситуации. Хотя такой ход событий выглядит достаточно маловероятным, для полной уверенности можно попробовать следовать соглашению языка программирования Java, согласно которому для придания уникальности именам пакетов нужно использовать префиксы, начинающиеся с имени вашего домена в Интернете. При этом порядок следования имен доменов следует менять на обратный, чтобы имя домена верхнего уровня (*.com* или нечто подобное) стояло первым, и указывать получившееся имя в качестве префикса для всех ваших JavaScript-модулей. Поскольку мой сайт называется *davidflanagan.com*, я должен буду сохранить свои модули в файле с именем *com/davidflanagan/Class.js* и использовать пространство имен com.davidflanagan.Class. Если все JavaScript-разработчики будут следовать этому соглашению, никто не сможет создать пространство имен com.davidflanagan, поскольку только я владею доменом *davidflanagan.com*.

Это соглашение может оказаться излишним для большинства JavaScript-модулей, и вам не обязательно следовать ему в точности. Но вы должны знать о его существовании. Старайтесь не создавать по ошибке пространства имен, имена которых могут быть именем чьего-то домена: *никогда не определяйте пространства имен с использованием имен доменов, не являющихся вашей собственностью*.

Пример 10.1 демонстрирует порядок создания пространства имен com.davidflanagan.Class. Здесь добавлен код проверки ошибок, отсутствующий в предыдущем примере; в процессе этой проверки генерируется исключение, если пространство имен com.davidflanagan.Class уже существует или существуют пространства имен com или com.davidflanagan, но они не являются объектами. Здесь также демонстрируется, как можно создать и заполнить пространство имен с помощью единственного литерала объекта.

Пример 10.1. Создание пространства имен на основе имени домена

```
// Создать глобальный символ "com", если его еще не существует
// Генерировать исключение, если он существует, но не является объектом
var com;
if (!com) com = {};
else if (typeof com != "object")
    throw new Error("имя com существует, но не является объектом");

// Повторить процедуру создания и проверки типов на более низких уровнях
if (!com.davidflanagan) com.davidflanagan = {}
else if (typeof com.davidflanagan != "object")
    throw new Error("com.davidflanagan существует, но не является объектом");

// Генерировать исключение, если com.davidflanagan.Class уже существует
if (com.davidflanagan.Class)
    throw new Error("com.davidflanagan.Class уже существует");

// В противном случае создать и заполнить пространство имен
// с помощью одного большого литерала объекта
```

```
com.davidflanagan.Class = {
    define: function(data)   { /* здесь находится реализация функции */ },
    provides: function(o, c) { /* здесь находится реализация функции */ }
};
```

10.1.1. Проверка доступности модуля

Когда пишется программный код, использующий внешний модуль, узнать о его наличии можно, просто проверив, находится ли он в пространстве имен. Хитрость заключается в том, чтобы последовательно проверить наличие каждого компонента из этого пространства. Примечательно, что следующий фрагмент объявляет глобальное имя `com` до проверки его наличия. Сама проверка выполняется точно так же, как и при объявлении пространства имен:

```
var com; // Перед проведением проверки объявляется глобальный символ
if (!com || !com.davidflanagan || !com.davidflanagan.Class)
    throw new Error("com/davidflanagan/Class.js не был загружен");
```

Если автор модуля следует соглашениям по обозначению версий, таким как объявление версии модуля с помощью свойства `VERSION` в пространстве имен, можно проверить не только присутствие модуля, но и узнать его версию. В конце главы приводится пример, где проверка выполняется именно таким образом.

10.1.2. Классы в качестве модулей

Модуль `Class`, использовавшийся в примере 10.1, представляет собой просто набор согласованных вспомогательных функций. Однако нет никаких ограничений на иную организацию модуля. Он может состоять из единственной функции, объявлять JavaScript-класс или даже набор классов и функций.

В примере 10.2 приводится фрагмент программного кода, который создает модуль, состоящий из единственного класса. Данный модуль использует наш гипотетический модуль `Class` и функцию `define()`. (Если вы забыли, для чего предназначена эта функция, обратитесь к примеру 9.10.)

Пример 10.2. Класс комплексных чисел в виде модуля

```
/**
 * com/davidflanagan/Complex.js: класс, реализующий представление комплексных чисел
 *
 * Данный модуль определяет функцию-конструктор com.davidflanagan.Complex()
 * Использует модуль com/davidflanagan/Class.js
 **/
// Прежде всего необходимо проверить присутствие модуля Class
var com; // Объявляется глобальный символ перед проверкой его наличия
if (!com || !com.davidflanagan || !com.davidflanagan.Class)
    throw new Error("com/davidflanagan/Class.js не был загружен");

// В результате проведенной проверки мы выяснили, что пространство имен
// com.davidflanagan существует, поэтому нам не нужно создавать его.
// Достаточно просто объявить класс Complex внутри этого пространства
com.davidflanagan.Complex = com.davidflanagan.Class.define({
    name: "Complex",
    construct: function(x,y) { this.x = x; this.y = y; },
```

```
    methods: {
        add: function(c) {
            return new com.davidflanagan.Complex(this.x + c.x,
                                                 this.y + c.y);
        }
    },
});
```

Существует также возможность определить модуль, состоящий более чем из одного класса. В примере 10.3 приводится пример модуля, который определяет различные классы, представляющие геометрические фигуры.

Пример 10.3. Модуль классов, представляющих геометрические фигуры

```
/**
 * com/davidflanagan/Shapes.js: модуль классов, представляющих геометрические фигуры
 *
 * Данный модуль объявляет классы в пространстве имен com.davidflanagan.shapes
 * Использует модуль com/davidflanagan/Class.js
 **/
// Прежде всего необходимо проверить наличие модуля Class
var com; // Объявляется глобальный символ перед проверкой его наличия
if (!com || !com.davidflanagan || !com.davidflanagan.Class)
    throw new Error("com/davidflanagan/Class.js не был загружен");

// Импортировать символ из этого модуля
var define = com.davidflanagan.Class.define;

// В результате проведенной проверки мы выяснили, что пространство имен
// com.davidflanagan существует, поэтому нам не нужно создавать его.
// Достаточно просто создать пространство имен с фигурами
if (com.davidflanagan.shapes)
    throw new Error("пространство имен com.davidflanagan.shapes существует");

// Создать пространство имен
com.davidflanagan.shapes = {};

// Объявить классы, функции-конструкторы которых
// будут храниться в нашем пространстве имен
com.davidflanagan.shapes.Circle = define   ({ /* данные класса */ });
com.davidflanagan.shapes.Rectangle = define({ /* данные класса */ });
com.davidflanagan.shapes.Triangle = define ({ /* данные класса */});
```

10.1.3. Инициализация модуля

Нередко мы представляем себе модуль как набор функций (или классов). Но как видно из предыдущих примеров, модули – это несколько больше, чем простое объявление функций, которые будут использоваться позже. Они включают в себя программный код, который вызывается при первой загрузке и выполняет операции по инициализации и наполнению пространства имен. Модуль может содержать любой объем такого программного кода однократного запуска, и вполне допустимо создавать модули, не объявляющие никаких функций или классов, а просто запускающие некоторый программный код. Единственное правило, которого следует при этом придерживаться, – модуль не должен загромождать глобальное пространство имен. Лучший способ добиться этого – поместить весь про-

граммный код в одну анонимную функцию, которая должна быть вызвана сразу же после того, как будет определена:

```
(function() { // Определить анонимную функцию. Отсутствие имени
              // означает отсутствие глобального символа
    // Тело функции находится здесь
    // Здесь можно без опаски объявлять любые переменные,
    // поскольку это не приведет к созданию глобальных символов.
})( );        // Конец определения функции и ее вызов.
```

Некоторые модули могут запускать свой программный код сразу же после загрузки. Другие требуют вызова функции инициализации позднее. Для клиентского языка JavaScript существует ставшее обычным требование: модули обычно предназначены для работы с HTML-документом и потому должны инициализироваться после того, как документ полностью загружен веб-броузером.

Модуль может занимать пассивную позицию по отношению к процедуре инициализации, просто определяя и документируя функцию инициализации и предлагая пользователю вызвать эту функцию в нужное время. Это достаточно безопасный и консервативный подход, но он требует, чтобы HTML-документ содержал достаточный объем программного JavaScript-кода, чтобы инициализировать по крайней мере те модули, с которыми он будет взаимодействовать.

Существует парадигма программирования (называемая *ненавязчивым JavaScript-кодом* и описываемая в разделе 13.1.5), в соответствии с которой модули должны быть полностью самодостаточными, а HTML-документы вообще не должны содержать JavaScript-код. Для создания таких «ненавязчивых» модулей необходимо средство, посредством которого модули смогут самостоятельно регистрировать свои функции инициализации, чтобы те автоматически вызывались в подходящие моменты времени.

Пример 10.5 в конце этой главы включает в себя решение, позволяющее модулю самостоятельно зарегистрировать свою функцию инициализации. Внутри броузера все зарегистрированные функции инициализации будут автоматически вызваны в ответ на событие «onload», генерируемое броузером. (Подробнее о событиях и обработчиках событий рассказывается в главе 17.)

10.2. Импорт символов из пространств имен

Проблема придания уникальности названиям пространств имен, таким как `com.davidflanagan.Class`, влечет за собой другую проблему – увеличение длины имен функций, например `com.davidflanagan.Class.define()`. Это полное имя функции, но совсем не обязательно постоянно вводить его вручную всякий раз, когда в этом возникнет необходимость. Поскольку функции в JavaScript являются обычными данными, существует возможность сохранить ссылку на функцию в переменной с любым именем. Например, после загрузки модуля `com.davidflanagan.Class` пользователь модуля может вставить такую строку:

```
// Более простое для ввода имя.
var define = com.davidflanagan.Class.define;
```

Использование пространств имен для предотвращения конфликтов – это обязанность, лежащая на плечах разработчика. Но пользователь модуля обладает прерогативой импортировать символы из пространства имен модуля в глобальное

пространство имен. Программисту, применяющему модуль, уже известно, какие модули он задействует и какие потенциальные конфликты имен возможны. Он в состоянии определить, какие символы и как импортировать, чтобы избежать конфликтов имен.

Обратите внимание: в предыдущем фрагменте используется глобальный символ `define` для представления вспомогательной функции определения классов. Это не очень описательное имя для глобальной функции, поскольку по такому имени трудно сказать, что именно она определяет. Более предпочтительным будет имя:

```
var defineClass = com.davidflanagan.Class.define;
```

Но такое изменение имен методов – тоже не лучший выход. Другой программист, который пользовался этим же модулем ранее, может прийти в недоумение, встретив имя `defineClass()`, потому что он знаком с другой функцией – `define()`. Нередко разработчики модулей вкладывают определенный смысл в имена своих функций, и изменять эти имена несправедливо по отношению к модулям. Другой способ заключается в отказе от использования глобального пространства имен и импорте символов в пространства имен с более короткими именами:

```
// Создать простое пространство имен. При этом нет необходимости выполнять проверку
// на наличие ошибок, т. к. пользователь знает, какие символы существуют, а какие нет.
var Class = {};
// Импортировать символ в новое пространство имен.
Class.define = com.davidflanagan.Class.define;
```

Существует несколько моментов, которые связаны с импортом символов и которые необходимо понимать. Первый момент: *допускается импортировать только те символы, которые являются ссылками на функции, объекты или массивы*. Если импортируется символ, представляющий значение элементарного типа, такого как число или строка, тем самым создается статическая копия этого значения. Любые изменения такого значения, выполняемые в пределах пространства имен, никак не сказываются на импортированной копии. Предположим, что метод `Class.define()` обслуживает счетчик классов, которые определяются с его помощью, и наращивает значение `com.davidflanagan.Class.counter` при каждом вызове. Если попытаться импортировать это значение, будет просто создана статическая копия текущего значения счетчика:

```
// Всего лишь создает статическую копию. Изменения в пространстве имен не будут отражаться
// на импортированном свойстве, поскольку это значение принадлежит элементарному типу.
Class.counter = com.davidflanagan.Class.counter;
```

Это урок для разработчиков модулей – если предполагается объявлять внутри модуля свойства со значениями элементарных типов, необходимо реализовать методы доступа к ним, чтобы эти методы можно было импортировать:

```
// Свойство элементарного типа, оно не должно импортироваться
com.davidflanagan.Class.counter = 0;

// Это метод доступа, который можно импортировать
com.davidflanagan.Class.getCounter = function() {
    return com.davidflanagan.Class.counter;
}
```

Второй важный момент, который необходимо понимать, – модули создаются для пользователей модулей. *Разработчики модулей всегда должны указывать*

полные имена своих символов. Следование этому правилу можно наблюдать в только что продемонстрированном методе `getCounter()`. Поскольку JavaScript не обладает встроенной поддержкой модулей и пространств имен, сокращения здесь неуместны и необходимо указать полное имя свойства `counter`, даже при том, что метод доступа `getCounter()` принадлежит тому же пространству имен. Разработчики модулей не должны полагаться на то, что их функции будут импортироваться в глобальное пространство имен. Функции, которые вызывают другие функции модуля, для корректной работы должны использовать полные имена, даже если не предполагается возможность импорта функции. (Исключением из этого правила являются замыкания, о чем рассказывается в разделе 10.2.2.)

10.2.1. Общедоступные и частные символы

Не все символы, объявляемые в модуле, предназначены для применения за его пределами. Модули могут иметь свои внутренние функции и переменные, не предназначенные для непосредственного использования в сценарии, который работает с данным модулем. В JavaScript отсутствует возможность определять, какие символы пространства имен будут общедоступными, а какие нет. Здесь опять же приходится довольствоваться соглашениями, предотвращающими некорректное использование частных символов за пределами модуля.

Наиболее прямолинейный способ – подробное документирование таких символов. Разработчик модуля должен четко указать в документации, какие функции и прочие свойства представляют общедоступный прикладной интерфейс модуля. Пользователь модуля, в свою очередь, должен ограничиться общедоступным интерфейсом и не поддаться искушению вызвать какую-либо другую функцию или обратиться к какому-нибудь другому свойству.

Одно из соглашений, которое поможет отличать общедоступные символы от частных даже без обращения к документации, заключается в использовании символа подчеркивания в качестве префикса имен частных символов. Что касается обсуждаемой функции доступа `getCounter()`, можно четко обозначить, что свойство `counter` является частным, изменив его имя на `_counter`. Это не исключает возможность использования свойства за пределами модуля, но помешает программисту допустить обращение к частному свойству по неосторожности.

Модули, распространяемые через JSAN, пошли еще дальше. Определения модулей включают в себя массивы, где перечислены все общедоступные символы. Модуль из архива JSAN с именем *JSAN* включает в себя вспомогательные функции, с помощью которых можно импортировать символы модуля, и эти функции отвергают попытки импорта символов, отсутствующих в этих массивах.

10.2.2. Замыкания как частные пространства имен и область видимости

В разделе 8.8 говорилось, что замыкание – это функция вместе с областью видимости, которая действовала на момент определения функции.[1] Определяя функ-

[1] Замыкания – это тема повышенной сложности. Если вы пропустили обсуждение замыканий в главе 8, вам необходимо сначала прочитать о замыканиях, а затем вернуться к этому разделу.

цию таким способом, появляется возможность использовать локальную область видимости как пространство имен. Вложенные функции, объявляемые внутри объемлющих функций, имеют возможность доступа к таким частным пространствам имен. Подобный подход имеет два преимущества. Первое основано на том, что поскольку частное пространство имен является первым объектом в цепочке областей видимости, функции в частном пространстве имен могут ссылаться на другие функции и свойства в этом же пространстве имен без необходимости указывать полные имена.

Второе преимущество заключается в том, что эти пространства имен действительно являются частными. Нет никакой возможности обратиться к символам, объявленным внутри функции, за ее пределами. Эти символы будут доступны во внешнем, общедоступном пространстве имен, только если функция экспортирует их. Это означает, что модуль может экспортировать только общедоступные функции и скрыть подробности реализации, такие как служебные методы и переменные внутри замыканий.

Эту возможность иллюстрирует пример 10.4. Здесь с помощью замыкания создается частное пространство имен, после чего общедоступные методы экспортируются в общедоступное пространство имен.

Пример 10.4. Определение частного пространства имен с помощью замыкания

```
// Создать объект пространства имен.
// Для краткости проверка ошибок отсутствует.
var com;
if (!com) com = {};
if (!com.davidflanagan) com.davidflanagan = {};
com.davidflanagan.Class = {};

// Здесь ничего не создается непосредственно в пространстве имен.
// Вместо этого объявляется и вызывается анонимная функция, которая
// создает замыкание, используемое как частное пространство имен.
// Данная функция экспортирует общедоступные символы из замыкания
// в объект com.davidflanagan.Class
// Обратите внимание: функция не имеет имени, поэтому не создается
// никаких глобальных символов.
(function( ) { // Начало определения анонимной функции
    // Вложенные функции создают символы внутри замыкания
    function define(data) { counter++; /* тело функции */ }
    function provides(o, c) { /* тело функции */ }

    // Локальные переменные - это символы, расположенные внутри замыкания.
    // Этот символ останется частным и будет доступен только внутри замыкания
    var counter = 0;

    // Эта функция может обращаться к переменной с помощью простого имени
    // и не пользоваться полным именем, определяющим пространство имен
    function getCounter( ) { return counter; }

    // Теперь, когда внутри замыкания были определены свойства,
    // которые должны оставаться частными, можно экспортировать символы,
    // доступные во внешнем пространстве имен
    var ns = com.davidflanagan.Class;
    ns.define = define;
    ns.provides = provides;
```

```
    ns.getCounter = getCounter;
})( ); // Конец определения анонимной функции и ее вызов
```

10.3. Модуль со вспомогательными функциями

В этом разделе представлен расширенный пример модуля, содержащего функции для работы с модулями. Функция `Module.createNamespace()` создает пространство имен и выполняет проверку на наличие ошибок. Автор модуля может использовать эту функцию следующим образом:

```
// Создать пространство имен модуля
Module.createNamespace("com.davidflanagan.Class");

// Заполнить это пространство
com.davidflanagan.Class.define = function(data) { /* тело функции */ };
com.davidflanagan.Class.provides = function(o, c) { /* тело функции */ };
```

Функция `Module.require()` проверяет присутствие заданной (или более поздней) версии модуля и возбуждает исключение, если он отсутствует. Используется она следующим образом:

```
// Модуль Complex требует, чтобы предварительно был загружен модуль Class
Module.require("com.davidflanagan.Class", 1.0);
```

Функция `Module.importSymbols()` упрощает задачу импорта символов в глобальное пространство имен или любое другое заданное пространство имен. Вот пример ее использования:

```
// Импортировать символы по умолчанию модуля Module в глобальное пространство имен
// Одним из таких символов по умолчанию является сама функция importSymbols
Module.importSymbols(Module); // Отметьте, что мы передаем пространство
                              // имен, а не имя модуля

// Импортировать класс Complex в глобальное пространство имен
importSymbols(com.davidflanagan.Complex);

// Импортировать метод com.davidflanagan.Class.define() в объект Class
var Class = {};
importSymbols(com.davidflanagan.Class, Class, "define");
```

Наконец, функция `Module.registerInitializationFunction()` позволяет модулю зарегистрировать функцию инициализации, которая будет запущена позднее.[1] Когда эта функция используется в клиентском языке JavaScript, автоматически производится регистрация обработчика события, который по окончании загрузки документа вызовет все функции инициализации всех загруженных модулей. В других (не клиентских) контекстах функции инициализации автоматически не вызываются, но есть возможность явно сделать это с помощью функции `Module.runInitializationFunctions()`.

Исходные тексты модуля `Module` приводятся в примере 10.5. Этот пример достаточно длинный, но его детальное изучение окупит себя с лихвой. Подробное описание каждой функции есть в тексте примера.

---

[1] Похожая функция регистрации функций инициализации приводится в примере 17.6.

Пример 10.5. Модуль с функциями для обслуживания модулей

```
/**
 * Module.js: Функции для работы с модулями и пространствами имен
 *
 * Этот модуль содержит функции для работы с модулями, которые
 * совместимы с модулями из архива JSAN.
 * Данный модуль определяет пространство имен Module.
 */

// Убедиться, что данный модуль еще не загружен
var Module;
if (Module && (typeof Module != "object" || Module.NAME))
    throw new Error("Пространство имен 'Module' уже существует");

// Создать собственное пространство имен
Module = {};

// Далее располагается метаинформация об этом пространстве имен
Module.NAME = "Module";     // Название этого пространства имен
Module.VERSION = 0.1;       // Версия этого пространства имен

// Далее следует список общедоступных символов, которые будут
// экспортироваться этим пространством имен.
// Эта информация интересна тем, кто будет пользоваться модулем
Module.EXPORT = ["require", "importSymbols"];

// Далее следует перечень символов, которые также будут экспортироваться.
// Но они, как правило, используются только авторами модулей
// и обычно не импортируются.
Module.EXPORT_OK = ["createNamespace", "isDefined",
                    "registerInitializationFunction",
                    "runInitializationFunctions",
                    "modules", "globalNamespace"];

// Начинается добавление символов в пространство имен
Module.globalNamespace = this;        // Так мы всегда ссылаемся
                                      // на глобальную область видимости
Module.modules = { "Module": Module };// Соответствие Module [name]->namespace.

/**
 * Данная функция создает и возвращает объект пространства имен с заданным
 * именем и выполняет проверку на наличие конфликта между этим именем
 * и именами из любых ранее загруженных модулей.
 * Если какой-либо компонент пространства имен уже существует
 * и не является объектом, генерируется исключение.
 *
 * В качестве значения свойства NAME устанавливается имя этого пространства имен.
 * Если был задан аргумент version, устанавливает свойство пространство имен VERSION.
 *
 * Отображение нового пространства имен добавляется в объект Module.modules
 */
Module.createNamespace = function(name, version) {
    // Проверить корректность имени. Оно должно существовать и не должно
    // начинаться или заканчиваться символом точки или содержать в строке
    // два символа точки подряд.
```

```
    if (!name) throw new Error("Module.createNamespace(): не указано имя");
    if (name.charAt(0) == '.' ||
        name.charAt(name.length-1) == '.' ||
        name.indexOf("..") != -1)
        throw new Error("Module.createNamespace(): неверное имя: " + name);

    // Разбить имя по символам точки и создать иерархию объектов
    var parts = name.split('.');

    // Для каждого компонента пространства имен следует создать объект
    // либо убедиться, что объект с таким именем уже существует.
    var container = Module.globalNamespace;
    for(var i = 0; i < parts.length; i++) {
        var part = parts[i];
        // Если свойства или контейнера с таким именем не существует,
        // создать пустой объект.
        if (!container[part]) container[part] = {};
        else if (typeof container[part] != "object") {
            // Если свойство уже существует, убедиться, что это объект
            var n = parts.slice(0,i).join('.');
            throw new Error(n + " уже существует, но не является объектом");
        }
        container = container[part];
    }

    // Последний контейнер, который был рассмотрен последним, - это то, что нам нужно.
    var namespace = container;

    // Было бы ошибкой определить одно и то же пространство имен дважды,
    // но нет никакого криминала, если объект уже существует и у него
    // не определено свойство NAME.
    if (namespace.NAME) throw new Error("Модуль "+name+" уже определен ");

    // Инициализировать поля с именем и версией пространства имен
    namespace.NAME = name;
    if (version) namespace.VERSION = version;

    // Зарегистрировать это пространство имен в списке модулей
    Module.modules[name] = namespace;

    // Вернуть объект пространства имен вызывающей программе
    return namespace;
}

/**
 * Проверить, был ли определен модуль с заданным именем.
 * Вернуть true, если определен, и false - в противном случае.
 */
Module.isDefined = function(name) {
    return name in Module.modules;
};

/**
 * Эта функция возбуждает исключение, если модуль с таким именем не определен
 * или его версия меньше указанной. Если пространство имен существует
 * и имеет допустимый номер версии, эта функция просто возвращает управление,
 * не предпринимая никаких действий. Эта функция может стать источником
 * фатальной ошибки, если модуль, требуемый вашему программному коду, отсутствует.
```

```
 */
Module.require = function(name, version) {
    if (!(name in Module.modules)) {
        throw new Error("Модуль " + name + " не определен");
}

    // Если версия не указана, проверка не выполняется
    if (!version) return;

    var n = Module.modules[name];

    // Если номер версии модуля ниже, чем требуется, или пространство
    // имен не объявляет версию, генерируется исключение.
    if (!n.VERSION || n.VERSION < version)
        throw new Error("Модуль " + name + " имеет версию " +
                        n.VERSION + " требуется версия " + version +
                        " или выше.");
};

/**
 * Данная функция импортирует символы из заданного модуля. По умолчанию
 * импорт производится в глобальное пространство имен, однако с помощью
 * второго аргумента можно определить иное место назначения.
 *
 * Если никакие символы явно не указываются, будут импортированы символы
 * из массива EXPORT модуля. Если этот массив, а также массив EXPORT_OK
 * не определен, будут импортированы все символы модуля from.
 *
 * Чтобы импортировать явно указанный набор символов, их имена должны
 * передаваться в виде аргументов вслед за именем модуля и именем
 * пространства имен, куда производится импорт. Если модуль содержит
 * определение массива EXPORT или EXPORT_OK, импортированы будут только
 * те символы, которые перечислены в одном из этих массивов.
 */
Module.importSymbols = function(from) {
    // Убедиться, что модуль корректно задан. Функция ожидает получить объект
    // пространства имен модуля, но также допускается строка с именем модуля
    if (typeof from == "string") from = Module.modules[from];
    if (!from || typeof from != "object")
        throw new Error("Module.importSymbols(): " +
                        "требуется указать объект пространства имен");

    // Вслед за аргументом с источником импортируемых символов может
    // следовать объект пространства имен, куда производится импорт,
    // а также имена импортируемых символов.
    var to = Module.globalNamespace; // Место назначения по умолчанию
    var symbols = [];                // По умолчанию нет символов
    var firstsymbol = 1;             // Индекс первого аргумента с именем символа

    // Проверить, задано ли пространство имен места назначения
    if (arguments.length > 1 && typeof arguments[1] == "object") {
        if (arguments[1] != null) to = arguments[1];
        firstsymbol = 2;
    }

    // Получить список явно указанных символов
    for(var a = firstsymbol; a < arguments.length; a++)
```

```
        symbols.push(arguments[a]);

    // Если ни одного символа не было передано, импортировать набор символов,
    // определяемых модулем умолчанию, или просто импортировать все символы.
    if (symbols.length == 0) {
        // Если в модуле определен массив EXPORT, импортировать символы из этого массива.
        if (from.EXPORT) {
            for(var i = 0; i < from.EXPORT.length; i++) {
                var s = from.EXPORT[i];
                to[s] = from[s];
            }
            return;
        }
        // Иначе, если массив EXPORT_OK в модуле не определен,
        // импортировать все символы из пространства имен модуля
        else if (!from.EXPORT_OK) {
            for(s in from) to[s] = from[s];
            return;
        }
    }

    // В этой точке имеется массив импортируемых символов, определенных явно.
    // Если пространство имен определяет массивы EXPORT и/или EXPORT_OK,
    // прежде чем импортировать символы, убедиться, что каждый из них
    // присутствует в этих массивах. Генерировать исключение, если запрошенный
    // символ не определен или не предназначен для экспорта.
    var allowed;
    if (from.EXPORT || from.EXPORT_OK) {
        allowed = {};
        // Скопировать допустимые символы из массивов в свойства объекта.
        // Это даст возможность более эффективно проверить допустимость импорта символа.
        if (from.EXPORT)
            for(var i = 0; i < from.EXPORT.length; i++)
                allowed[from.EXPORT[i]] = true;
        if (from.EXPORT_OK)
            for(var i = 0; i < from.EXPORT_OK.length; i++)
                allowed[from.EXPORT_OK[i]] = true;
    }

    // Импортировать символы
    for(var i = 0; i < symbols.length; i++) {
        var s = symbols[i];             // Имя импортируемого символа
        if (!(s in from))               // Проверить его наличие
            throw new Error("Module.importSymbols(): символ " + s + " не определен ");
        if (allowed && !(s in allowed)) // Убедиться, что это общедоступный символ
            throw new Error("Module.importSymbols(): символ " + s +
                            " не является общедоступным " +
                            "и не может быть импортирован.");
        to[s] = from[s];                // Импортировать символ
    }
};

// Эта функция используется модулями для регистрации одной
// или более функций инициализации.
Module.registerInitializationFunction = function(f) {
```

```
    // Сохранить функцию в массиве функций инициализации
    Module._initfuncs.push(f);
    // Если обработчик события onload еще не зарегистрирован, сделать это сейчас.
    Module._registerEventHandler();
}

// Данная функция вызывает зарегистрированные функции инициализации.
// В клиентском JavaScript она автоматически вызывается по окончании загрузки документа.
// В других контекстах исполнения может потребоваться вызвать эту функцию явно.
Module.runInitializationFunctions = function() {
    // Запустить каждую из функций, перехватывая и игнорируя исключения,
    // чтобы ошибка в одном модуле не помешала инициализироваться другим модулям.
    for(var i = 0; i < Module._initfuncs.length; i++) {
        try { Module._initfuncs[i](); }
        catch(e) { /* игнорировать исключения */}
    }
    // Уничтожить массив, т. к. такие функции вызываются всего один раз.
    Module._initfuncs.length = 0;
}

// Частный массив, где хранятся функции инициализации для последующего вызова
Module._initfuncs = [];

// Если модуль был загружен веб-броузером, эта частная функция регистрируется
// как обработчик события onload, чтобы иметь возможность запустить все функции
// инициализации по окончании загрузки всех модулей.
// Она не допускает обращение к себе более одного раза.
Module._registerEventHandler = function() {
    var clientside =      // Проверить хорошо известные клиентские свойства
        "window" in Module.globalNamespace &&
        "navigator" in window;

    if (clientside) {
        if (window.addEventListener) { // Регистрация по стандарту W3C DOM
            window.addEventListener("load", Module.runInitializationFunctions,
                                    false);
        }
        else if (window.attachEvent) { // Регистрация в IE5+
            window.attachEvent("onload", Module.runInitializationFunctions);
        }
        else {
            // IE4 и более старые броузеры, если тег <body> определяет атрибут onload,
            // этот обработчиксобытия будет перекрыт и никогда не будет вызван.
            window.onload = Module.runInitializationFunctions;
        }
    }

    // Функция перекрывает сама себя пустой функцией,
    // чтобы предотвратить возможность повторного вызова.
    Module._registerEventHandler = function() {};
}
```

11

Шаблоны и регулярные выражения

Регулярное выражение – это объект, описывающий символьный шаблон. Класс `RegExp` в JavaScript представляет регулярные выражения, а объекты классов `String` и `RegExp` предоставляют методы, использующие регулярные выражения для выполнения поиска по шаблону и операций поиска в тексте с заменой.[1]

Регулярные JavaScript-выражения стандартизованы в ECMAScript v3. JavaScript 1.2 реализует только подмножество регулярных выражений, требуемых стандартом ECMAScript v3, а полностью стандарт реализован в JavaScript 1.5. Регулярные выражения в JavaScript в значительной степени базируются на средствах регулярных выражений из языка программирования Perl. Грубо говоря, мы можем сказать, что JavaScript 1.2 реализует регулярные выражения Perl 4, а JavaScript 1.5 – большое подмножество регулярных выражений Perl 5.

Эта глава начинается с определения синтаксиса, посредством которого в регулярных выражениях описываются текстовые шаблоны. Затем мы перейдем к описанию тех методов классов `String` и `RegExp`, которые используют регулярные выражения.

11.1. Определение регулярных выражений

В JavaScript регулярные выражения представлены объектами `RegExp`. Объекты `RegExp` могут быть созданы посредством конструктора `RegExp()`, но чаще они создаются с помощью специального синтаксиса литералов. Так же, как строковые литералы задаются в виде символов, заключенных в кавычки, литералы регулярных выражений задаются в виде символов, заключенных в пару символов слэша (/). Таким образом, JavaScript-код может содержать строки, похожие на эту:

---

[1] Происхождение малопонятного термина «регулярное выражение» уходит в далекое прошлое. Синтаксис, применяемый для описания текстового шаблона, действительно представляет собою особый тип выражения, однако, как мы увидим, этот синтаксис очень далек от регулярного! Регулярные выражения иногда называют «regexp», или просто «RE».

```
var pattern = /s$/;
```

Эта строка создает новый объект RegExp и присваивает его переменной pattern. Данный объект RegExp ищет любые строки, заканчивающиеся символом s. (Скоро мы поговорим о грамматике определения шаблонов.) Это же регулярное выражение может быть определено с помощью конструктора RegExp():

```
var pattern = new RegExp("s$");
```

Создание объекта RegExp – либо с помощью литерала, либо с помощью конструктора RegExp() – это самая простая часть работы. Более сложную задачу представляет собой описание нужного шаблона с помощью синтаксиса регулярных выражений. JavaScript поддерживает довольно полное подмножество синтаксиса регулярных выражений, используемых в Perl, поэтому если вы опытный Perl-программист, то уже знаете, как описывать шаблоны в JavaScript.

Спецификация шаблона регулярного выражения состоит из последовательности символов. Большинство символов, включая все алфавитно-цифровые, просто буквально описывают символы, которые должны присутствовать. То есть регулярное выражение /java/ ищет все строки, содержащие подстроку "java". Другие символы в регулярных выражениях не предназначены для поиска их точных эквивалентов, а имеют особое значение. Например, регулярное выражение /s$/ содержит два символа. Первый символ, s, обозначает поиск буквального символа. Второй, $, – это специальный метасимвол, обозначающий конец строки. Таким образом, это регулярное выражение соответствует любой строке, заканчивающейся символом s.

В следующих разделах описаны различные символы и метасимволы, используемые в регулярных JavaScript-выражениях. Следует заметить, что полное описание регулярных выражений выходит за рамки темы этой книги, его можно найти в книгах по Perl, таких как книга издательства O'Reilly «Programming Perl» Ларри Уолла (Larry Wall), Тома Кристиансена (Tom Christiansen) и Джона Орванта (Jon Orwant).[1] Еще один отличный источник информации по регулярным выражениям – книга издательства O'Reilly «Mastering Regular Expressions» Джеффри Фридла (Jeffrey E. F. Friedl).[2]

11.1.1. Символы литералов

Как отмечалось ранее, все алфавитные символы и цифры в регулярных выражениях соответствуют сами себе. Синтаксис регулярных выражений в JavaScript также поддерживает возможность указывать некоторые неалфавитные символы с помощью управляющих последовательностей, начинающихся с символа обратного слэша (\).[3] Например, последовательность \n соответствует символу перевода строки. Эти символы перечислены в табл. 11.1.

---

[1] Ларри Уолл, Том Кристиансен, Джон Орвант «Программирование на Perl», 3-е издание. – Пер. с англ. – СПб: Символ-Плюс, 2002.

[2] Джеффри Фридл «Регулярные выражения», 3-е издание. – Пер. с англ. – СПб: Символ-Плюс, 2008.

[3] Символ экранирования непосредственно за ним следующего символа. – *Примеч. науч. ред.*

Таблица 11.1. Символы литералов в регулярных выражениях

Символ	Соответствие
Алфавитно-цифровые символы	Соответствуют самим себе
\0	Символ NUL (\u0000)
\t	Табуляция (\u0009)
\n	Перевод строки (\u000A)
\v	Вертикальная табуляция (\u000B)
\f	Перевод страницы (\u000C)
\r	Возврат каретки (\u000D)
\x*nn*	Символ из набора Latin, задаваемый шестнадцатеричным числом *nn*; например, \x0A – это то же самое, что \n
\u*xxxx*	Unicode-символ, заданный шестнадцатеричным числом *xxxx*; например, \u0009 – это то же самое, что \t
\c*X*	Управляющий символ ^*X*; например, \cJ эквивалентно символу перевода строки \n

Некоторые знаки препинания имеют в регулярных выражениях особый смысл:

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

Значение этих символов раскрывается в последующих разделах. Некоторые из них имеют специальный смысл только в определенном контексте регулярного выражения, а в других контекстах трактуются буквально. Однако, как правило, чтобы включить какой-либо из этих символов в регулярное выражение буквально, необходимо поместить перед ним символ обратного слэша. Другие символы[1], такие как кавычки и @, не имеют специального значения и просто соответствуют в регулярных выражениях самим себе.

Если вы не можете точно вспомнить, каким из символов должен предшествовать символ \, можете спокойно помещать обратный слэш перед любым из символов. Однако имейте в виду, что многие буквы и цифры вместе с символом слэша обретают специальное значение, поэтому тем буквам и цифрам, которые вы ищете буквально, не должен предшествовать символ \. Чтобы включить в регулярное выражение сам символ обратного слэша, перед ним, очевидно, следует поместить другой символ обратного слэша. Например, следующее регулярное выражение соответствует любой строке, содержащей символ обратного слэша: /\\/.

11.1.2. Классы символов

Отдельные символы литералов могут объединяться в классы символов путем помещения их в квадратные скобки. Класс символов соответствует любому символу, содержащемуся в этом классе. Следовательно, регулярное выражение /[abc]/ соответствует одному из символов a, b или c. Могут также определяться классы

[1] Из числа знаков препинания. – *Примеч. науч. ред.*

символов с отрицанием, соответствующие любому символу, кроме тех, которые указаны в скобках. Класс символов с отрицанием задается символом ^ в качестве первого символа, следующего за левой скобкой. Регулярное выражение /[^abc]/ соответствует любому символу, отличному от a, b или c. В классах символов диапазон символов может задаваться при помощи дефиса. Поиск всех символов латинского алфавита в нижнем регистре осуществляется посредством выражения /[a-z]/, а любую букву или цифру из набора символов Latin можно найти при помощи выражения /[a-zA-Z0-9]/.

Некоторые классы символов используются особенно часто, поэтому синтаксис регулярных выражений в JavaScript включает специальные символы и управляющие (escape) последовательности для их обозначения. Так, \s соответствует символам пробела, табуляции и любым пробельным (whitespaces) символам-разделителям из набора Unicode, а \S – любым символам, не являющимся символами-разделителями из набора Unicode. В табл. 11.2 приводится перечень этих спецсимволов и синтаксиса классов символов. (Обратите внимание: некоторые из управляющих последовательностей классов символов соответствуют только ASCII-символам и не расширены для работы с Unicode-символами. Можно явно определить собственные классы Unicode-символов, например, выражение /[\u0400-\04FF]/ соответствует любому символу кириллицы.)

Таблица 11.2. Классы символов регулярных выражений

Символ	Соответствие
[...]	Любой из символов, указанных в скобках
[^...]	Любой из символов, не указанных в скобках
.	Любой символ, кроме перевода строки или другого разделителя Unicode-строки
\w	Любой текстовый ASCII-символ. Эквивалентно [a-zA-Z0-9_]
\W	Любой символ, не являющийся текстовым ASCII-символом. Эквивалентно [^a-zA-Z0-9_]
\s	Любой символ-разделитель из набора Unicode
\S	Любой символ, не являющийся символом-разделителем из набора Unicode. Обратите внимание: \w и \S – это не одно и то же
\d	Любые ASCII-цифры. Эквивалентно [0-9]
\D	Любой символ, отличный от ASCII-цифр. Эквивалентно [^0-9]
[\b]	Литерал символа «забой» (особый случай)

Обратите внимание: управляющие последовательности специальных классов символов могут находиться в квадратных скобках. \s соответствует любому символу-разделителю, а \d соответствует любой цифре, следовательно, /[\s\d]/ соответствует любому символу-разделителю или цифре. Обратите внимание на особый случай. Как мы увидим позже, последовательность \b имеет особый смысл. Однако когда она используется в классе символов, то обозначает символ «забой». Поэтому, для того чтобы обозначить символ «забой» в регулярном выражении буквально, используйте класс символов с одним элементом: /[\b]/.

11.1.3. Повторение

Изучив синтаксис регулярных выражений, мы можем описать числа из двух цифр `/\d\d/` или из четырех цифр `/\d\d\d\d/`, но не сможем, например, описать число, состоящее из любого количества цифр, или строку из трех букв, за которыми следует необязательная цифра. Эти более сложные шаблоны используют синтаксис регулярных выражений, указывающий, сколько раз может повторяться данный элемент регулярного выражения.

Символы, обозначающие повторение, всегда следуют за шаблоном, к которому они применяются. Некоторые виды повторений используются довольно часто, и для обозначения этих случаев имеются специальные символы. Например, `+` соответствует одному или нескольким экземплярам предыдущего шаблона. В табл. 11.3 приведена сводка синтаксиса повторений.

Таблица 11.3. Символы повторения в регулярных выражениях

Символ	Значение
`{n,m}`	Соответствует предшествующему шаблону, повторенному не менее *n*, но не более *m* раз
`{n,}`	Соответствует предшествующему шаблону, повторенному *n* или более раз
`{n}`	Соответствует в точности *n* экземплярам предшествующего шаблона
`?`	Соответствует нулю или одному экземпляру предшествующего шаблона; предшествующий шаблон является необязательным. Эквивалентно `{0,1}`
`+`	Соответствует одному или более экземпляру предшествующего шаблона. Эквивалентно `{1,}`
`*`	Соответствует нулю или более экземплярам предшествующего шаблона.[a] Эквивалентно `{0,}`

[a] Как и для символа `?`, предшествующий символу `*` шаблон может отсутствовать, этот случай толкуется как: «предшествующий символ шаблона – любой из символов». Именно это делает символы повторения `?` и `*` одними из наиболее используемых.

В следующих строках показано несколько примеров:

```
/\d{2,4}/        // Соответствует числу, содержащему от двух до четырех цифр
/\w{3}\d?/       // Соответствует в точности трем текстовым символам
                 // и необязательной цифре
/\s+java\s+/     // Соответствует слову "java" с одним или несколькими
                 // пробелами до и после него
/[^"]*/          // Соответствует нулю или более символам, отличным от кавычек
```

Будьте осторожны при использовании символов повторения `*` и `?`. Они могут соответствовать отсутствию указанного перед ними шаблона и, следовательно, отсутствию символов. Например, регулярному выражению `/a*/` соответствует строка `"bbbb"`, поскольку в ней нет символа `a`!

11.1.3.1. «Нежадное» повторение

Символы повторения, перечисленные в табл. 11.3, соответствуют максимально возможному количеству повторений, при котором обеспечивается поиск последующих частей регулярного выражения. Мы говорим, что это – «жадное» повторение. Помимо него в JavaScript 1.5 и более поздних версиях (это одна из возможностей Perl 5, не реализованная в JavaScript 1.2) поддерживается повторение, выполняемое «нежадным» способом. Достаточно указать после символа (или символов) повторения вопросительный знак: `??`, `+?`, `*?` или даже `{1,5}?`. Например, регулярное выражение `/a+/` соответствует одному или более экземплярам буквы `a`. Примененное к строке `"aaa"`, оно соответствует всем трем буквам. При этом выражение `/a+?/` соответствует одному или более экземплярам буквы `a` и выбирает наименее возможное число символов. Примененный к той же строке, этот шаблон соответствует только первой букве `a`.

«Нежадное» повторение не всегда дает ожидаемый результат. Рассмотрим шаблон `/a*b/`, соответствующий нулю или более символов `a`, за которыми следует символ `b`. Применительно к строке `"aaab"`, ему соответствует вся строка. Теперь проверим «нежадную» версию `/a*?b/`. Она должна соответствовать символу `b`, перед которым следует наименьшее возможное количество букв `a`. В случае применения к той же строке `"aaab"` можно ожидать соответствия лишь последнего символа `b`. Однако на самом деле этому шаблону соответствует и вся строка, так же как и в случае «жадной» версии. Дело в том, что поиск по шаблону регулярного выражения выполняется путем нахождения первой позиции в строке, начиная с которой соответствие становится возможным. «Нежадная» версия шаблона находит соответствие с первым символом строки, и именно это соответствие является окончательным, а соответствие последующих символов даже не рассматривается.

11.1.4. Альтернативы, группировка и ссылки

Грамматика регулярных выражений включает специальные символы определения альтернатив, подвыражений группировки и ссылок на предыдущие подвыражения. Символ вертикальной черты `|` служит для разделения альтернатив. Например, `/ab|cd|ef/` соответствует либо строке «ab», либо строке «cd», либо строке «ef», а шаблон `/\d{3}|[a-z]{4}/` – либо трем цифрам, либо четырем строчным буквам.

Обратите внимание: альтернативы обрабатываются слева направо до тех пор, пока не будет найдено соответствие. Если левая альтернатива найдена, правая игнорируется, даже если может добиться «лучшего» соответствия. Поэтому когда к строке «ab» применяется шаблон `/a|ab/`, он будет соответствовать только первому символу.

Круглые скобки имеют в регулярных выражениях несколько значений. Одно из них – группировка отдельных элементов в одно подвыражение, так что элементы при использовании спецсимволов `|`, `*`, `+`, `?` и прочих спецсимволов рассматриваются как одно целое. Например, `/java(script)?/` соответствует слову «java», за которым следует необязательное слово «script», а `/(ab|cd)+|ef)/` соответствует либо строке «ef», либо одному или более повторений одной из строк «ab» или «cd».

Другим применением скобок в регулярных выражениях является определение подшаблонов внутри шаблона. Когда в целевой строке найдено соответствие ре-

гулярному выражению, можно извлечь часть целевой строки, соответствующую любому конкретному подшаблону, заключенному в скобки. (Мы увидим, как получить эти подстроки, позднее в этой главе.) Предположим, что требуется отыскать одну или более букв в нижнем регистре, за которыми следует одна или несколько цифр. Для этого можно воспользоваться шаблоном `/[a-z]+\d+/`. Но предположим также, что нам нужны только цифры в конце каждого соответствия. Если мы поместим эту часть шаблона в круглые скобки (`/[a-z]+(\d+)/`), то сможем извлечь цифры из любых найденных нами соответствий. Позднее я объясню, как это делается.

С этим связано еще одно применение подвыражений в скобках, позволяющее делать ссылку назад к подвыражению из предыдущей части того же регулярного выражения. Это достигается путем указания одной или нескольких цифр после символа `\`. Цифры ссылаются на позицию подвыражения в скобках внутри регулярного выражения. Например, `\1` ссылается на первое подвыражение, а `\3` — на третье. Обратите внимание: поскольку подвыражения могут быть вложены одно в другое, при подсчете используется позиция левой скобки. Например, в следующем регулярном выражении ссылка на вложенное подвыражение (`[Ss]cript`) будет выглядеть как `\2`:

```
/([Jj]ava([Ss]cript)?)\sis\s(fun\w*)/
```

Ссылка на подвыражение регулярного выражения указывает не на шаблон этого подвыражения, а на найденный текст, соответствующий этому шаблону. Поэтому ссылки могут использоваться для наложения ограничения, выбирающего части строки, содержащие точно такие же символы. Например, следующее регулярное выражение соответствует нулю или более символам внутри одинарных или двойных кавычек. Однако оно не требует, чтобы открывающие и закрывающие кавычки соответствовали друг другу (т. е. чтобы обе кавычки были одинарными или двойными):

```
/['"][^'"]*['"]/
```

Соответствия кавычек мы можем потребовать посредством такой ссылки:

```
/(['"])[^'"]*\1/
```

Здесь `\1` соответствует результату поиска в соответствии с первым подвыражением. В этом примере ссылка налагает ограничение, требующее, чтобы закрывающая кавычка соответствовала открывающей. Это регулярное выражение не допускает присутствия одинарных кавычек внутри двойных, и наоборот. Недопустимо помещать ссылки внутрь классов символов, т. е. мы не можем написать:

```
/(['"])[^\1]*\1/
```

Позднее в этой главе мы увидим, что этот вид ссылок на подвыражения представляет собой мощное средство для использования регулярных выражений в операциях поиска/замены.

В JavaScript 1.5 (но не в JavaScript 1.2) возможна группировка элементов в регулярном выражении без создания нумерованной ссылки на эти элементы. Вместо простой группировки элементов между (и) начните группу с символов (`?:` и закончите ее символом). Рассмотрим, например, следующий шаблон:

```
/([Jj]ava(?:[Ss]cript)?)\sis\s(fun\w*)/
```

Здесь подвыражение (?:[Ss]cript) нужно только для группировки, чтобы к группе мог быть применен символ повторения ?. Эти модифицированные скобки не создают ссылку, поэтому в данном регулярном выражении \2 ссылается на текст, соответствующий шаблону (fun\w*).

В табл. 11.4 приводится перечень операторов альтернативы, группировки и ссылки в регулярных выражениях.

Таблица 11.4. Символы альтернативы, группировки и ссылки в регулярных выражениях

Символ	Значение
|	Альтернативы. Соответствует либо подвыражению слева, либо подвыражению справа.
(...)	Группировка. Группирует элементы в единое целое, которое может использоваться с символами *, +, ?, | и т. п. Также запоминает символы, соответствующие этой группе, для использования в последующих ссылках.
(?:...)	Только группировка. Группирует элементы в единое целое, но не запоминает символы, соответствующие этой группе.
n	Соответствует тем же символам, которые были найдены при первом соответствии группе с номером *n*. Группы – это подвыражения внутри скобок (возможно, вложенных). Номера группам присваиваются путем подсчета левых скобок слева направо. Группы, сформированные с помощью символов (?:, не нумеруются.

11.1.5. Задание позиции соответствия

Как описывалось ранее, многие элементы регулярного выражения соответствуют одному символу в строке. Например, \s соответствует одному символу-разделителю. Другие элементы регулярных выражений соответствуют позициям в тексте, а не самим символам. Например, \b соответствует границе слова – границе между \w (текстовый ASCII-символ) и \W (нетекстовый символ), или границе между текстовым ASCII-символом и началом или концом строки.[1] Такие элементы, как \b, не задают каких-либо символов, которые должны присутствовать в найденной строке, однако они определяют допустимые позиции для проверки соответствия. Иногда эти элементы называются *якорными элементами регулярных выражений*, т. к. они закрепляют шаблон за определенной позицией в строке. Чаще других используются такие якорные элементы, как ^ и $, привязывающие шаблоны соответственно к началу и концу строки.

Например, слово «JavaScript», находящееся на отдельной строке, можно найти с помощью регулярного выражения /^JavaScript$/. Отдельное слово «Java» (а не префикс, например в «JavaScript») можно поискать по шаблону /\sJava\s/, который требует наличия пробела[2] до и после слова. Но такое решение порождает две проблемы. Во-первых, оно найдет слово «Java», только если оно окружено пробелами с обеих сторон, и не сможет найти его в начале или в конце строки. Во-

[1] За исключением класса символов (квадратных скобок), где \b соответствует символу «забой».

[2] Точнее, любого разделителя. – *Примеч. науч. ред.*

вторых, когда этот шаблон действительно найдет соответствие, возвращаемая им строка будет содержать ведущие и замыкающие пробелы, а это не совсем то, что нам нужно. Поэтому вместо шаблона для реальных символов-разделителей `\s` мы воспользуемся шаблоном (или якорем) для границ слова `\b`. Получится следующее выражение: `/\bJava\b/`. Элемент `\B` представляет собою якорь для позиции, не являющейся границей слова. То есть шаблону `/\B[Ss]cript/` будут соответствовать слова «JavaScript» и «postscript» и не соответствовать слова «script» или «Scripting».

В JavaScript 1.5 (но не в JavaScript 1.2) в качестве якорных условий могут также выступать произвольные регулярные выражения. Если поместить выражение между символами `(?=` и `)`, оно станет условием на последующие символы, требующим, чтобы эти символы соответствовали указанному шаблону, но не включались в строку соответствия. Например, найти имя языка программирования JavaScript, но только там, где за ним следует двоеточие, можно посредством выражения `/[Jj]ava([Ss]cript)?(?=\:)/`. Этот шаблон найдет слово «JavaScript» в предложении «JavaScript: The Definitive Guide», но проигнорирует слово «Java» в предложении «Java in a Nutshell», т. к. после него нет двоеточия.

Если же ввести условие символами `(?!`, то это будет отрицательное условие на последующие символы, требующее, чтобы следующие символы не соответствовали указанному шаблону. Так, `/Java(?!Script)([A-Z]\w*)/` соответствует слову «Java», за которым следует прописная буква и произвольное количество дополнительных текстовых ASCII-символов, если только за «Java» не следует «Script». Этому шаблону соответствует «JavaBeans», но не соответствует «Javanese», соответствует «JavaScrip», но не соответствует «JavaScript» или «JavaScripter».

В табл. 11.5 приводится перечень якорных символов регулярных выражений.

Таблица 11.5. Якорные символы регулярных выражений

Символ	Значение
`^`	Соответствует началу строкового выражения или началу строки при многострочном поиске.
`$`	Соответствует концу строкового выражения или концу строки при многострочном поиске.
`\b`	Соответствует границе слова, т. е. соответствует позиции между символом `\w` и символом `\W` или между символом `\w` и началом или концом строки. (Однако обратите внимание, что `[\b]` соответствует символу забоя.)
`\B`	Соответствует позиции, не являющейся границей слов.
`(?=`*p*`)`	Положительное условие на последующие символы. Требует, чтобы последующие символы соответствовали шаблону *p*, но не включает эти символы в найденную строку.
`(?!`*p*`)`	Отрицательное условие на последующие символы. Требует, чтобы следующие символы не соответствовали шаблону *p*.

11.1.6. Флаги

И еще один, последний элемент грамматики регулярных выражений. Флаги регулярных выражений задают высокоуровневые правила соответствия шаблонам.

В отличие от остальной грамматики регулярных выражений, флаги указываются не между символами слэша, а после второго из них. JavaScript 1.2 поддерживает два флага. Флаг `i` указывает, что поиск по шаблону должен быть нечувствительным к регистру символов, а флаг `g` – что поиск должен быть глобальным, т. е. должны быть найдены все соответствия в строке. Оба флага могут быть объединены для выполнения глобального поиска без учета регистра символов.

Например, чтобы выполнить безразличный к регистру поиск первого вхождения слова «java» (или «Java», «JAVA» и т. д.), можно воспользоваться нечувствительным к регистру регулярным выражением `/\bjava\b/i`. А чтобы найти все вхождения этого слова в строке, надо добавить флаг `g`: `/\bjava\b/gi`.

JavaScript 1.5 поддерживает дополнительный флаг `m`, который выполняет поиск по шаблону в многострочном режиме. Если строковое выражение, в котором выполняется поиск, содержит символы перевода строк, то в этом режиме якорные символы `^` и `$`, помимо того, что они соответствуют началу и концу всего строкового выражения, также соответствуют началу и концу строки. Например, шаблону `/Java$/im` соответствует как слово «java», так и «Java\nis fun».

В табл. 11.6 приводится перечень флагов регулярных выражений. Заметим, что флаг `g` более подробно рассматривается далее в этой главе вместе с методами классов `String` и `RegExp`, используемых для фактической реализации поиска.

Таблица 11.6. Флаги регулярных выражений

Символ	Значение
`i`	Выполняет поиск, нечувствительный к регистру.
`g`	Выполняет глобальный поиск, т. е. находит все соответствия, а не останавливается после первого из них.
`m`	Многострочный режим. `^` соответствует началу строки или началу всего строкового выражения, а `$` – концу строки или всего выражения.

11.1.7. Средства регулярных выражений Perl, не поддерживаемые в JavaScript

Мы говорили, что ECMAScript v3 определяет относительно полное подмножество средств регулярных выражений из Perl 5. Развитые средства Perl, не поддерживаемые ECMAScript, включают следующее:

- флаги `s` (однострочный режим) и `x` (расширенный синтаксис);
- управляющие последовательности `\a`, `\e`, `\l`, `\u`, `\L`, `\U`, `\E`, `\Q`, `\A`, `\Z`, `\z` и `\G`;
- `(?<=` – положительное условие на предыдущие символы и `(?<!` – отрицательное условие на предыдущие символы;
- комментарий `(?#` и прочий расширенный синтаксис `(?`.

11.2. Методы класса String для поиска по шаблону

До этого момента мы обсуждали грамматику создаваемых регулярных выражений, но не рассматривали, как эти регулярные выражения могут реально использоваться в JavaScript-сценариях. В данном разделе мы обсудим методы объ-

екта String, в которых регулярные выражения применяются для поиска по шаблону, а также для поиска и замены. А затем продолжим разговор о поиске по шаблону с регулярными JavaScript-выражениями, рассмотрев объект RegExp, его методы и свойства. Обратите внимание: последующее обсуждение – это лишь обзор различных методов и свойств, относящихся к регулярным выражениям. Как обычно, полное описание можно найти в третьей части книги.

Строки поддерживают четыре метода, опирающихся на регулярные выражения. Простейший из них – метод search(). Он принимает в качестве аргумента регулярное выражение и возвращает либо позицию символа в начале первой найденной подстроки, либо -1, если соответствие не найдено. Например, следующий вызов возвращает 4:

```
"JavaScript".search(/script/i);
```

Если аргумент метода search() не является регулярным выражением, он сначала преобразуется путем передачи конструктору RegExp. Метод search() не поддерживает глобальный поиск и игнорирует флаг g в своем аргументе.

Метод replace() выполняет операцию поиска с заменой. Он принимает в качестве первого аргумента регулярное выражение, а в качестве второго – строку замены. Метод ищет в строке, для которой он вызван, соответствие указанному шаблону. Если регулярное выражение содержит флаг g, метод replace() заменяет все вхождения строки строкой замены, в противном случае он заменяет только первое найденное вхождение. Если первый аргумент метода replace() представляет собой не регулярное выражение, а строку, то метод выполняет буквальный поиск строки, а не преобразует ее в регулярное выражение с помощью конструктора RegExp(), как это делает метод search(). В качестве примера мы можем воспользоваться методом replace() для единообразной расстановки прописных букв в слове «JavaScript» для всей строки текста:

```
// Независимо от регистра символов заменяем словом в нужном регистре
text.replace(/javascript/gi, "JavaScript");
```

Однако метод replace() представляет собой более мощное средство, чем можно судить по этому примеру. Вспомните, что подвыражения в скобках, находящиеся внутри регулярного выражения, нумеруются слева направо, и что регулярное выражение запоминает текст, соответствующий каждому из подвыражений. Если в строке замены присутствует знак $ с цифрой, метод replace() заменяет эти два символа текстом, соответствующим указанному подвыражению. Это очень полезная возможность. Мы можем использовать ее, например, для замены прямых кавычек в строке типографскими кавычками, которые имитируются ASCII-символами:

```
// Цитата - это кавычка, за которой следует любое число символов,
// отличных от кавычек (их мы запоминаем), за этими символами следует
// еще одна кавычка.
var quote = /"([^"]*)"/g;
// Заменяем прямые кавычки типографскими и оставляем без
// изменений содержимое цитаты, хранящееся в $1.
text.replace(quote, "``$1''");
```

Метод replace() предоставляет и другие ценные возможности, о которых рассказывается в третьей части книги при описании конструкции String.replace(). Са-

мое важное, что следует отметить, – второй аргумент replace() может быть функцией, динамически вычисляющей строку замены.

Метод match() – это наиболее общий из методов класса String, опирающихся на регулярные выражения. Он принимает в качестве единственного аргумента регулярное выражение (или преобразует свой аргумент в регулярное выражение, передав его конструктору RegExp()) и возвращает массив, содержащий результаты поиска. Если в регулярном выражении установлен флаг g, метод возвращает массив всех соответствий, присутствующих в строке. Например:

```
"1 плюс 2 равно 3".match(/\d+/g) // возвращает ["1", "2", "3"]
```

Если регулярное выражение не содержит флаг g, метод match() не выполняет глобальный поиск; он просто ищет первое соответствие. Однако match() возвращает массив, даже когда метод не выполняет глобальный поиск. В этом случае первый элемент массива – это найденная строка, а все оставшиеся элементы представляют собой подвыражения регулярного выражения. Поэтому если match() возвращает массив a, то a[0] будет содержать найденную строку целиком, a[1] – подстроку, соответствующую первому подвыражению, и т. д. Проводя параллель с методом replace(), можно сказать, что в a[*n*] заносится содержимое $*n*.

В качестве примера рассмотрите следующий программный код, выполняющий разбор URL-адреса:

```
var url = /(\w+):\/\/([\w.]+)\/(\S*)/;
var text = "Посетите мою домашнюю страницу http://www.isp.com/~david";
var result = text.match(url);
if (result != null) {
    var fullurl = result[0];   // Содержит "http://www.isp.com/~david"
    var protocol = result[1]; // Содержит "http"
    var host = result[2];      // Содержит "www.isp.com"
    var path = result[3];      // Содержит "~david"
}
```

И наконец, имеется еще одна особенность метода match(), о которой следует знать. Возвращаемый им массив имеет, как и все массивы, свойство length. Однако когда match() вызывается с регулярным выражением без флага g, возвращаемый массив имеет еще два свойства: index, содержащее номер позиции символа внутри строки, с которого начинается соответствие, и input, являющееся копией строки, в которой выполнялся поиск. То есть в приведенном примере значение result.index будет равно 31, т. к. найденный URL-адрес начинается в тексте с 31-й позиции. Свойство result.input должно содержать ту же строку, что и переменная text. Для регулярного выражения r, в котором не установлен флаг g, вызов s.match(r) возвращает то же значение, что и r.exec(s). Немного позднее в этой главе мы обсудим метод RegExp.exec().

Последний из методов объекта String, в котором используются регулярные выражения, – это split(). Этот метод разбивает строку, для которой он вызван, на массив подстрок, используя аргумент в качестве разделителя. Например:

```
"123,456,789".split(","); // Возвращает ["123","456","789"]
```

Метод split() может также принимать в качестве аргумента регулярное выражение. Это делает метод более мощным. Например, мы можем указать разделитель, допускающий произвольное число пробельных символов с обеих сторон:

```
"1,2, 3 , 4 ,5".split(/\s*,\s*/); // Возвращает ["1","2","3","4","5"]
```

Метод split() имеет и другие возможности. Полное описание приведено в третьей части книги при описании конструкции String.split().

11.3. Объект RegExp

Как было упомянуто в начале этой главы, регулярные выражения представлены в виде объектов RegExp. Помимо конструктора RegExp(), объекты RegExp поддерживают три метода и несколько свойств. Особенность класса RegExp состоит в том, что он определяет как свойства класса (или статические свойства), так и свойства экземпляра. То есть он определяет глобальные свойства, принадлежащие конструктору RegExp(), а также свойства, принадлежащие конкретным объектам RegExp. Методы поиска и свойства класса RegExp описаны в следующих двух подразделах.

Конструктор RegExp() принимает один или два строковых аргумента и создает новый объект RegExp. Первый аргумент конструктора – это строка, содержащая тело регулярного выражения, т. е. текст, который должен находиться между косыми чертами в литерале регулярного выражения. Обратите внимание: в строковых литералах и регулярных выражениях для обозначения управляющих последовательностей используется символ \, поэтому передавая конструктору RegExp() регулярное выражение в виде строкового литерала, необходимо заменить все символы \ символами \\. Второй аргумент RegExp() может отсутствовать. Если он указан, то задает флаги регулярного выражения. Это должен быть один из символов g, i, m либо комбинация этих символов. Например:

```
// Находит все пятизначные числа в строке. Обратите внимание на использование
// в этом примере символов \\
var zipcode = new RegExp("\\d{5}", "g");
```

Конструктор RegExp() удобно использовать, когда регулярное выражение создается динамически и поэтому не может быть представлено с помощью синтаксиса литералов регулярных выражений. Например, чтобы найти строку, введенную пользователем, надо создать регулярное выражение во время выполнения с помощью RegExp().

11.3.1. Методы класса RegExp для поиска по шаблону

Объекты RegExp определяют два метода, выполняющие поиск по шаблону; они ведут себя аналогично методам класса String, описанным ранее. Основной метод класса RegExp, используемый для поиска по шаблону, – это exec(). Он похож на упоминавшийся метод match() класса String, за исключением того, что является методом класса RegExp, принимающим в качестве аргумента строку, а не методом класса String, принимающим аргумент RegExp. Метод exec() исполняет регулярное выражение для указанной строки, т. е. ищет в строке соответствие. Если соответствие не найдено, метод возвращает null. Однако если соответствие найдено, он возвращает такой же массив, как массив, возвращаемый методом match() для поиска без флага g. Нулевой элемент массива содержит строку, соответствующую регулярному выражению, а все последующие элементы – подстроки, соответствующие всем подвыражениям. Кроме того, свойство index содержит но-

мер позиции символа, которым начинается соответствующий фрагмент, а свойство input ссылается на строку, в которой выполнялся поиск.

В отличие от match(), метод exec() возвращает массив, структура которого не зависит от наличия в регулярном выражении флага g. Вспомните, что при передаче глобального регулярного выражения метод match() возвращает массив найденных соответствий. А exec() всегда возвращает одно соответствие, но предоставляет о нем полную информацию. Когда exec() вызывается для регулярного выражения, содержащего флаг g, метод устанавливает свойство lastIndex объекта регулярного выражения равным номеру позиции символа, следующего непосредственно за найденной подстрокой. Когда метод exec() вызывается для того же регулярного выражения второй раз, он начинает поиск с символа, позиция которого указана в свойстве lastIndex. Если exec() не находит соответствия, свойство lastIndex равно 0. (Вы также можете установить lastIndex в ноль в любой момент, что следует делать во всех тех случаях, когда вы завершаете поиск до того, как нашли последнее соответствие в одной строке и начинаете поиск в другой строке с тем же объектом RegExp.) Это особое поведение позволяет нам вызывать exec() повторно для перебора всех соответствий регулярному выражению в строке. Например:

```
var pattern = /Java/g;
var text = "JavaScript - это более забавная штука, чем Java!";
var result;
while((result = pattern.exec(text)) != null) {
    alert("Найдено `" + result[0] + "`" +
          " в позиции " + result.index +
          "; следующий поиск начинается с " + pattern.lastIndex);
}
```

Еще один метод объекта RegExp — test(), который намного проще метода exec(). Он принимает строку и возвращает true, если строка соответствует регулярному выражению:

```
var pattern = /java/i;
pattern.test("JavaScript"); // Возвращает true
```

Вызов test() эквивалентен вызову exec(), возвращающему true, если exec() возвращает не null. По этой причине метод test() ведет себя так же, как метод exec() при вызове для глобального регулярного выражения: он начинает искать указанную строку с позиции, заданной свойством lastIndex, и если находит соответствие, устанавливает свойство lastIndex равным номеру позиции символа, непосредственно следующего за найденным соответствием. Поэтому мы можем сформировать с помощью метода test() цикл обхода строки так же, как с помощью метода exec().

Методы search(), replace() и match() класса String не задействуют свойство lastIndex, в отличие от методов exec() и test(). На самом деле методы класса String просто сбрасывают lastIndex в 0. Если мы используем exec() или test() с шаблоном, в котором установлен флаг g, и выполняем поиск в нескольких строках, то мы должны либо найти все соответствия в каждой строке, чтобы свойство lastIndex автоматически сбросилось в ноль (это происходит, когда последний поиск оказывается неудачным), либо явно установить свойство lastIndex, равным нулю. Если этого не сделать, то поиск в новой строке может начаться с некоторой

произвольной позиции, а не с начала. И наконец, помните, что особое поведение свойства `lastIndex` относится только к регулярным выражениям с флагом `g`. Методы `exec()` и `test()` игнорируют свойство `lastIndex` объектов `RegExp`, в которых отсутствует флаг `g`.

11.3.2. Свойства экземпляра RegExp

Каждый объект `RegExp` имеет пять свойств. Свойство `source` — это доступная только для чтения строка, содержащая текст регулярного выражения. Свойство `global` — это доступное только для чтения логическое значение, определяющее, имеется ли в регулярном выражении флаг `g`. Свойство `ignoreCase` — это доступное только для чтения логическое значение, определяющее, имеется ли в регулярном выражении флаг `i`. Свойство `multiline` — это доступное только для чтения логическое значение, определяющее, имеется ли в регулярном выражении флаг `m`. И последнее свойство `lastIndex` — это целое число, доступное для чтения и записи. Для шаблонов с флагом `g` это свойство содержит номер позиции в строке, с которой должен быть начат следующий поиск. Как описано в предыдущем разделе, оно используется методами `exec()` и `test()`.

12

Разработка сценариев для Java-приложений

Несмотря на свое название язык JavaScript не имеет ничего общего с языком Java. Разве только некоторое синтаксическое подобие, обусловленное тем, что оба языка программирования заимствовали синтаксис языка программирования C. Но при более глубоком рассмотрении оба языка оказываются совершенно разными. Однако в результате своего развития JavaScript теперь может использоваться в программах, написанных на языке Java.[1] Этот факт учитывается в реализации Java 6, в составе которой распространяется встроенный интерпретатор JavaScript, что позволяет без труда встраивать JavaScript-сценарии в любое Java-приложение. Кроме того, некоторые интерпретаторы JavaScript (как тот, что поставляется в составе Java 6) обладают функциональными возможностями, позволяющими JavaScript-сценариям взаимодействовать с Java-объектами, устанавливать и запрашивать значения свойств и вызывать методы объектов.

Эта глава в первую очередь описывает, как внедрять интерпретатор JavaScript в приложения, написанные на языке Java 6, и как запускать JavaScript-сценарии из этих приложений. Затем продемонстрировано, как организовать непосредственное взаимодействие с Java-объектами из JavaScript-сценариев.

К теме Java мы еще вернемся в главе 23, где будет говориться о Java-апплетах и модулях расширения Java для веб-броузеров.

12.1. Встраивание JavaScript

Мы приближаемся к концу первой части книги, где описываются основы JavaScript. Вторая часть этой книги полностью посвящена использованию JavaScript в веб-броузерах. Однако прежде чем приступить к обсуждению этой темы,

---

[1] Эта глава предназначена для Java-программистов, и многие примеры в ней написаны целиком или частично на языке Java. Если вы не знакомы с этим языком программирования, можете просто пропустить эту главу.

коротко рассмотрим вопрос встраивания JavaScript в другие приложения. Необходимость встраивания JavaScript в приложения обычно диктуется стремлением дать пользователю возможность подстраивать приложение под свои нужды с помощью сценариев. Веб-броузер Firefox, например, позволяет управлять пользовательским интерфейсом с помощью JavaScript-сценариев. Многие другие приложения, обладающие широчайшими возможностями, поддерживают настройку с помощью языков сценариев того или иного вида.

В рамках проекта Mozilla реализовано два интерпретатора JavaScript, распространяемых с открытыми исходными текстами. Интерпретатор SpiderMonkey – оригинальная версия JavaScript, реализованная на языке C. Версия Rhino реализована на языке Java. Обе версии имеют прикладной интерфейс для встраивания. Если возникнет необходимость добавить возможность управлять с помощью JavaScript-сценариев приложением, написанным на языке C, выбирайте версию SpiderMonkey. В случае необходимости добавить возможность управлять Java-приложением с помощью сценариев, следует выбрать версию Rhino. Подробнее узнать об использовании этих интерпретаторов в своих приложениях можно по адресам *http://www.mozilla.org/js/spidermonkey* и *http://www.mozilla.org/rhino*.

С появлением Java 6.0 стало еще проще ввести поддержку JavaScript-сценариев в Java-приложения. Именно эта тема и является предметом обсуждения данной главы. В составе Java 6 появились новый пакет `javax.script`, реализующий обобщенный интерфейс для подключения языков сценариев, и встроенная версия интерпретатора JavaScript – Rhino, которая использует этот пакет в своей работе.[1]

В примере 12.1 демонстрируются основы использования пакета `javax.script`: в этом примере создаются объект `ScriptEngine`, который представляет собой экземпляр интерпретатора JavaScript, и объект `Bindings`, хранящий значения JavaScript-переменных. После этого запускается сценарий, хранящийся во внешнем файле, за счет передачи объекта-потока `java.io.Reader` и связующего объекта `Bindings` методу `eval()` объекта `ScriptEngine`. Метод `eval()` возвращает результат работы сценария или генерирует исключение `ScriptException`, если в процессе исполнения сценария возникла ошибка.

Пример 12.1. Программа на языке Java, запускающая JavaScript-сценарии

```
import javax.script.*;
import java.io.*;

// Запускает файл JavaScript-сценария и выводит результаты его работы
public class RunScript {
    public static void main(String[] args) throws IOException {
        // Создать экземпляр интерпретатора, или "ScriptEngine", для запуска сценария.
        ScriptEngineManager scriptManager = new ScriptEngineManager();
        ScriptEngine js = scriptManager.getEngineByExtension("js");

        // Файл запускаемого сценария
        String filename = null;
```

[1] К моменту написания этих строк реализация Java 6 еще находилась в стадии разработки. Пакет `javax.script` уже в достаточной степени проработан, чтобы его можно было здесь описывать, однако есть некоторая вероятность, что прикладной интерфейс пакета может претерпеть какие-либо изменения в окончательной версии.

```
        // Объект Bindings – это таблица символов, или пространство имен,
        // для интерпретатора. Он хранит имена и значения переменных
        // и делает их доступными в сценарии.
        Bindings bindings = js.createBindings( );

        // Обработка аргументов. Строка может содержать произвольное число аргументов
        // вида -Dname=value, которые определяют переменные для использования в сценарии.
        // Любые аргументы, начинающиеся не с -D, воспринимаются как имена файлов
        for(int i = 0; i < args.length; i++) {
            String arg = args[i];
            if (arg.startsWith("-D")) {
                int pos = arg.indexOf('=');
                if (pos == -1) usage();
                String name = arg.substring(2, pos);
                String value = arg.substring(pos+1);
                // Обратите внимание: все объявляемые переменные являются строковыми.
               // Сценарии могут преобразовывать их в другие типы по мере необходимости.
                // Кроме того, существует возможность передавать java.lang.Number,
                // java.lang.Boolean и любые другие объекты Java или значение null.
                bindings.put(name, value);
            }
            else {
                if (filename != null) usage();// файл должен быть единственным
                filename = arg;
            }
        }
        // Убедиться, что строка аргументов содержала имя файла.
        if (filename == null) usage();

        // Добавить одну или более связок с помощью специальной
        // зарезервированной переменной, чтобы передать интерпретатору имя
        // файла сценария, который следует исполнить.
        // Это позволит получать более информативные сообщения об ошибках.
        bindings.put(ScriptEngine.FILENAME, filename);

        // Создать объект-поток для чтения файла сценария.
        Reader in = new FileReader(filename);

        try {
           // Выполнить сценарий, используя объекты с переменными, и получить результат.
            Object result = js.eval(in, bindings);
            // Вывести результат.
            System.out.println(result);
        }
        catch(ScriptException ex) {
            // Или вывести сообщение об ошибке.
            System.out.println(ex);
        }
    }

    static void usage() {
        System.err.println(
         "Порядок использования: java RunScript [-Dname=value...] script.js");
        System.exit(1);
    }
}
```

Объект Bindings, создаваемый в этом примере, не является статическим – любые переменные, создаваемые JavaScript-сценарием, хранятся в этом объекте. В примере 12.2 приводится более практичный пример на языке Java. Здесь объект Bindings сохраняется в объекте ScriptContext на более высоком уровне области видимости, что позволяет обращаться к переменным, но новые переменные в объекте Bindings не сохраняются. Пример представляет собой реализацию простейшего класса обработки файлов с возможностью настройки: параметры хранятся в текстовом файле в виде пар имя/значение, получить которые можно с помощью описываемого здесь класса Configuration. Значения могут быть строковыми, числовыми или логическими, а если значение окружено фигурными скобками, оно передается интерпретатору JavaScript для вычисления. Интересно, как объект java.util.Map, хранящий пары имя/значение, обернут в объект SimpleBindings, благодаря чему интерпретатор JavaScript может также обращаться к значениям других переменных, определенных в том же самом файле.[1]

Пример 12.2. Класс обработки файлов с возможностью настройки, который интерпретирует JavaScript-выражения

```
import javax.script.*;
import java.util.*;
import java.io.*;

/**
 * Этот класс напоминает java.util.Properties, но позволяет определять
 * значения свойств в виде выражений на языке JavaScript.
 */
public class Configuration {
    // Здесь по умолчанию будут храниться пары имя/значение
    Map<String,Object> defaults = new HashMap<String,Object>( );

    // Методы доступа к значениям параметров
    public Object get(String key) { return defaults.get(key); }
    public void put(String key, Object value) { defaults.put(key, value); }

    // Инициализировать содержимое объекта Map из файла с парами имя/значение.
    // Если значение окружено фигурными скобками, оно должно вычисляться
    // как выражение на языке JavaScript.
    public void load(String filename) throws IOException, ScriptException {
        // Создать экземпляр интерпретатора
        ScriptEngineManager manager = new ScriptEngineManager();
        ScriptEngine engine = manager.getEngineByExtension("js");

        // Использовать собственные пары имя/значение в качестве JavaScript-переменных.
        Bindings bindings = new SimpleBindings(defaults);

        // Создать контекст исполнения сценариев.
        ScriptContext context = new SimpleScriptContext();
```

[1] Как будет показано далее в этой же главе, сценарию на языке JavaScript доступны любые общедоступные члены любых общедоступных классов. Поэтому из соображений безопасности программный Java-код, запускающий пользовательские сценарии, обычно исполняется с ограниченными привилегиями. Однако обсуждение системы безопасности Java выходит за рамки темы этой книги.

```
        // Определить контекст переменных, чтобы они были доступны из сценария,
        // но чтобы переменные, создаваемые в сценарии, не попадали в объект Map
        context.setBindings(bindings, ScriptContext.GLOBAL_SCOPE);

        BufferedReader in = new BufferedReader(new FileReader(filename));
        String line;
        while((line = in.readLine( )) != null) {
            line = line.trim( ); // отбросить ведущие и завершающие пробелы
            if (line.length( ) == 0) continue;   // пропустить пустые строки
            if (line.charAt(0) == '#') continue; // пропустить комментарии

            int pos = line.indexOf(":");
            if (pos == -1)
                throw new IllegalArgumentException("syntax: " + line);

            String name = line.substring(0, pos).trim();
            String value = line.substring(pos+1).trim();
            char firstchar = value.charAt(0);
            int len = value.length( );
            char lastchar = value.charAt(len-1);

            if (firstchar == '"' && lastchar == '"') {
                // Строки в двойных кавычках - это строковые значения
                defaults.put(name, value.substring(1, len-1));
            }
            else if (Character.isDigit(firstchar)) {
                // Если значение начинается с цифры, попробовать
                // интерпретировать его как число
                try {
                    double d = Double.parseDouble(value);
                    defaults.put(name, d);
                }
                catch(NumberFormatException e) {
                    // Ошибка. Это не число. Сохранить как строку
                    defaults.put(name, value);
                }
            }
            else if (value.equals("true")) // логическое значение
                defaults.put(name, Boolean.TRUE);
            else if (value.equals("false"))
                defaults.put(name, Boolean.FALSE);
            else if (value.equals("null"))
                defaults.put(name, null);
            else if (firstchar == '{' && lastchar == '}') {
                // Значения в фигурных скобках вычисляются как JavaScript-выражения
                String script = value.substring(1, len-1);
                Object result = engine.eval(script, context);
                defaults.put(name, result);
            }
            else {
                // По умолчанию просто сохранить значение как строку
                defaults.put(name, value);
            }
        }
    }
```

```
    // Простейший тест класса
    public static void main(String[] args) throws IOException, ScriptException
    {
        Configuration defaults = new Configuration();
        defaults.load(args[0]);
        Set<Map.Entry<String,Object>> entryset = defaults.defaults.entrySet();
        for(Map.Entry<String,Object> entry : entryset) {
            System.out.printf("%s: %s%n", entry.getKey(), entry.getValue());
        }
    }
}
```

12.1.1. Преобразование типов с помощью пакета javax.script

Всякий раз, когда программный код, написанный на одном языке программирования, обращается к программному коду, написанному на другом языке программирования, необходимо учитывать, как отображаются типы данных в одном языке программирования на типы данных в другом языке программирования.[1] Предположим, что имеется необходимость присвоить значения типов `java.lang.String` и `java.lang.Integer` переменным в объекте `Bindings`. Когда JavaScript-сценарий обратится к этим переменным, значения каких типов он обнаружит? А если результатом работы JavaScript-сценария будет логическое значение, значение какого типа вернет метод `eval()`?

В случае Java и JavaScript ответить на этот вопрос достаточно просто. Когда в объекте `Bindings` сохраняется Java-объект (просто не существует способа сохранения значений элементарных типов), он преобразуется в JavaScript-значение в соответствии со следующими правилами:

- Объекты логических значений преобразуются в логический JavaScript-тип.
- Все объекты `java.lang.Number` преобразуются в JavaScript-числа.
- Java-объекты `Character` и `String` преобразуются в JavaScript-строки.
- Java-значение `null` преобразуется в JavaScript-значение `null`.
- Все остальные Java-объекты просто обертываются в JavaScript-объект `JavaObject` (подробнее о типе `JavaObject` рассказывается далее в этой главе, где речь идет об организации взаимодействий с Java-объектами из JavaScript-сценариев).

Есть несколько замечаний по поводу преобразования чисел. Все Java-числа преобразуются в JavaScript-числа. Это относится к типам `Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`, а также `java.math.BigInteger` и `java.math.BigDouble`. Специальные вещественные значения, такие как `Infinity` и `NaN`, поддерживаются обоими языками и легко преобразуются один в другой. Обратите внимание, числовой JavaScript-тип основан на 64-разрядном формате представления вещественных чисел и соот-

[1] И вообще следует учитывать наличие типов, представимых в одном из языков, но отсутствующих в другом. В этом случае может оказаться необходимым моделирование отсутствующего типа искусственным пользовательским типом. – *Примеч. науч. ред.*

ветствует Java-типу double. Не все значения Java-типа long могут быть преобразованы в тип double без потери точности, поэтому в случае передачи JavaScript-сценариям значений типа long данные могут быть потеряны. То же относится к типам BigInteger и BigDecimal: младшие разряды числа могут оказаться утраченными, если числовые значения в Java будут иметь более высокую точность, чем может быть представлена в JavaScript. Или если числовое значение в Java будет больше, чем Double.MAX_VALUE, оно преобразуется в JavaScript-значение Infinity.

Преобразования в обратном направлении выполняются аналогичным образом. Когда JavaScript-сценарий сохраняет значение в переменной (т. е. в объекте Bindings) или вычисляется значение JavaScript-выражения, на стороне Java-программы преобразование типов значений выполняется в соответствии со следующими правилами:

- JavaScript-значения логического типа преобразуются в Java-объекты Boolean.
- JavaScript-значения строкового типа преобразуются в Java-объекты String.
- JavaScript-значения числового типа преобразуются в Java-объекты Double. Значения Infinity и NaN преобразуются в соответствующие Java-значения.
- JavaScript-значения null и undefined преобразуются в Java-значение null.
- Объекты и массивы JavaScript преобразуются в Java-объекты неопределенного типа. Эти значения могут передаваться обратно JavaScript-сценарию, но имеют прикладной интерфейс, который не предназначен для использования в программах на языке Java. Обратите внимание: объекты-обертки типа String, Boolean и Number языка JavaScript преобразуются в Java-объекты неопределенного типа, а не в значения соответствующего им типа на стороне Java.

12.1.2. Компиляция сценариев

Если возникает необходимость выполнить один и тот же сценарий несколько раз (возможно с разными наборами переменных), гораздо эффективнее скомпилировать сценарий однократно, а затем вызывать уже скомпилированную версию. Сделать это можно, например, следующим образом:

```
// Это текст сценария, который требуется скомпилировать.
String scripttext = "x * x";

// Создать экземпляр интерпретатора.
ScriptEngineManager scriptManager = new ScriptEngineManager( );
ScriptEngine js = scriptManager.getEngineByExtension("js");

// Привести его к типу интерфейса Compilable, чтобы получить возможность компиляции.
Compilable compiler = (Compilable)js;

// Скомпилировать сценарий в представление, которое даст возможность.
// запускать его многократно
CompiledScript script = compiler.compile(scripttext);

// Теперь запустить сценарий пять раз, используя всякий раз разные значения переменной x
Bindings bindings = js.createBindings( );
for(int i = 0; i < 5; i++) {
    bindings.put("x", i);
    Object result = script.eval(bindings);
    System.out.printf("f(%d) = %s%n", i, result);
}
```

12.1.3. Вызов JavaScript–функций

Помимо всего прочего пакет `javax.script` **позволяет выполнить сценарий единожды, а затем многократно вызывать функции, определенные в этом сценарии. Сделать это можно, например, следующим образом:**

```
// Создать экземпляр интерпретатора, или "ScriptEngine", для запуска сценария
ScriptEngineManager scriptManager = new ScriptEngineManager();
ScriptEngine js = scriptManager.getEngineByExtension("js");

// Запустить сценарий. Результат его работы отбрасывается, поскольку
// интерес для нас представляет только определение функции.
js.eval("function f(x) { return x*x; }");

// Теперь можно вызвать функцию, объявленную в сценарии.
try {
    // Привести ScriptEngine к типу интерфейса Invokable,
    // чтобы получить возможность вызова функций.
    Invocable invocable = (Invocable) js;
    for(int i = 0; i < 5; i++) {
        Object result = invocable.invoke("f", i);     // Вызов функции f(i)
        System.out.printf("f(%d) = %s%n", i, result); // Вывод результата
    }
}
catch(NoSuchMethodException e) {
    // Эта часть программы выполняется, если сценарий не содержит
    // определение функции с именем "f".
    System.out.println(e);
}
```

12.1.4. Реализация интерфейсов в JavaScript

Интерфейс `Invocable`**, продемонстрированный в предыдущем разделе, помимо всего прочего позволяет реализовать интерфейсы на языке JavaScript. В примере 12.3 используется программный JavaScript-код из файла *listener.js* для реализации интерфейса** `java.awt.event.KeyListener`**.**

Пример 12.3. Реализация Java-интерфейса средствами JavaScript-кода

```
import javax.script.*;
import java.io.*;
import java.awt.event.*;
import javax.swing.*;

public class Keys {
    public static void main(String[] args) throws ScriptException, IOException
    {
        // Создать экземпляр интерпретатора, или "ScriptEngine", для запуска сценария.
        ScriptEngineManager scriptManager = new ScriptEngineManager( );
        ScriptEngine js = scriptManager.getEngineByExtension("js");

        // Запустить сценарий. Результат его работы отбрасывается, поскольку
        // интерес для нас представляют только определения функций.
        js.eval(new FileReader("listener.js"));

       // Привести к типу Invocable и получить объект, реализующий интерфейс KeyListener
        Invocable invocable = (Invocable) js;
        KeyListener listener = invocable.getInterface(KeyListener.class);
```

```
        // Теперь использовать KeyListener при создании простейшего
        // графического интерфейса пользователя.
        JFrame frame = new JFrame("Keys Demo");
        frame.addKeyListener(listener);
        frame.setSize(200, 200);
        frame.setVisible(true);
    }
}
```

Реализация интерфейса на языке JavaScript просто означает определение функций с именами, совпадающими с именами тех методов, которые определены в интерфейсе. Вот пример простого сценария, реализующего интерфейс `KeyListener`:

```
function keyPressed(e) {
    print("нажата клавиша: " + String.fromCharCode(e.getKeyChar()));
}
function keyReleased(e) { /* ничего не делает */ }
function keyTyped(e)    { /* ничего не делает */ }
```

Обратите внимание: объявленная здесь JavaScript-функция `keyPressed()` принимает в качестве аргумента объект `java.awt.event.KeyEvent` и фактически вызывает метод Java-объекта. В следующем разделе рассказывается о том, как это делается.

12.2. Взаимодействие с Java-кодом

Интерпретаторы JavaScript часто поддерживают возможность обращения к полям и вызова методов Java-объектов. Если сценарий обладает средствами доступа к объектам через аргументы методов или объект `Bindings`, он может взаимодействовать с Java-объектами практически так же, как с JavaScript-объектами. И даже если сценарию не передается никаких ссылок на Java-объекты, он может создавать собственные Java-объекты. Netscape стала первой компанией, реализовавшей возможность взаимодействия JavaScript-сценариев с Java-приложениями, когда включила в свой интерпретатор SpiderMonkey средства взаимодействия с Java-апплетами в веб-броузерах. В Netscape эта технология получила название LiveConnect. Интерпретатор Rhino, а также реализация JScript, созданная компанией Microsoft, были адаптированы в соответствии с синтаксисом LiveConnect, поэтому название LiveConnect используется на протяжении этой главы для обозначения любой реализации, объединяющей JavaScript и Java.

Начнем этот раздел с обзора некоторых характеристик LiveConnect. В последующих подразделах приводится более подробное описание технологии LiveConnect.

Обратите внимание: Rhino и SpiderMonkey реализуют несколько отличающиеся версии LiveConnect. Функциональные возможности, описываемые здесь, относятся к интерпретатору Rhino и могут использоваться в сценариях, встроенных в Java 6. Интерпретатор SpiderMonkey, реализующий лишь часть этих возможностей, рассматривается в главе 23.

Когда Java-объект передается JavaScript-сценарию через объект `Bindings` или в качестве аргумента функции, JavaScript-код может работать с ним практически так же, как если бы это был обычный JavaScript-объект. Все общедоступные поля и методы Java-объекта становятся доступны как свойства объекта-обертки языка JavaScript. Например, допустим, что в сценарий передается Java-объект,

выполняющий рисование диаграмм. Теперь предположим, что в этом объекте объявлено поле с именем `lineColor` типа `String`, и что JavaScript-сценарий сохраняет ссылку на этот объект в переменной с именем `chart`. Тогда JavaScript-код может обращаться к этому полю следующим образом:

```
var chartcolor = chart.lineColor; // Чтение поля Java-объекта.
chart.lineColor = "#ff00ff";      // Запись в поле Java-объекта.
```

Более того, JavaScript-сценарий может даже работать с полями-массивами. Допустим, что объект рисования диаграмм определяет следующие два поля (на языке Java):

```
public int numPoints;
public double[] points;
```

Тогда JavaScript-программа может обращаться к этим полям следующим образом:

```
for(var i = 0; i < chart.numPoints; i++)
    chart.points[i] = i*i;
```

Помимо работы с полями Java-объектов, JavaScript-сценарии могут вызывать методы этих объектов. Например, предположим, что объект рисования диаграмм имеет метод с именем `redraw()`. Данный метод не имеет аргументов и просто сообщает объекту, что содержимое массива `points[]` изменилось и следует перерисовать диаграмму. JavaScript-сценарий может вызвать этот метод, как если бы это был метод JavaScript-объекта:

```
chart.redraw();
```

Кроме того, JavaScript-сценарий может передавать методам аргументы и получать возвращаемые значения. Преобразование типов значений аргументов и возвращаемых значений производится по мере необходимости. Предположим, что объект рисования диаграмм объявляет следующие методы:

```
public void setDomain(double xmin, double xmax);
public void setChartTitle(String title);
public String getXAxisLabel();
```

Тогда JavaScript-сценарий может вызывать эти методы следующим образом:

```
chart.setDomain(0, 20);
chart.setChartTitle("y = x*x");
var label = chart.getXAxisLabel();
```

Наконец следует отметить, что возвращаемыми значениями Java-методов могут быть Java-объекты, и JavaScript-сценарий может обращаться к общедоступным полям и вызывать общедоступные методы этих объектов. Кроме того, из JavaScript-кода можно даже передавать Java-объекты в виде аргументов Java-методов. Допустим, что объект рисования диаграмм содержит метод с именем `getXAxis()`, который возвращает другой Java-объект – экземпляр класса `Axis`. Допустим также, что этот объект имеет еще один метод с именем `setYAxis()`, который принимает в качестве аргумента экземпляр класса `Axis`. И наконец, допустим, что класс `Axis` объявляет метод с именем `setTitle()`. Тогда обращаться к этим методам из JavaScript можно следующим образом:

```
var xaxis = chart.getXAxis(); // Получить объект Axis
var newyaxis = xaxis.clone(); // Создать его копию
```

```
newyaxis.setTitle("Y");        // Вызвать его метод...
chart.setYAxis(newyaxis);      // ...и передать его другому методу
```

Технология LiveConnect позволяет JavaScript-коду создавать собственные Java-объекты, т. е. JavaScript-сценарий может взаимодействовать с Java-объектами, даже не получая их извне.

Глобальный символ `Packages` предоставляет возможность доступа к любым Java-объектам, которые известны интерпретатору JavaScript. Выражение `Package.java.lang` является ссылкой на пакет `java.lang`, а выражение `Package.java.lang.System` — на класс `java.lang.System`. Для удобства было введено еще одно глобальное имя `java`, которое является сокращением от `Package.java`. Вызвать статический метод класса `java.lang.System` из JavaScript-сценария можно следующим образом:

```
// Вызвать статический Java-метод System.getProperty()
var javaVersion = java.lang.System.getProperty("java.version");
```

Этим возможности LiveConnect не ограничиваются, т. к. в JavaScript-сценариях допускается применять оператор `new` для создания новых экземпляров Java-классов. Для примера рассмотрим фрагмент JavaScript-сценария, в котором создается и отображается элемент графического интерфейса из Java Swing:

```
// Определить сокращение для обозначения иерархии пакета javax.*
var javax = Packages.javax;

// Создать некоторые Java-объекты.
var frame = new javax.swing.JFrame("Hello World");
var button = new javax.swing.JButton("Hello World");
var font = new java.awt.Font("SansSerif", java.awt.Font.BOLD, 24);

// Вызвать методы новых объектов.
frame.add(button);
button.setFont(font);
frame.setSize(200, 200);
frame.setVisible(true);
```

Чтобы понять, как LiveConnect организует взаимодействие между JavaScript- и Java-кодом, необходимо понимать, какие типы данных языка JavaScript используются в LiveConnect. Эти типы данных описываются в следующих разделах.

12.2.1. Класс JavaPackage

Пакет в языке программирования Java — это набор взаимосвязанных Java-классов. Класс `JavaPackage` — это тип данных языка JavaScript, представляющий Java-пакет. Свойствами `JavaPackage` являются классы (классы представляются в виде класса `JavaClass`, о котором мы вскоре поговорим), а также любые другие пакеты, входящие в состав данного пакета. Классы в `JavaPackage` не поддаются перечислению, вследствие этого невозможно использовать цикл `for/in` для выяснения содержимого пакета.

Все объекты `JavaPackage` содержатся внутри родительского объекта `JavaPackage`. Глобальное свойство с именем `Packages` — это объект `JavaPackage` верхнего уровня, который выступает в качестве корня этого дерева иерархии пакетов. Данный объект обладает такими свойствами, как `java` и `javax`, которые также являются объектами `JavaPackage`, представляющими различные иерархии Java-классов, доступных интерпретатору. Например, объект класса `JavaPackage` — это `Pack-`

ages.java, он содержит объект класса JavaPackage — Packages.java.awt. Для удобства глобальный объект имеет еще одно свойство java, которое является сокращением Packages.java. Таким образом, вместо того чтобы вводить длинное имя Packages.java.awt, можно просто ввести java.awt.

Продолжая наш пример, скажем, что java.awt — это объект JavaPackage, содержащий объекты JavaClass, такие как класс java.awt.Button. Кроме того он содержит еще один объект JavaPackage — класс java.awt.image, который представляет в Java пакет java.awt.image.

Класс JavaPackage имеет некоторые недостатки. Не существует способа сказать заранее, является ли свойство объекта JavaPackage ссылкой на Java-класс или другой Java-пакет, вследствие чего интерпретатор JavaScript исходит из предположения, что это класс, и пытается загрузить его. Таким образом, когда используется выражение, такое как java.awt, LiveConnect сначала ищет класс с таким именем. Если класс не найден, LiveConnect предполагает, что свойство ссылается на пакет, но при этом нет никакой возможности проверить наличие пакета и узнать, существуют ли реальные классы в этом пакете. Это порождает еще один серьезный недостаток: если программист допускает опечатку в имени класса, LiveConnect благополучно воспримет опечатку как имя пакета, вместо того чтобы сообщить, что класс с таким именем не существует.

12.2.2. Класс JavaClass

Класс JavaClass — это тип данных языка JavaScript, представляющий Java-класс. Объект класса JavaClass не имеет своих свойств: все его свойства являются представлениями одноименных свойств общедоступных статических полей и методов Java-класса. Эти статические поля и методы иногда называются *полями класса* и *методами класса*, чтобы обозначить, что они принадлежат всему классу, а не отдельному экземпляру класса. В отличие от JavaPackage, класс JavaClass допускает возможность перечисления своих свойств в цикле for/in. Примечательно, что объекты класса JavaClass не имеют свойств, представляющих поля и методы экземпляра Java-класса, — отдельные экземпляры Java-классов представляет класс JavaObject, который описывается в следующем разделе.

Как отмечалось ранее, объекты класса JavaClass содержатся в объектах класса JavaPackage. Например, объект java.lang класса JavaPackage содержит свойство System. Таким образом, java.lang.System — это объект класса JavaClass, представляющий Java-класс java.lang.System. Этот объект JavaClass в свою очередь имеет такие свойства, как out и in, представляющие статические поля класса java.lang.System. Точно таким же способом можно обращаться из JavaScript-сценария к любому стандартному системному Java-классу. Например, класс java.lang.Double имеет имя java.lang.Double (или Packages.java.lang.Double), а класс javax.swing.JButton — имя Packages.javax.swing.JButton.

Еще один способ получить в JavaScript объект класса JavaClass заключается в использовании функции getClass(). Передавая функции getClass() любой объект класса JavaObject, можно получить объект JavaClass, который будет являться представлением Java-класса этого объекта.[1]

---

[1] Не следует путать функцию getClass(), возвращающую объект JavaClass, с Java-методом getClass(), который возвращает объект java.lang.Class.

Как только экземпляр класса JavaClass получен, с ним можно выполнять некоторые действия. Класс JavaClass реализует функциональность LiveConnect, которая позволяет JavaScript-программам получать и записывать значения общедоступных статических полей Java-классов и вызывать общедоступные статические методы Java-классов. Например, java.lang.System – это экземпляр класса JavaClass, а получать и записывать значения статических полей java.lang.System можно следующим образом:

```
var java_console = java.lang.System.out;
```

Аналогичным образом производится вызов статических методов java.lang.System:

```
var java_version = java.lang.System.getProperty("java.version");
```

Ранее уже говорилось, что Java является строго типизированным языком: все поля, методы и аргументы обладают своими типами. Если попытаться записать значение в поле или передать аргумент неверного типа, генерируется исключение.

Класс JavaClass имеет одну очень важную особенность. Допускается использовать объекты класса JavaClass в операторе new для создания новых экземпляров Java-классов, т. е. для создания объектов JavaObject. Синтаксически в JavaScript (так же как и в Java) эта операция ничем не отличается от создания обычного JavaScript-объекта:

```
var d = new java.lang.Double(1.23);
```

Теперь, когда мы создали объект JavaObject таким способом, можно вернуться к функции getClass() и продемонстрировать, как она используется:

```
var d = new java.lang.Double(1.23);   // Создать JavaObject
var d_class = getClass(d);            // Получить JavaClass для JavaObject
if (d_class == java.lang.Double) ...; // Это сравнение даст в результате true
```

Чтобы не обращаться к объекту класса JavaClass с помощью громоздкого выражения, такого как java.lang.Double, можно определить переменную, которая послужит сокращенным псевдонимом:

```
var Double = java.lang.Double;
```

Такой прием может служить аналогом применения инструкции import в языке Java и повысить эффективность программ, т. к. в этом случае LiveConnect не придется искать ни свойство lang объекта java, ни свойство Double объекта java.lang.

12.2.3. Импорт пакетов и классов

В реализации LiveConnect интерпретатора Rhino определены глобальные функции, выполняющие импорт Java-пакетов и Java-классов. Для импорта пакета необходимо передать объект JavaPackage функции importPackage(), а для импорта класса – объект JavaClass функции importClass():

```
importPackage(java.util);
importClass(java.awt.List);
```

Функция importClass() копирует единственный объект JavaClass из объекта JavaPackage в глобальный объект. Предыдущий вызов функции importClass() эквивалентен следующей строке:

```
var List = java.awt.List;
```

На самом деле функция importPackage() не копирует все объекты JavaClass из JavaPackage в глобальный объект. Вместо этого она (с аналогичным эффектом) просто добавляет пакет во внутренний список пакетов, применяемый для разрешения неизвестных идентификаторов, и копирует только те объекты JavaClass, которые фактически используются. Таким образом, после представленного вызова функции importPackage() появляется возможность задействовать в JavaScript идентификатор Map. Если не объявлялось переменной с именем Map, этот идентификатор распознается как объект java.util.Map класса JavaClass и записывается во вновь созданное свойство Map глобального объекта.

Следует отметить, что импортировать пакет java.lang с помощью функции importPackage() нежелательно, поскольку пакет java.lang определяет множество функций, чьи имена совпадают с именами встроенных конструкторов и функций преобразования в JavaScript. Вместо импорта пакетов можно просто скопировать объект JavaPackage в более удобное место:

```
var swing = Packages.javax.swing;
```

Функции importPackage() и importClass() отсутствуют в версии SpiderMonkey, но смоделировать импорт одного класса достаточно просто и к тому же это гораздо безопаснее, поскольку это не приводит к загромождению глобального пространства имен импортируемыми пакетами.

12.2.4. Класс JavaObject

Класс JavaObject – это тип JavaScript-данных, представляющий Java-объект. Класс JavaObject во многом похож на класс JavaClass. Как и JavaClass, объект JavaObject не имеет собственных свойств – все его свойства являются представлениями (с теми же именами) общедоступных полей экземпляра и общедоступных методов экземпляра Java-объекта, который он представляет. Как и в случае с JavaClass, имеется возможность перечислить все свойства объекта JavaObject с помощью цикла for/in. Класс JavaObject реализует функциональность LiveConnect, которая позволяет получать и записывать значения общедоступных полей экземпляра и вызывать общедоступные методы Java-объекта.

Например, если предположить, что d – это объект JavaObject, представляющий экземпляр класса java.lang.Double, тогда вызвать метод Java-объекта из JavaScript-сценария можно следующим образом:

```
n = d.doubleValue();
```

Как было показано ранее, класс java.lang.System имеет статическое поле out. Это поле ссылается на Java-объект класса java.io.PrintStream. В JavaScript ссылка на соответствующий объект JavaObject выглядит следующим образом:

```
java.lang.System.out
```

Вызов метода этого объекта выполняется так:

```
java.lang.System.out.println("Hello world!");
```

Кроме того, объект JavaObject позволяет получать и записывать значения общедоступных полей экземпляра Java-объекта, который он представляет. Хотя ни класс java.lang.Double, ни класс java.io.PrintStream из предыдущих примеров не имеют общедоступных полей экземпляра, предположим, что JavaScript-сценарий создает экземпляр класса java.awt.Rectangle:

```
r = new java.awt.Rectangle( );
```

Тогда обратиться к общедоступным полям экземпляра из JavaScript-сценария можно следующим образом:

```
r.x = r.y = 0;
r.width = 4;
r.height = 5;
var perimeter = 2*r.width + 2*r.height;
```

Вся прелесть LiveConnect состоит в том, что благодаря этой технологии появляется возможность использовать Java-объекты так, как если бы они были обычными JavaScript-объектами. Однако здесь следует сделать несколько замечаний: `r` – это экземпляр класса `JavaObject`, и он ведет себя не совсем так, как обычные JavaScript-объекты (подробнее о различиях рассказывается далее). Кроме того, не следует забывать, что в отличие от JavaScript, поля Java-объектов и аргументы Java-методов являются типизированными. Если передать им JavaScript-значение неверного типа[1], интерпретатор JavaScript сгенерирует исключение.

12.2.5. Методы Java

Поскольку LiveConnect организует доступ к Java-объектам как к JavaScript-свойствам, Java-методы можно рассматривать как значения, точно так же, как и JavaScript-функции. Тем не менее следует отметить, что методы экземпляров на самом деле являются методами, а не функциями, и потому должны вызываться посредством Java-объектов. Однако статические Java-методы могут рассматриваться как JavaScript-функции, и для удобства их можно импортировать в глобальное пространство имен:

```
var isDigit = java.lang.Character.isDigit;
```

12.2.5.1. Методы доступа к свойствам

Если Java-объект в реализации LiveConnect интерпретатора Rhino обладает методами экземпляра, которые в соответствии с соглашениями JavaBeans об именовании выглядят как методы доступа к свойствам (методы чтения/записи), LiveConnect делает возможным прямой доступ к этим свойствам, как к обычным JavaScript-свойствам. Например, рассмотрим объекты `javax.swing.JFrame` и `javax.swing.JButton`, которые уже упоминались ранее. Объект `JButton` имеет методы `setFont()` и `getFont()`, а объект `JFrame` – методы `setVisible()` и `getVisible()`. LiveConnect делает доступными эти методы, но, кроме того, в объекте `JButton` создается свойство `font`, а в объекте `JFrame` – свойство `visible`. Рассмотрим пример:

```
button.setFont(font);
frame.setVisible(true);
```

Благодаря указанным свойствам появляется возможность заменить эти строки следующими:

---

[1] То есть с точки зрения интерпретатора JavaScript эти значения остаются корректными до момента их присваивания полям объекта `JavaObject`, но оказываются некорректными в отношении синтаксиса Java, в результате чего генерируется исключение. – *Примеч. науч. ред.*

```
button.font = font;
frame.visible = true;
```

12.2.5.2. Перегруженные методы

Java-классы могут определять несколько методов с одинаковыми именами. Если попытаться перечислить свойства объекта JavaObject, который имеет перегруженный метод экземпляра, удастся увидеть лишь одно свойство с именем перегруженного метода. Обычно реализация LiveConnect будет пытаться вызвать корректный метод, основываясь на типах передаваемых аргументов.

Однако иногда может потребоваться явно указать, какой из перегруженных методов должен быть вызван. Доступ к перегруженным методам в объектах JavaObject и JavaClass выполняется через специальные свойства, которые включают в себя как имя перегруженного метода, так и типы его аргументов. Предположим, что имеется объект o класса JavaObject, в котором есть два метода с именем f, один из которых принимает аргумент типа int, а другой – типа boolean. Тогда свойство o.f будет представлять функцию, вызывающую наиболее подходящий Java-метод, основываясь на типе входного аргумента. В то же время есть возможность явно указать, какой из двух Java-методов следует вызвать:

```
var f = o['f'];                 // Вызов наиболее подходящего метода
var boolfunc = o['f(boolean)']; // Метод с аргументом типа boolean
var intfunc = o['f(int)'];      // Метод с аргументом типа int
```

Когда круглые скобки используются как часть имени свойства, обычная точечная нотация для обращения к нему не подходит – это должно быть строковое выражение в квадратных скобках.

Примечательно, что тип JavaClass может также различать перегруженные статические методы.

12.2.6. Класс JavaArray

Последний тип данных LiveConnect в JavaScript – класс JavaArray. Как следует из названия, экземпляры этого класса являются представлением массивов в языке Java и реализуют функциональность LiveConnect, которая позволяет обращаться к элементам Java-массивов из JavaScript-сценария. Подобно JavaScript- и Java-массивам, объект JavaArray имеет свойство length, которое определяет количество элементов, содержащихся в массиве. Для обращения к элементам объекта JavaArray можно использовать оператор индексирования массивов []. Кроме того, элементы массива могут быть перечислены с помощью цикла for/in. Объекты JavaArray могут применяться для доступа к многомерным массивам (фактически массивам массивов) так же, как это делается в JavaScript или в Java.

В качестве примера попробуем создать экземпляр класса java.awt.Polygon:

```
p = new java.awt.Polygon( );
```

Объект p класса JavaObject имеет свойства xpoints и ypoints, которые являются объектами класса JavaArray, представляющими массивы целых чисел. (Чтобы узнать имена и типы этих свойств, следует заглянуть в описание класса java.awt.Polygon в справочном руководстве по языку Java.) Эти свойства можно использовать для инициализации координат вершин многоугольника в случайном порядке:

```
for(var i = 0; i < p.xpoints.length; i++)
    p.xpoints[i] = Math.round(Math.random( )*100);
for(var i = 0; i < p.ypoints.length; i++)
    p.ypoints[i] = Math.round(Math.random( )*100);
```

12.2.6.1. Создание Java-массивов

Реализация LiveConnect не предусматривает возможность создания Java-массивов или преобразования JavaScript-массивов в Java-массивы. Если возникает необходимость создать Java-массив, делать это необходимо явно с помощью пакета `java.lang.reflect`:

```
var p = new java.awt.Polygon();
p.xpoints = java.lang.reflect.Array.newInstance(java.lang.Integer.TYPE,5);
p.ypoints = java.lang.reflect.Array.newInstance(java.lang.Integer.TYPE,5);
for(var i = 0; i < p.xpoints.length; i++) {
    p.xpoints[i] = i;
    p.ypoints[i] = i * i;
}
```

12.2.7. Реализация интерфейсов с помощью LiveConnect

Версия LiveConnect в интерпретаторе Rhino позволяет JavaScript-сценариям реализовывать Java-интерфейсы с использованием несложного синтаксиса: интерфейсы `JavaClass` следует трактовать просто как конструкторы и передавать в JavaScript-объекты, которые имеют свойства для каждого из методов интерфейса. Эту возможность можно задействовать, например, для добавления обработчиков событий в программный код, создающий графический пользовательский интерфейс, как это уже было показано ранее:

```
// Импортировать все, что требуется.
importClass(Packages.javax.swing.JFrame);
importClass(Packages.javax.swing.JButton);
importClass(java.awt.event.ActionListener);

// Создать Java-объекты.
var frame = new JFrame("Hello World");
var button = new JButton("Hello World");

// Реализовать интерфейс ActionListener.
var listener = new ActionListener({
        actionPerformed: function(e) { print("Hello!"); }
    });

// Добавить обработчик события к кнопке.
button.addActionListener(listener);

// Вставить кнопку в ее фрейм и вывести на экран.
frame.add(button);
frame.setSize(200, 200);
frame.setVisible(true);
```

12.2.8. Преобразование данных в LiveConnect

Java – это строго типизированный язык, обладающий относительно большим числом типов данных. В то же время JavaScript является нетипизированным

языком, обладающим сравнительно небольшим числом типов данных. Поскольку между этими языками имеется такое существенное структурное различие, одна из основных обязанностей LiveConnect заключается в выполнении корректного преобразования типов данных. Когда JavaScript-сценарий записывает значение в поле Java-объекта или передает аргумент Java-методу, JavaScript-значение должно быть преобразовано в эквивалентное Java-значение. Когда JavaScript-сценарий читает значение поля Java-объекта или получает возвращаемое значение Java-метода, Java-значение должно быть преобразовано в совместимый тип данных языка JavaScript. К сожалению, преобразование данных в LiveConnect реализовано несколько иначе, нежели в пакете `javax.script`.

Рисунки 12.1 и 12.2 иллюстрируют порядок преобразования данных при записи значений из JavaScript-сценария в Java-программу, и наоборот.

Обратите внимание на следующие замечания, касающиеся порядка преобразования данных на рис. 12.1:

- На рисунке показаны не все возможные варианты преобразования типов JavaScript-данных в типы Java-данных, потому что перед преобразованием из JavaScript в Java может произойти внутреннее преобразование JavaScript-данных. Например, если JavaScript-сценарий передает число Java-методу, который ожидает получить аргумент типа `java.lang.String`, интерпретатор JavaScript сначала преобразует число в строку, а затем преобразует ее в Java-строку.
- JavaScript-число может быть преобразовано в любой из элементарных числовых типов языка Java. Какое из преобразований будет выбрано, зависит от типа целевого Java-поля или аргумента Java-метода. Обратите внимание: в ходе преобразования может быть утеряна точность числа, например, когда слишком большое число записывается в Java-поле типа `short` или производится преобразование вещественного значения в целочисленный Java-тип.

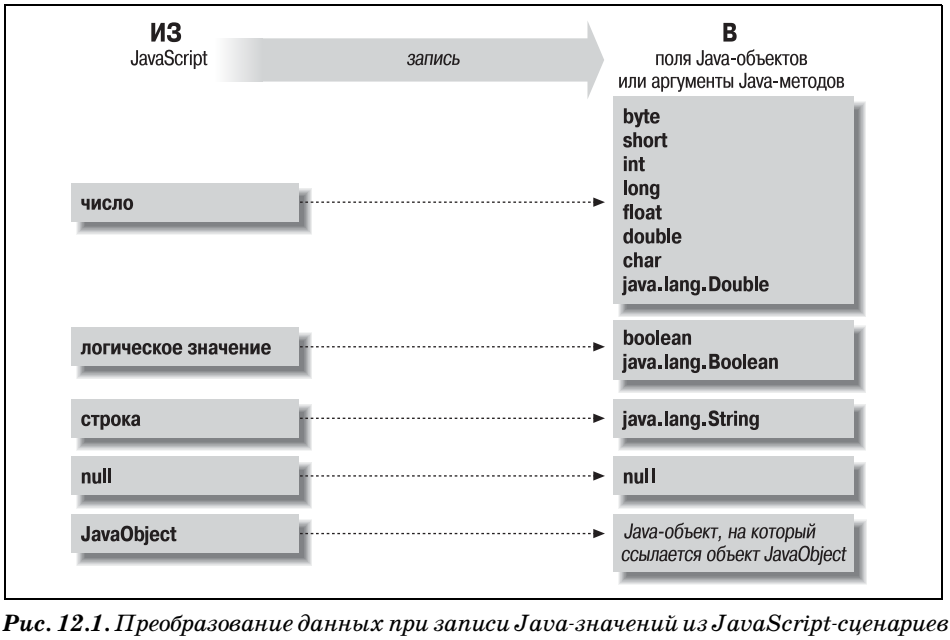

Рис. 12.1. *Преобразование данных при записи Java-значений из JavaScript-сценариев*

- JavaScript-число может быть также преобразовано в экземпляр Java-класса `java.lang.Double`, но никогда – в экземпляры родственных классов, таких как `java.lang.Integer` и `java.lang.Float`.
- В JavaScript отсутствует тип данных для представления символов, поэтому числа JavaScript могут быть преобразованы в элементарный Java-тип `char`.
- Когда из JavaScript в Java передается объект `JavaObject`, он «разворачивается»[1], благодаря чему преобразуется в тот Java-объект, который он представляет. Однако объекты класса `JavaClass` в JavaScript не преобразуются в экземпляры класса `java.lang.Class`, как того можно было бы ожидать.
- JavaScript-массивы никак не преобразуются в Java-массивы. Объекты, массивы и функции языка JavaScript преобразуются в Java-объекты, не имеющие стандартизованного прикладного интерфейса и обычно рассматриваемые как «черные ящики».

К преобразованиям данных, которые приводятся на рис. 12.2, также имеется несколько пояснений:

- Поскольку JavaScript не имеет типа для представления символьных данных, элементарный Java-тип `char` преобразуется в числовой JavaScript-тип, а не в строку, как можно было бы ожидать.

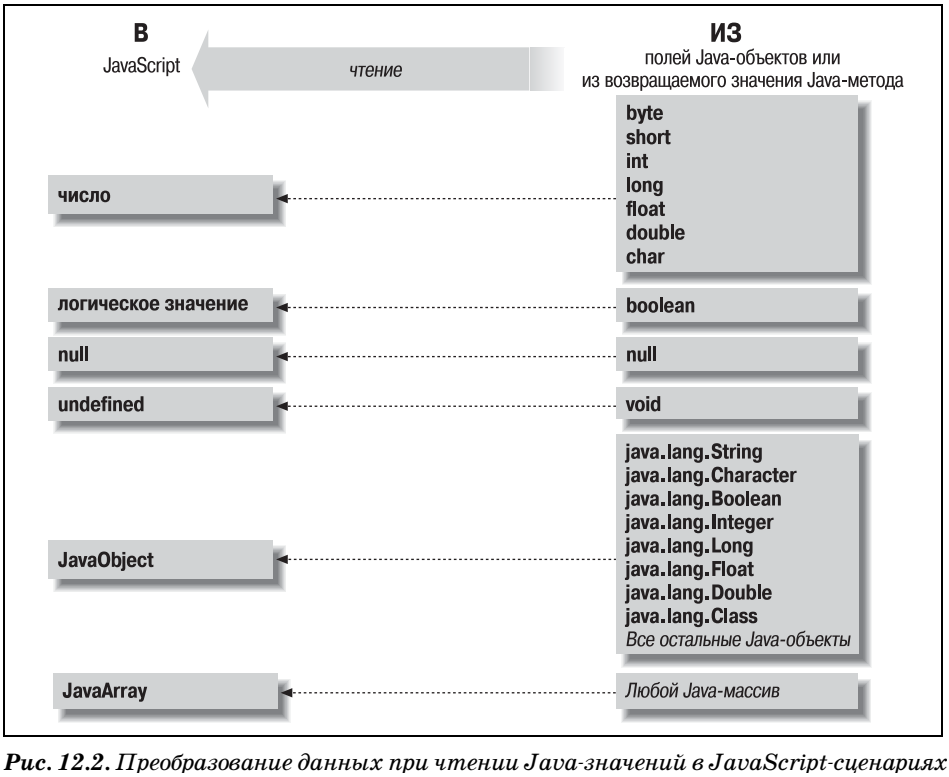

***Рис. 12.2.** Преобразование данных при чтении Java-значений в JavaScript-сценариях*

[1] Имеется в виду, что с него «снимается» обертка `JavaObject` и «остается» ссылка на Java-объект. – *Примеч. науч. ред.*

- Экземпляры java.lang.Double, java.lang.Integer и аналогичных им классов не преобразуются в JavaScript-числа. Подобно любым другим Java-объектам, они преобразуются в объекты JavaObject.
- Строки в языке Java являются экземплярами класса java.lang.String, поэтому, подобно любым другим Java-объектам, они преобразуются в объекты JavaObject, а не в JavaScript-строки.
- Java-массивы любого типа в JavaScript преобразуются в объекты JavaArray.

12.2.8.1. Преобразование JavaObject в JavaScript

Обратите внимание: на рис. 12.2 показано, что достаточно большое число типов Java-данных, включая строки (экземпляры класса java.lang.String), в JavaScript-сценариях преобразуются в объекты JavaObject, а не в значения элементарных типов данных, такие как строки. Это означает, что при использовании LiveConnect часто приходится работать с объектами JavaObject. Объекты JavaObject отличаются своим поведением от других JavaScript-объектов, поэтому вы должны знать о некоторых наиболее распространенных ловушках.

Первая странность в том, что чаще всего приходится работать с объектами JavaObject, которые являются представлениями экземпляров java.lang.Double или других числовых типов. В большинстве случаев такого рода объект JavaObject ведет себя подобно значению элементарного числового типа, но при использовании оператора сложения (+) следует быть начеку. Когда в операции сложения участвует объект JavaObject (или любой другой JavaScript-объект), определяется строковый контекст операции, вследствие чего объект преобразуется в строку и вместо операции сложения числовых значений выполняется операция конкатенации строк. Чтобы выполнить явное преобразование, нужно объект JavaObject передать функции преобразования Number().

Чтобы преобразовать объект JavaObject в строковое JavaScript-значение, следует использовать функцию преобразования String(), а не вызывать метод toString(). Все Java-классы имеют унаследованный метод toString(), поэтому вызов метода toString() объекта JavaObject ведет к вызову Java-метода, а тот в свою очередь вернет другой объект JavaObject, в который будет обернут экземпляр java.lang.String, как показано в следующем фрагменте:

```
var d = new java.lang.Double(1.234);
var s = d.toString(); // Преобразует в java.lang.String, а не в строку
print(typeof s);      // Выведет "object", поскольку s - это JavaObject
s = String(d);        // Теперь получится строка JavaScript
print(typeof s);      // Выведет "string".
```

Обратите внимание: JavaScript-строки имеют числовое свойство length. В то же время объект JavaObject, в который обернут экземпляр java.lang.String, также имеет свойство length, являющееся представлением метода length() объекта-строки языка Java.

Другой странный случай – объект JavaObject класса java.lang.Boolean.FALSE. При использовании в строковом контексте значение этого объекта преобразуется в false, а при использовании в логическом контексте – в true! Это происходит потому, что JavaObject не является значением null. Значение, хранящееся в этом объекте, просто не предназначено для такого рода преобразований.

II

Клиентский JavaScript

В данной части книги в главах с 13 по 23 язык JavaScript описан в том виде, в котором он реализован в веб-броузерах. В этих главах вводится много новых JavaScript-объектов, представляющих веб-броузер, а также содержимое HTML- и XML-документов.

- Глава 13 «JavaScript в веб-броузерах»
- Глава 14 «Работа с окнами броузера»
- Глава 15 «Работа с документами»
- Глава 16 «CSS и DHTML»
- Глава 17 «События и обработка событий»
- Глава 18 «Формы и элементы форм»
- Глава 19 «Cookies и механизм сохранения данных на стороне клиента»
- Глава 20 «Работа с протоколом HTTP»
- Глава 21 «JavaScript и XML»
- Глава 22 «Работа с графикой на стороне клиента»
- Глава 23 «Сценарии с Java-апплетами и Flash-роликами»

13

JavaScript в веб-броузерах

Первая часть этой книги была посвящена базовому языку JavaScript. Теперь мы перейдем к тому языку JavaScript, который используется в веб-броузерах и обычно называется клиентским JavaScript (client-side JavaScript).[1] Большинство примеров, которые мы видели до сих пор, будучи корректным JavaScript-кодом, не имели определенного контекста; это были JavaScript-фрагменты, не предназначенные для запуска в какой-либо определенной среде. Эта глава предоставляет такой контекст. Она начинается с абстрактного введения в среду программирования веб-броузера и базовые концепции клиентского языка JavaScript. Затем в ней рассказывается о том, каким образом JavaScript-код фактически встраивается в HTML-документы и как в JavaScript используются тег `<script>`, HTML-атрибуты обработчиков событий и URL-адреса. Вслед за разделом, описывающим встраивание JavaScript-сценариев, следует раздел с описанием модели исполнения, объясняющий, как и когда запускаются JavaScript-программы в веб-броузере. Далее следуют разделы с обсуждением трех важных тем программирования на JavaScript: совместимость, удобство и безопасность. Завершает главу короткое описание некоторых других реализаций JavaScript, имеющих отношение к Всемирной паутине, но не относящихся к клиентскому языку JavaScript.

При встраивании JavaScript в веб-броузер последний получает мощный и многообразный набор характеристик, которыми можно управлять из сценариев. Каждая из следующих глав фокусируется на одной из основных функциональных областей клиентского языка JavaScript.

- Глава 14 «Работа с окнами броузера» описывает, как JavaScript может управлять окнами веб-броузера, например открывать и закрывать окна броузера,

---

[1] Термин «client-side JavaScript» остался с тех времен, когда язык JavaScript применялся только в веб-броузерах (клиентах) и веб-серверах. Поскольку JavaScript в качестве языка сценариев распространяется во все большем количестве сред, слова «client-side» имеют все меньше и меньше смысла ввиду частого отсутствия клиентской стороны. Тем не менее в этой книге мы будем по-прежнему употреблять этот термин.

выводить диалоговые окна, переходить по заданному URL-адресу или по списку посещенных ранее страниц вперед и назад. В этой главе описываются также некоторые другие особенности клиентского языка JavaScript, которые имеют отношение к объекту `Window`.

- Глава 15 «Работа с документами» описывает, как из JavaScript управлять содержимым документа, отображаемого в окне броузера, и как искать, вставлять, удалять или изменять части документа.
- Глава 16 «CSS и DHTML» рассказывает о порядке взаимодействия между JavaScript-кодом и CSS-таблицами, а также показывает, как JavaScript-сценарий может изменять представление документа, изменяя CSS-стили, классы и таблицы стилей. Особенно интересный результат получается при объединении возможностей CSS-таблиц и динамического языка HTML (или DHTML), при использовании которого HTML-содержимое может быть скрыто, отображено, перемещено и даже анимировано.
- Глава 17 «События и обработка событий» описывает события и порядок их обработки, а также показывает, как с помощью JavaScript сделать веб-страницу интерактивной, способной откликаться на действия пользователя.
- Глава 18 «Формы и элементы форм» посвящена работе с HTML-формами. В ней показано, как с помощью JavaScript организовать сбор, проверку, обработку и передачу данных, полученных от пользователя.
- Глава 19 «Cookies и механизм сохранения данных на стороне клиента» демонстрирует, как организовать хранение данных на стороне клиента с помощью cookies.
- Глава 20 «Работа с протоколом HTTP» описывает приемы работы с протоколом HTTP (технология, известная под названием Ajax) и демонстрирует, как можно организовать взаимодействие JavaScript-сценариев с сервером.
- Глава 21 «JavaScript и XML» показывает, как создавать, загружать, анализировать, преобразовывать и сериализовывать XML-документы, а также как извлекать из них данные.
- Глава 22 «Работа с графикой на стороне клиента» демонстрирует широко распространенные приемы работы с графикой, позволяющие создавать на веб-страницах интерактивные изображения и анимацию. В ней также показаны некоторые приемы динамического создания векторных графических изображений с помощью JavaScript-сценариев.
- Глава 23 «Сценарии с Java-апплетами и Flash-роликами» рассказывает, как организовать взаимодействие JavaScript-кода с Java-апплетами и Flash-роликами, внедренными в веб-страницу.

13.1. Среда веб-броузера

Чтобы понять, что такое клиентский язык JavaScript, необходимо разобраться с концептуальной основой среды программирования, предоставляемой веб-броузером. Следующие разделы представляют собой введение в три основные составляющие этой среды программирования:

- объект `Window`, который представляет собой глобальный объект и глобальный контекст исполнения для клиентского JavaScript-кода;

- иерархия объектов на стороне клиента и объектная модель документа (DOM), формирующие ее часть;
- управляемая событиями модель программирования.

Эти разделы сопровождаются обсуждением роли JavaScript в разработке веб-приложений.

13.1.1. Окно как глобальный контекст исполнения

Основная задача веб-броузера состоит в отображении HTML-документа в окне. В клиентском языке JavaScript объект `Document` представляет HTML-документ, а объект `Window` – окно (или отдельный фрейм), в котором отображается этот документ. Хотя в клиентском JavaScript оба этих объекта важны, объект `Window` более важен по одной существенной причине – это глобальный объект при программировании на стороне клиента.

Вспомните из главы 4, что в любой реализации JavaScript на вершине цепочки областей видимости всегда расположен глобальный объект; свойства глобального объекта являются глобальными переменными. В клиентском JavaScript объект `Window` – это глобальный объект. Объект `Window` определяет несколько свойств и методов, позволяющих манипулировать окном веб-броузера. Он также определяет свойства, ссылающиеся на другие важные объекты, такие как свойство `document` объекта `Document`. И наконец, объект `Window` имеет два свойства для ссылки на себя – `window` и `self`. Любая из этих глобальных переменных может использоваться для ссылки непосредственно на объект `Window`.

Поскольку объект `Window` – это глобальный объект клиентского JavaScript, все глобальные переменные определяются как свойства окна. Например, следующие две строки выполняют по существу одно и то же действие:

```
var answer = 42;     // Объявляем и инициализируем глобальную переменную
window.answer = 42; // Создаем новое свойство объекта Window
```

Объект `Window` представляет окно веб-броузера (или фрейм внутри окна; для клиентского JavaScript окна верхнего уровня и фреймы по существу эквивалентны). Существует возможность написать приложение, работающее с несколькими окнами (или фреймами). Каждое окно приложения имеет уникальный объект `Window` и определяет уникальный контекст исполнения для кода клиентского JavaScript. Другими словами, глобальная переменная, объявленная JavaScript-кодом в одном окне, не является глобальной в другом окне. Однако JavaScript-код второго окна *может* обращаться к глобальной переменной первого фрейма, хотя эта возможность нередко ограничивается из соображений безопасности. Эти проблемы подробно рассматриваются в главе 14.

13.1.2. Иерархия объектов клиентского JavaScript и объектная модель документа

Мы видели, что объект `Window` – это ключевой объект в клиентском JavaScript. Через него доступны все остальные объекты. Например, любой объект `Window` содержит свойство `document`, ссылающееся на связанный с окном объект `Document`, и свойство `location`, ссылающееся на связанный с окном объект `Location`. Объект `Window` также содержит массив `frames[]`, ссылающийся на объекты `Window`, пред-

ставляющие фреймы исходного окна. То есть `document` представляет объект `Document` текущего окна, а `frames[1].document` ссылается на объект `Document` второго дочернего фрейма текущего окна.

Объект `Document` (и другие объекты клиентского JavaScript) имеют также свойства, которые позволяют ссылаться на другие объекты. Например, в каждом объекте `Document` имеется массив `forms[]`, содержащий объекты `Form`, которые представляют любые присутствующие в документе HTML-формы. Для ссылки на одну из этих форм можно использовать выражение:

```
window.document.forms[0]
```

Продолжим тот же пример: в каждом объекте `Form` имеется массив `elements[]`, содержащий объекты, которые представляют различные элементы HTML-форм (поля ввода, кнопки и т. д.), присутствующие внутри формы. В некоторых случаях программисту приходится писать код, ссылающийся на объект в конце всей цепочки объектов, получая, например, такие сложные выражения:

```
parent.frames[0].document.forms[0].elements[3].options[2].text
```

Как мы видели ранее, объект `Window` – это глобальный объект в начале цепочки областей видимости, и все клиентские объекты в JavaScript доступны как свойства других объектов. Это значит, что имеется иерархия JavaScript-объектов, в корне которой находится объект `Window`. Эта иерархия показана на рис. 13.1.

Обратите внимание: на рис. 13.1 показаны только свойства объектов, ссылающиеся на другие объекты. Большинство объектов, изображенных на диаграмме, имеют немало свойств, которые здесь не показаны.

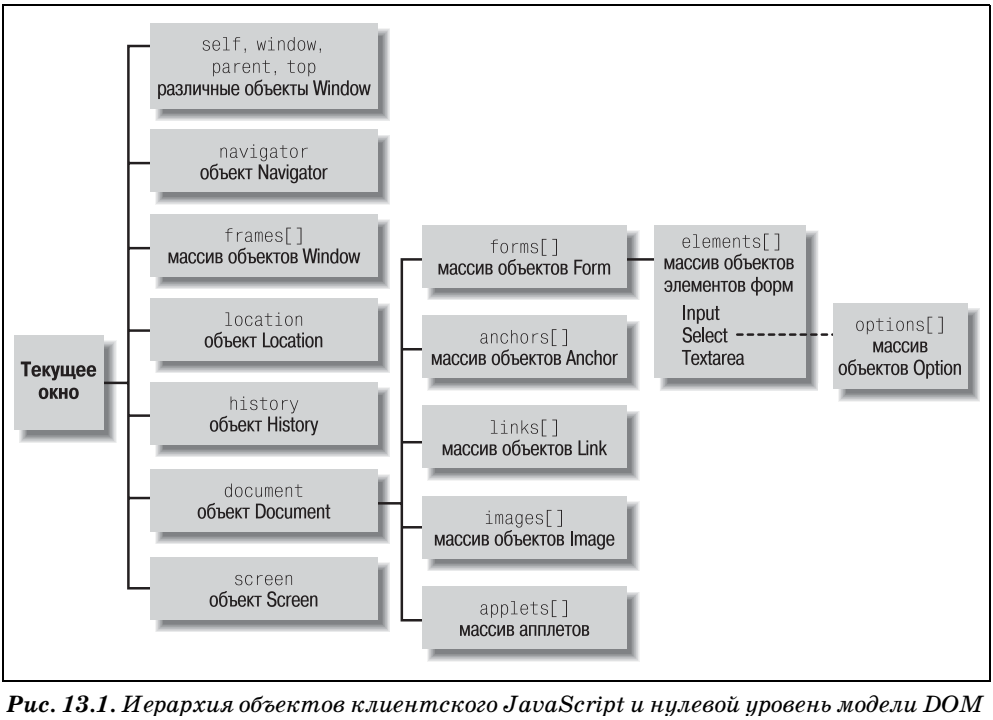

***Рис. 13.1.** Иерархия объектов клиентского JavaScript и нулевой уровень модели DOM*

Многие объекты, изображенные на этом рисунке, происходят от объекта `Document`. Это поддерево большой иерархии объектов на стороне клиента известно как объектная модель документа (Document Object Model, DOM) и интересно тем, что на нем сконцентрировались усилия по стандартизации. На рисунке показаны объекты документа, которые стали стандартом «де-факто», т. к. согласованно реализованы во всех основных броузерах. Все вместе они известны как нулевой уровень модели DOM (DOM Level 0), поскольку образуют базовый уровень функциональности документа, на которую JavaScript-программисты могут опираться во всех броузерах. Эти основные объекты документа рассмотрены в главе 15, где также описывается усложненная объектная модель документа, стандартизованная W3C. HTML-формы являются частью DOM, но это настолько специализированная тема, что ее обсуждение выделено в отдельную главу 18.

13.1.3. Управляемая событиями модель программирования

В прошлом компьютерные программы часто работали в пакетном режиме – читали пакет данных, выполняли какие-то вычисления, а затем выводили результат. Позднее вместе с режимом разделения времени и текстовыми терминалами стали возможны ограниченные виды интерактивности – программа могла запросить от пользователя данные, а тот мог их ввести. Затем компьютер обрабатывал данные и выводил на экран результат.

С появлением графических дисплеев и указывающих устройств, таких как мыши, ситуация изменилась. Программы в основном стали управляться событиями, реагируя на асинхронный пользовательский ввод в виде щелчков мыши и нажатий клавиш, способ интерпретации которых зависит от положения указателя мыши. Веб-броузер – как раз такая графическая среда. HTML-документ имеет графический пользовательский интерфейс (GUI), и клиентский JavaScript использует управляемую событиями модель программирования.

Можно написать статическую JavaScript-программу, не принимающую пользовательских данных и делающую всегда одно и то же. Иногда такие программы полезны. Однако чаще мы пишем динамические программы, взаимодействующие с пользователем. Чтобы это сделать, у нас должна быть возможность реагировать на его действия.

В клиентском JavaScript веб-броузер уведомляет программы о действиях пользователя, генерируя события. Имеются различные типы событий, такие как нажатие клавиш, перемещение мыши и т. д. Когда происходит событие, веб-броузер пытается вызвать соответствующую функцию-обработчик события для реакции на него. Поэтому для написания динамических, интерактивных клиентских JavaScript-программ мы должны определить нужные обработчики событий и зарегистрировать их в системе, чтобы броузер мог вызывать их в нужные моменты.

Тем, кто еще не знаком с управляемой событиями моделью программирования, придется потратить немного времени, чтобы к ней привыкнуть. В старой модели программист писал единый монолитный блок кода, выполнение которого осуществлялось в каком-либо определенном порядке от начала до конца. Управляемое событиями программирование переворачивает эту модель с ног на голову. В управляемом событиями программировании создается несколько независимых (но взаимодействующих между собой) обработчиков событий. Программист

не вызывает их непосредственно, а позволяет системе вызывать их в нужный момент. Так как обработчики запускаются от действий пользователя, они могут исполняться в непредсказуемые, или асинхронные, моменты времени. Большую часть времени программа вообще не работает, а просто ожидает, пока система вызовет один из ее обработчиков событий.

В следующем разделе объясняется, как встраивать JavaScript-код в HTML-файлы, как определять и статические блоки кода, работающие синхронно от начала до конца, и обработчики событий, вызываемые системой асинхронно. События и их обработку мы обсудим подробнее в главе 15, а затем более глубокое обсуждение событий продолжим в главе 17.

13.1.4. Роль JavaScript в Web

В начале этой главы коротко были перечислены характеристики веб-броузеров, доступные для управления из JavaScript-сценариев. Однако перечень характеристик, *доступных* в JavaScript, значительно отличается от перечня характеристик, которые *могли бы* использоваться в JavaScript. В этом разделе предпринята попытка объяснить роль JavaScript в разработке веб-приложений.

Веб-броузеры отображают структурированный текст HTML-документа с использованием каскадных таблиц стилей (Cascading Style Sheets, CSS). HTML определяет содержимое, а CSS – представление. При надлежащем использовании JavaScript добавляет к содержимому и представлению *поведение*. Роль JavaScript заключается в расширении возможностей пользователя, облегчая для него получение и передачу информации. Возможности пользователя не должны зависеть от JavaScript, но JavaScript может расширить эти возможности. Сделать это можно разными способами. Вот несколько примеров:

- Создание визуальных эффектов, таких как анимация графических изображений, ненавязчиво помогающих пользователю ориентироваться при просмотре страницы.
- Сортировка столбцов таблицы, упрощающая поиск нужной пользователю информации.
- Сокрытие части содержимого и раскрытие элементов с подробными сведениями по выбору пользователя.
- Упрощение просмотра за счет прямого взаимодействия с веб-сервером, что позволяет обновлять информацию без необходимости полной перезагрузки всей страницы.

13.1.5. Ненавязчивый JavaScript–код

Новая парадигма программирования клиентских сценариев, известная как *ненавязчивый JavaScript-код* (*unobtrusive JavaScript*), получила широкое распространение в сообществе разработчиков веб-приложений. Как следует из названия, данная парадигма утверждает, что JavaScript-код не должен привлекать к себе внимание, т. е. он не должен «навязываться».[1] JavaScript-код не должен

[1] Слово «навязывать» в некотором смысле можно считать синонимом слова «злоупотреблять». Цитата из толкового словаря «The American Heritage dictionary»: «Навязываться ... другим с неуместной настойчивостью или без приглашения».

навязываться пользователям, просматривающим веб-страницы, авторам, создающим HTML-разметку, или веб-дизайнерам, разрабатывающим HTML-шаблоны или CSS-таблицы.

Нет строгих правил, которых следовало бы придерживаться при создании ненавязчивого JavaScript-кода. Однако ряд полезных приемов, обсуждаемых в разных местах этой книги, укажут вам верное направление.

Основная цель парадигмы ненавязчивого JavaScript-кода – отделить программный код от HTML-разметки. По сути, хранить содержимое отдельно от поведения – все равно, что хранить CSS-таблицы во внешних файлах, т. е. отделять содержимое от представления. Для достижения этой цели весь JavaScript-код должен быть вынесен в отдельные файлы, которые следует подключать к HTML-страницам с помощью тега `<script src=>` (подробнее см. раздел 13.2.2). Если подходить еще более строго к разделению содержимого и поведения, можно даже не включать JavaScript-код в атрибуты обработчиков событий в HTML-файле. Вместо этого можно написать JavaScript-код в виде отдельного файла, который будет регистрировать обработчики событий в необходимых HTML-элементах (о том, как это делается, рассказывается в главе 17).

Как следствие этого, необходимо стремиться делать внешние файлы с JavaScript-кодом настолько модульными, насколько это возможно, используя методы, описанные в главе 10. Это даст возможность подключать массу независимых модулей к одной и той же веб-странице, не беспокоясь о том, что переменные и функции одного модуля перекроют переменные и функции другого.

Вторая цель ненавязчивого JavaScript-кода состоит в том, чтобы возможные ограничения функциональности не слишком сказывались на возможностях самой страницы. Сценарии должны задумываться и разрабатываться как расширения к HTML-содержимому, при этом само содержимое должно быть доступно для просмотра и без JavaScript-кода (например, когда пользователь отключает в броузере режим исполнения JavaScript-кода). Важное место здесь занимает методика, которая называется *проверкой возможностей* (*feature testing*): прежде чем предпринять какое-либо действие, JavaScript-модули должны убедиться, что функциональные особенности, требуемые для выполнения этого действия, доступны в броузере, на котором исполняется сценарий. Методика проверки возможностей более подробно описывается в разделе 13.6.3.

Третья цель ненавязчивого JavaScript-кода состоит в том, чтобы не сделать HTML-страницу менее доступной (в идеале она должна стать более доступной). Если включение JavaScript-кода делает обращение с веб-страницей более сложным, такой JavaScript-код будет мешать пользователям с ограниченными возможностями, для которых простота доступа имеет важное значение. Более подробно тема обеспечения доступности средствами JavaScript описывается в разделе 13.7.

Другие формулировки ненавязчивого JavaScript-кода могут включать иные цели в дополнение к трем перечисленным. Основным источником сведений о ненавязчивом JavaScript-кода является документ «The JavaScript Manifesto», опубликованный проблемной группой DOM Scripting Task Force по адресу *http://domscripting.webstandards.org/?page_id=2*.

13.2. Встраивание JavaScript-кода в HTML-документы

Клиентский JavaScript-код может встраиваться в HTML-документы несколькими способами:

- между парой тегов `<script>` и `</script>`;
- из внешнего файла, заданного атрибутом `src` тега `<script>`;
- в обработчик события, заданный в качестве значения HTML-атрибута, такого как `onclick` или `onmouseover`;
- как тело URL-адреса, использующего специальный спецификатор псевдопротокола `javascript:`.

В этом разделе описываются теги `<script>`. Порядок встраивания JavaScript-кода в обработчики событий и URL описан в этой главе позже.

13.2.1. Тег <script>

Клиентские JavaScript-сценарии представляют собой часть HTML-файла и находятся между тегами `<script>` и `</script>`:

```
<script>
// Здесь располагается JavaScript-код
</script>
```

В языке разметки XHTML содержимое тега `<script>` обрабатывается наравне с содержимым любого другого тега. Если JavaScript-код содержит символы `<` или `&`, они интерпретируются как элементы XML-разметки. Поэтому в случае применения языка XHTML лучше помещать весь JavaScript-код внутрь секции `CDATA`:

```
<script><![CDATA[// Здесь располагается JavaScript-код
]]></script>
```

Единственный HTML-документ может содержать произвольное число элементов `<script>`. При наличии нескольких отдельных сценариев, они будут запускаться в порядке их следования в документе (исключение составляет атрибут `defer`, описанный в разделе 13.2.4). Хотя отдельные сценарии в одном файле исполняются в различные моменты времени, в процессе загрузки и анализа HTML-файла они представляют собой части одной JavaScript-программы: функции и переменные, определенные в одном сценарии, доступны всем сценариям, находящимся в том же файле. Например, в HTML-странице может быть следующий сценарий:

```
<script>function square(x) { return x*x; }</script>
```

Ниже на той же HTML-странице вы можете вызывать функцию `square()`, причем даже в другом блоке сценария. Контекстом является HTML-страница, а не блок сценария:[1]

---

[1] Здесь функция `alert()` используется просто для отображения информации: она преобразует свой аргумент в строку и выводит ее в диалоговом окне. Подробнее метод `alert()` описывается в разделе 14.5. В примере 15.9 приводится альтернатива функции `alert()`, не создающая всплывающих диалоговых окон, для закрытия которых требуется выполнить щелчок.

```
<script>alert(square(2));</script>
```

В примере 13.1 показан HTML-файл, включающий простую JavaScript-программу. Обратите внимание на различие между этим примером и многими фрагментами кода, показанными в этой книге ранее: пример интегрирован в HTML-файл и имеется понятный контекст, в котором он работает. Обратите также внимание на атрибут `language` в теге `<script>`. Его описание приводится в разделе 13.2.3.

Пример 13.1. Простая JavaScript-программа в HTML-файле

```
<html>
<head>
<title>Сегодняшняя дата</title>
<script language="JavaScript">
// Определяем функцию для дальнейшего использования
function print_todays_date() {
    var d = new Date();                  // Получаем текущие дату и время
    document.write(d.toLocaleString()); // Вставляем это в документ
}
</script>
</head>
<body>
Дата и время:<br>
<script language="JavaScript">
    // Теперь вызываем определенную ранее функцию
    print_todays_date();
</script>
</body>
</html>
```

Пример 13.1 помимо всего прочего демонстрирует использование функции `document.write()`. В клиентском JavaScript эта функция может применяться для вывода HTML-текста в той точке документа, в которой располагается сценарий (более подробно об этом методе рассказывается в главе 15). Обратите внимание: способность сценариев генерировать текст для вставки в HTML-документ означает, что синтаксический анализатор HTML-кода должен интерпретировать JavaScript-сценарии как часть общего процесса синтаксического разбора документа. Невозможно просто взять и объединить все сценарии в документе и запустить получившийся результат как один большой сценарий уже после окончания разбора документа, потому что любой сценарий, находящийся в документе, может изменить этот документ (обсуждение атрибута `defer` приводится в разделе 13.2.4).

13.2.2. Сценарии во внешних файлах

Тег `<script>` поддерживает атрибут `src`. Значение этого атрибута задает URL-адрес файла, содержащего JavaScript-код. Используется он следующим образом:

```
<script src="../../javascript/util.js"></script>
```

Файл JavaScript-кода обычно имеет расширение `.js` и содержит JavaScript-код в «чистом виде» без тегов `<script>` или любого другого HTML-кода.

Тег `<script>` с атрибутом `src` ведет себя точно так же, как если бы содержимое указанного файла JavaScript-кода находилось непосредственно между тегами `<script>` и `</script>`. Любой код, указанный между этими тегами, игнорируется броузера-

ми. Обратите внимание, что закрывающий тег </script> обязателен даже в том случае, когда указан атрибут src и между тегами отсутствует JavaScript-код.

Использование тега с атрибутом src дает ряд преимуществ:

- HTML-файлы становятся проще, т. к. из них можно убрать большие блоки JavaScript-кода, что помогает отделить содержимое от поведения. Атрибут src является краеугольным камнем применения парадигмы ненавязчивого JavaScript-кода (подробнее об этой парадигме рассказывалось в разделе 13.1.5).
- JavaScript-функцию или другой JavaScript-код, используемый несколькими HTML-файлами, можно держать в одном файле и считывать при необходимости. Это уменьшает объем занимаемой дисковой памяти и намного облегчает поддержку программного кода.
- Когда JavaScript-функции требуются нескольким страницам, размещение кода в виде отдельного файла позволяет броузеру кэшировать его и тем самым ускорять загрузку. Когда JavaScript-код совместно используется несколькими страницами, экономия времени, достигаемая за счет кэширования, явно перевешивает небольшую задержку, требуемую броузеру для открытия отдельного сетевого соединения и загрузки файла JavaScript-кода при первом запросе на его исполнение.
- Атрибут src принимает в качестве значения произвольный URL-адрес, поэтому JavaScript-программа или веб-страница с одного веб-сервера может воспользоваться кодом (например, из библиотеки подпрограмм), предоставляемым другими веб-серверами.

У последнего пункта есть важное следствие, имеющее отношение к обеспечению безопасности. Политика общего происхождения, описываемая в разделе 13.8.2, предотвращает возможность взаимодействия документов из одного домена с содержимым из другого домена. Однако следует отметить, что источник получения самого сценария не имеет значения, значение имеет источник получения документа, в который встраивается сценарий. Таким образом, политика общего происхождения в данном случае неприменима: JavaScript-код может взаимодействовать с документами, в которые он встраивается, даже если этот код получен из другого источника, нежели сам документ. Включая сценарий в свою веб-страницу с помощью атрибута src, вы предоставляете автору сценария (или веб-мастеру домена, откуда загружается сценарий) полный контроль над своей веб-страницей.

13.2.3. Определение языка сценариев

Хотя JavaScript изначально был языком сценариев для Всемирной паутины и остается в ней самым распространенным, он не единственный. HTML-спецификации нейтральны к выбору языка сценариев, благодаря чему производители броузеров могут выбирать языки сценариев по своему усмотрению. На практике же единственной серьезной альтернативой JavaScript является язык Visual Basic Scripting Edition корпорации Microsoft[1], который поддерживается Internet Explorer.

---

[1] Также известный как VBScript. Он поддерживается *только* Internet Explorer, поэтому сценарии, написанные на этом языке, непереносимы. VBScript взаимодействует с HTML-объектами так же, как и JavaScript, но синтаксис самого языка сильно отличается от JavaSript. В данной книге VBScript не описывается.

Поскольку существует возможность использования более одного языка сценариев, необходимо сообщить веб-броузеру, на каком языке написан сценарий. Это позволяет корректно интерпретировать сценарии и пропускать сценарии, написанные на языках, которые не поддерживаются. Существует возможность определить язык сценариев для всего файла с помощью HTTP-заголовка `Content-Script-Type`. Имитировать этот заголовок в HTML-файле можно с помощью тега `<meta>`. Чтобы указать, что все сценарии написаны на языке JavaScript (если не указано иное), достаточно просто поместить следующий тег в секцию `<head>` HTML-документа:

```
<meta http-equiv="Content-Script-Type" content="text/javascript">
```

На практике броузеры полагают, что JavaScript является языком сценариев по умолчанию, даже если сервер не присылает заголовок `Content-Script-Type` и в странице опущен тег `<meta>`. Однако если язык сценариев по умолчанию не определен или возникает необходимость изменить значение по умолчанию, необходимо использовать атрибут `type` тега `<script>`:

```
<script type="text/javascript"></script>
```

Традиционно для программ на языке JavaScript указывался MIME-тип `"text/javascript"`. Другой используемый тип – `"application/x-javascript"` (где префикс `x-` указывает, что это нестандартный экспериментальный тип). Тип `"text/javascript"` стандартизован в RFC 4329 как наиболее распространенный. Однако поскольку JavaScript-программы в действительности не являются текстовыми документами, такой тип считается устаревшим и рекомендуется указывать вместо него тип `"application/javascript"` (без префикса `x-`). Однако на момент написания этих строк тип `"application/javascript"` не обладал достаточной поддержкой. Как только эта поддержка появится, правильнее будет использовать теги `<script>` и `<meta>` следующим образом:

```
<script type="application/javascript"></script>
<meta http-equiv="Content-Script-Type" content="application/javascript">
```

Когда тег `<script>` только появился, он был просто нестандартным расширением языка HTML и не поддерживал атрибут `type`. В то время язык сценариев определялся с помощью атрибута `language`:

```
<script language="JavaScript">
    // Здесь располагается JavaScript-код
</script>
```

А если сценарий был написан на языке VBScript, атрибут выглядел следующим образом:

```
<script language="VBScript">
  ' Программный код VBScript (' - признак комментария, аналог // в JavaScript)
</script>
```

Спецификация HTML 4 стандартизует тег `<script>`, но отвергает атрибут `language`, т. к. стандартный набор имен языков сценариев не определен. Однако иногда можно встретить тег `<script>`, в котором используются и атрибут `type` (в соответствии с требованиями стандарта), и атрибут `language` (для сохранения обратной совместимости с устаревшими версиями броузеров):

```
<script type="text/javascript" language="JavaScript"></script>
```

Атрибут `language` иногда служит для указания версии языка JavaScript, на котором написан сценарий:

```
<script language="JavaScript1.2"></script>
<script language="JavaScript1.5"></script>
```

Теоретически веб-броузеры игнорируют сценарии, написанные на неподдерживаемой версии JavaScript. Так, устаревшие версии броузеров, не поддерживающие JavaScript 1.5, не должны запускать сценарии, для которых в атрибуте `language` указана строка `"JavaScript1.5"`. Старые версии броузеров принимают во внимание номер версии, но поскольку ядро языка JavaScript остается стабильным на протяжении последних нескольких лет, многие современные броузеры игнорируют любые номера версий, указанные в атрибуте `language`.

13.2.4. Атрибут defer

Как уже упоминалось, сценарий может вызывать метод `document.write()` для динамического добавления содержимого в документ. Поэтому когда HTML-анализатор встречает сценарий, он должен прекратить разбор документа и ожидать, пока сценарий не завершит свою работу. Стандарт HTML 4 определяет атрибут `defer` для тега `<script>`, который имеет отношение к этой проблеме.

Если сценарий не выполняет какого-либо вывода в документ, например определяет функцию `document.write()`, но нигде ее не вызывает, то с помощью атрибута `defer` тега `<script>` можно сообщить броузеру, чтобы он спокойно продолжал обработку HTML-документа и отложил исполнение сценария до тех пор, пока не будет найден сценарий, выполнение которого отложено быть не может. Задержка исполнения сценария полезна, когда сценарий загружается из внешнего файла; если исполнение сценария не задержать, броузер вынужден будет ждать окончания загрузки и только потом сможет продолжить разбор содержимого документа. Задержка исполнения может привести к повышению производительности броузеров, способных использовать преимущества атрибута `defer`. В HTML у атрибута `defer` не может быть значения; он просто должен присутствовать в теге:

```
<script defer>
    // Любой JavaScript-код, не вызывающий document.write()
</script>
```

Однако в XHTML значение этого атрибута должно быть указано:

```
<script defer="defer"></script>
```

К моменту написания этих строк Internet Explorer был единственным броузером, использующим атрибут `defer`. При этом задержка выполняется, только когда тег `<script>` содержит атрибут `src`. Однако реализация задержки выполнена не совсем корректно, поскольку исполнение сценария с атрибутом `defer` всегда откладывается до окончания разбора документа, а не до того момента, когда встретится первый сценарий, исполнение которого нельзя отложить. Это означает, что отложенные сценарии в IE могут исполняться не в том порядке, в котором они располагаются в теле документа. В результате некоторые функции или переменные, востребованные в сценариях, исполнение которых не откладывалось, могут быть не определены.

13.2.5. Тег <noscript>

Язык разметки HTML определяет элемент <noscript>, предназначенный для хранения отображаемого содержимого на случай, когда в броузере включен режим, запрещающий исполнение JavaScript-кода. В идеале веб-страницы должны создаваться так, чтобы JavaScript-код лишь расширял их функциональные возможности, а в случае его отключения страницы сохраняли свою работоспособность. Однако если это невозможно, с помощью тега <noscript> можно известить пользователя о том, что требуется включить поддержку JavaScript и, возможно, предоставить ссылку на альтернативную страницу.

13.2.6. Тег </script>

В какой-то момент вам может потребоваться с помощью метода document.write() или свойства innerHTML вывести некоторый другой сценарий (обычно в другое окно или фрейм). Тогда для завершения генерируемого сценария потребуется вывести тег </script>. Здесь необходима осторожность – HTML-анализатор не пытается понять JavaScript-код, и встретив строку "/script" даже внутри кавычек, он предположит, что это закрывающий тег выполняемого в данный момент сценария. Чтобы обойти это препятствие, разбейте тег на части и запишите его, например, в виде выражения "</" + "script>", как показано в следующем фрагменте:

```
<script>
f1.document.write("<script>");
f1.document.write("document.write('<h2>Это сценарий в кавычках</h2>')");
f1.document.write("</" + "script>");
</script>
```

В качестве альтернативы можно экранировать символ слэша / в теге </script> с помощью символа обратного слэша:

```
f1.document.write("<\/script>");
```

В XHTML сценарии заключаются в секцию CDATA и потому проблема с закрывающим тегом </script> никак не проявляется.

13.2.7. Сокрытие сценариев от устаревших броузеров

Когда JavaScript еще был в диковинку, некоторые броузеры не распознавали тег <script> и потому (вполне корректно) отображали содержимое этого тега, как простой текст. Пользователь, посетивший веб-страницу, мог увидеть JavaScript-код, оформленный в виде больших и бессмысленных абзацев и представляемый как содержимое веб-страницы! Чтобы обойти эту проблему, внутри тега <script> использовались HTML-комментарии. Обычно программисты оформляли свои сценарии следующим образом:

```
<script language="JavaScript">
<!--Начало HTML-комментария, который скрывает расположенный здесь
    // текст JavaScript-сценария
    //              .
    //              .
    // Конец HTML-комментария, скрывающего текст сценария -->
</script>
```

Или более компактно:

```
<script><!--
// здесь находится тело сценария
//--></script>
```

Это повлекло за собой внесение изменений в ядро языка JavaScript, чтобы последовательность символов <!-- в начале сценария воспринималась как однострочный комментарий //.

Хотя броузеры, для которых требовалось оформлять сценарий в виде комментария, давно сошли со сцены, подобный код еще можно встретить в существующих веб-страницах.

13.2.8. Нестандартные атрибуты тега <script>

В корпорации Microsoft были определены два нестандартных атрибута тега <script>, которые работают только в Internet Explorer. Атрибуты event и for позволяют задавать обработчики событий с помощью тега <script>. Атрибут event определяет имя обрабатываемого события, а атрибут for – имя, или идентификатор (ID), элемента, для которого этот обработчик предназначается. Сценарий исполняется, когда в заданном элементе возникает заданное событие.

Эти атрибуты работают только в IE, а достигаемый ими эффект легко может быть реализован другими способами. Эти атрибуты никогда не следует использовать – они упомянуты здесь лишь для того, чтобы вы знали об их существовании, если вдруг придется столкнуться с ними в существующих веб-страницах.

13.3. Обработчики событий в HTML

JavaScript-код, расположенный в теге <script>, исполняется один раз, когда содержащий его HTML-файл считывается в веб-броузер. Такие статические сценарии не могут динамически реагировать на действия пользователя. В динамических программах определяются обработчики событий, автоматически вызываемые веб-броузером при возникновении определенных событий, например при щелчке на кнопке в форме. События в клиентском языке JavaScript генерируются HTML-объектами (такими как кнопки), поэтому обработчики событий определяются как атрибуты этих объектов. Например, чтобы задать обработчик события, который вызывается, когда пользователь щелкает на флажке в форме, код обработчика указывается в качестве атрибута HTML-тега, определяющего флажок:

```
<input type="checkbox" name="options" value="gifwrap"
       onclick="gifwrap = this.checked;"
>
```

Здесь нас интересует атрибут onclick. Строковое значение атрибута onclick может содержать одну или несколько JavaScript-инструкций. Если имеется несколько инструкций, они должны отделяться друг от друга точками с запятой. Когда с флажком происходит указанное событие (в данном случае щелчок мыши), исполняется JavaScript-код, указанный в этой строке.

В определение обработчика события можно включать любое количество JavaScript-инструкций, но обычно для обработки события в атрибут вставляется вызов

функции, которая определена где-нибудь в другом месте между тегами <script> и </script>. Это позволяет держать большую часть JavaScript-кода внутри тегов <script> и ограничивает степень взаимопроникновения JavaScript- и HTML-кода.

Примечательно, что атрибуты обработчиков событий являются не единственным местом определения JavaScript-обработчиков. В главе 17 показано, что существует возможность определять обработчики событий для HTML-элементов, располагая JavaScript-код внутри тега <script>. Некоторые JavaScript-разработчики призывают отказаться от использования HTML-атрибутов для определения обработчиков событий, мотивируя это требование парадигмой ненавязчивого JavaScript-кода, в соответствии с которой необходимо полное отделение содержимого от поведения. Согласно такому стилю программирования весь JavaScript-код должен размещаться во внешних файлах, ссылки на которые должны оформляться в виде атрибутов src тегов <script>. Этот внешний JavaScript-код во время своей работы может определить любые обработчики событий, какие только потребуются.

Намного более подробно события и их обработчики обсуждаются в главе 17, но многие примеры их использования мы уже неоднократно рассматривали. В главе 17 содержится полный список обработчиков событий, а наиболее распространенные из них мы перечислим здесь:

onclick

Этот обработчик поддерживается всеми элементами форм, подобными кнопкам, а также тегами <a> и <area>. Он вызывается, когда пользователь щелкает на элементе. Если обработчик onclick возвращает false, броузер не выполняет стандартное действие, связанное с элементом или ссылкой, например, не открывает ссылку (для тега <a>) или не передает данные формы (для кнопки Submit).

onmousedown, onmouseup

Эти два обработчика во многом похожи на onclick, но вызываются по отдельности, когда пользователь нажимает и отпускает кнопку мыши. Большинство элементов документа поддерживают эти обработчики.

onmouseover, onmouseout

Эти два обработчика события вызываются, когда указатель мыши соответственно оказывается на элементе документа или покидает его.

onchange

Этот обработчик события поддерживается тегами <input>, <select> и <textarea>. Он вызывается, когда пользователь изменяет значение, отображаемое элементом, а затем перемещает фокус с помощью клавиши табуляции либо другим способом.

onload

Этот обработчик событий может использоваться в теге <body>. Данное событие возникает, когда документ и все содержимое из внешних файлов (например, изображения) полностью загружено. Обработчик onload часто используется для запуска программного кода, который манипулирует содержимым документа, т. к. это событие свидетельствует о том, что документ достиг состояния готовности и его можно изменять.

Реализацию обработчиков событий можно найти в интерактивном сценарии выплат по закладной в примере 1.3. HTML-форма в этом примере содержит несколько атрибутов обработчиков событий. Тело этих обработчиков просто: они лишь вызывают функцию `calculate()`, определенную в другом месте внутри тега `<script>`.

13.4. JavaScript в URL

Еще один способ исполнения JavaScript-кода на стороне клиента – написание этого кода в URL-адресе вслед за спецификатором псевдопротокола `javascript:`. Этот специальный тип протокола обозначает, что тело URL-адреса представляет собою произвольный JavaScript-код, который должен быть выполнен интерпретатором JavaScript. URL-адрес интерпретируется как единственная строка и потому инструкции в ней должны быть отделены друг от друга точками с запятой, а для комментариев следует использовать комбинации символов `/* */`, а не `//`. Подобный URL-адрес может выглядеть, например, так:

```
javascript:var now = new Date(); "<h1>Время:</h1>" + now;
```

Когда броузер загружает такой URL-адрес, он исполняет содержащийся в нем код и использует строковое значение последней JavaScript-инструкции в качестве содержимого нового отображаемого документа. Это строковое значение может содержать HTML-теги, оно форматируется и отображается точно так же, как любой другой документ, загруженный в броузер.

URL-адрес с JavaScript-кодом может также содержать JavaScript-инструкции, выполняющие действия, но не возвращающие значения. Например:

```
javascript:alert("Hello World!")
```

Когда загружается подобный URL-адрес, броузер исполняет JavaScript-код, но т. к. значения для вывода в новом документе нет, он не изменяет текущий документ.

Часто возникает необходимость использовать спецификатор `javascript:` в URL-адресе для исполнения некоторого кода без изменения текущего отображаемого документа. Для этого необходимо, чтобы последняя инструкция в URL-адресе не возвращала значение. Один из способов обеспечить отсутствие возвращаемого значения состоит в том, чтобы посредством оператора `void` явно указать неопределенное возвращаемое значение. Просто в конец URL-адреса со спецификатором `javascript:` поместите инструкцию:

```
void 0;
```

Вот, например, как выглядит URL-адрес, открывающий новое пустое окно броузера без изменения содержимого текущего окна:

```
javascript:window.open("about:blank"); void 0;
```

Без оператора `void` в этом URL-адресе значение, возвращаемое вызванным методом `Window.open()`, было бы преобразовано в строку и отображено, в результате текущий документ был бы замещен документом, в котором присутствовало что-то вроде следующего:

```
[object Window]
```

URL-адрес со спецификатором `javascript:` можно указывать везде, где используется обычный URL-адрес. Один из важных приемов применения этого синтаксиса – его ввод непосредственно в адресную строку броузера. Так можно проверять на исполнение произвольный JavaScript-код без необходимости открывать редактор и создавать HTML-файл с этим кодом.

Спецификатор псевдопротокола `javascript:` может использоваться в HTML-атрибутах везде, где используются строки URL-адресов. Атрибут `href` гиперссылки – одно из таких мест. Когда пользователь щелкает на такой ссылке, исполняется указанный JavaScript-код. В данном контексте URL-адрес со спецификатором `javascript:` является, по сути, заменой обработчика события `onclick`. (Следует отметить, что и использование обработчика события `onclick` или URL-адреса со спецификатором `javascript:` в HTML-гиперссылках – это признак плохо продуманного дизайна; для нужд приложения следует применять кнопки и другие элементы управления, а гиперссылки оставить только для загрузки новых документов.) Аналогичным образом URL-адрес со спецификатором `javascript:` может указываться в качестве значения атрибута `action` тега `<form>` – благодаря этому при принятии пользователем формы выполняется JavaScript-код.

URL-адрес со спецификатором `javascript:` может также передаваться методам, таким как `Window.open()` (подробности см. в главе 14), которые ожидают получить строку URL-адреса в качестве аргумента.

13.4.1. Букмарклеты

Одной из особенно важных областей применения URL-адресов со спецификатором `javascript:` являются закладки, где они выступают в качестве мини-программ на языке JavaScript, или *букмарклетов* (*bookmarklet*). Букмарклеты легко можно запустить из меню или панели инструментов с закладками. Следующий фрагмент кода в качестве значения атрибута `href` включает в себя тег `<a>`, содержащий URL-адрес со спецификатором `javascript:`. Щелчок на ссылке открывает простейший обработчик JavaScript-выражений, который позволяет вычислять выражения и исполнять инструкции в контексте страницы:

```
<a href='javascript:
var e = "", r = ""; /* Вычисляемое выражение и результат */
do {
    /* Отобразить выражение и результат, а затем запросить новое выражение */
    e = prompt("Выражение: " + e + "\n" + r + "\n", e);
    try { r = "Результат: " + eval(e); } /* Попробовать вычислить выражение */
    catch(ex) { r = ex; }                /* Или запомнить ошибку */
} while(e); /* продолжать, пока не будет введено пустое выражение, */
            /* или щелкнуть на кнопке отмены */
void 0;     /* Это предотвращает замену текущего документа */
'>
Обработчик JavaScript-выражений
</a>
```

Обратите внимание: несмотря на то, что этот программный код записан в нескольких строках, синтаксический анализатор обработает его как одну строку, а потому однострочные комментарии (`//`) здесь работать не будут. Вот как выглядит тот же программный код после удаления лишних пробелов и комментариев:

```
<a href='javascript:var e="",r="";do{e=prompt("Выражение: "+e+"\n"+r+"\n",e);
try{r="Результат: "+eval(e);}catch(ex){r=ex;}}while(e);void 0;'>Обработчик
                                                    JavaScript-выражений</a>
```

Ссылки, подобные этой, удобны, когда они «зашиты» в тело разрабатываемой страницы, но еще более удобны, когда они хранятся как закладки, которые можно запустить из любой страницы. Обычно закладки создаются щелчком правой кнопкой мыши на странице и выбором в контекстном меню пункта Добавить страницу в закладки или подобного ему. В броузере Firefox для этого достаточно просто перетащить ссылку на панель закладок.

Все приемы программирования на клиентском языке JavaScript, описываемые в этой книге, в равной степени могут использоваться для создания букмарклетов, но сами они в этой книге подробно не описываются. Если вас заинтересовали возможности этих маленьких программ, попробуйте выполнить поиск в Интернете по слову «bookmarklets». Вы найдете достаточное число сайтов, где есть масса интересных и полезных букмарклетов.

13.5. Исполнение JavaScript-программ

В предыдущем разделе обсуждались механизмы интеграции JavaScript-кода в HTML-файл. Теперь обсудим, как и когда интегрированный JavaScript-код исполняется интерпретатором JavaScript.

13.5.1. Сценарии

JavaScript-инструкции, расположенные между тегами `<script>` и `</script>`, исполняются в порядке их появления. Если в файле имеется более одного сценария, они исполняются в том порядке, в котором встречаются в документе (за исключением сценариев с атрибутом `defer` – такие сценарии IE исполняет не по порядку). Исполнение JavaScript-кода является частью процесса загрузки и разбора документа.

Любой тег `<script>`, в котором отсутствует атрибут `defer`, может вызывать метод `document.write()` (подробно описан в главе 15). Текст, переданный этому методу, вставляется в документ непосредственно в то место, где находится сценарий в документе. Когда сценарий завершает работу, анализатор продолжает разбор HTML-документа, начиная с текста, который был выведен сценарием.

Сценарии могут присутствовать в разделах `<head>` или `<body>` HTML-документа. Обычно в разделе `<head>` определяются функции, вызываемые из других сценариев. Здесь также могут объявляться и инициализироваться переменные, которые будут использоваться другим кодом. Обычно в сценариях раздела `<head>` документа определяется единственная функция, которая затем регистрируется как обработчик события `onload` для последующего исполнения. Вполне допустимо, хотя на практике почти не встречается, обращение к методу `document.write()` в разделе `<head>`.

Сценарии в теге `<body>` документа могут делать все то же самое, что сценарии в теге `<head>`. Однако здесь часто можно встретить вызов метода `document.write()`. Сценарии, размещенные в теге `<body>` документа, могут также (с использованием приемов, описываемых в главе 15) обращаться к элементам и содержимому до-

кумента, находящимся перед сценарием, и изменять их. Однако, как объясняется в этой главе далее, в момент исполнения сценария, находящегося в теге <body>, доступность и готовность элементов документа не гарантируется. Если сценарий просто определяет некоторые функции и переменные для последующего использования и не пытается изменить содержимое документа вызовом метода document.write() или каким-либо другим способом, в соответствии с общепринятыми соглашениями такой сценарий должен размещаться в теге <head>, а не <body>.

Как уже упоминалось, IE исполняет сценарии с атрибутом defer не в порядке их следования. Отложенные сценарии запускаются после того, как отработают все остальные сценарии и закончится полный разбор документа, но до того, как вызван обработчик события onload.

13.5.2. Обработчик события onload

После того как весь документ проанализирован, все сценарии исполнены и все дополнительное содержимое документа (например, изображения) загружено, броузер инициирует событие onload и исполняет JavaScript-код, зарегистрированный как обработчик события onload объекта Window. Регистрация обработчика события onload может быть выполнена установкой атрибута onload тега <body>. Но для отдельных модулей JavaScript-кода существует также возможность зарегистрировать собственные обработчики события onload (с помощью приемов, описываемых в главе 17). Если было зарегистрировано более одного обработчика события onload, броузер вызовет их все, но при этом не гарантируется, что вызываться они будут в том же порядке, в котором были зарегистрированы.

К моменту вызова обработчика события onload документ должен быть уже полностью загружен и проанализирован, а потому допускать манипулирование любыми элементами документа из JavaScript-сценария. По этой причине JavaScript-модули, модифицирующие содержимое документа, обычно содержат функцию, которая выполняет модификацию, и программный код, который регистрирует обработчик события onload. Это гарантирует вызов функции только после того, как документ будет полностью загружен.

Поскольку обработчики события onload вызываются уже после того, как завершится анализ документа, они не должны вызывать метод document.write(). Любой такой вызов вместо того, чтобы добавить новое содержимое в конец существующего документа, просто уничтожит текущий документ и начнет заполнение нового еще до того, как пользователь получит шанс просмотреть его.

13.5.3. Обработчики событий и URL-адреса в JavaScript

Когда загрузка и анализ документа завершаются, вызывается обработчик события onload и исполнение JavaScript-сценариев вступает в фазу исполнения по событиям. На протяжении всей этой фазы обработчики событий вызываются асинхронно в ответ на такие действия пользователя, как перемещение указателя мыши, щелчки кнопками мыши и нажатия клавиш. URL-адреса в JavaScript также могут вызываться асинхронно на протяжении всей этой фазы, когда, например, пользователь щелкает на ссылке, в которой спецификатор псевдопротокола javascript: указан в качестве значения атрибута href.

Теги <script> обычно используются для определения функций, а обработчики событий, как правило, вызывают эти функции в ответ на действия пользователя. Конечно, обработчики событий также могут содержать определения функций, но на практике такой подход обычно не используется.

Если обработчик события вызовет метод document.write() для документа, частью которого он является, это уничтожит текущий документ и будет начат новый. Обычно это совсем не то, что нужно, и на практике обработчики событий никогда не должны вызывать этот метод или функции, которые его вызывают. Исключение составляют многооконные приложения, в которых обработчик события одного окна может вызвать метод write() документа в другом окне. (Подробнее о многооконных приложениях рассказывается в разделе 14.8.)

13.5.4. Обработчик события onunload

Когда пользователь покидает веб-страницу, броузер вызывает обработчик события onunload, давая JavaScript-сценарию последнюю возможность выполнить заключительные действия. Определить обработчик события onunload можно в теге <body> с помощью атрибута onunload, или зарегистрировав обработчик события способом, который описывается в главе 17.

Событие onunload позволяет отменить действия, выполненные обработчиком события onload или другими сценариями веб-страницы. Например, если JavaScript-приложение открывает второе окно броузера, обработчик события onunload может, наоборот, закрывать это окно, когда пользователь покидает основную страницу. Обработчик события onunload не должен выполнять продолжительные по времени операции или вызывать всплывающие диалоговые окна. Заключительные операции должны выполняться максимально быстро, чтобы не препятствовать пользователю и не заставлять его долго ждать перехода на новую страницу.

13.5.5. Объект Window как контекст исполнения

Все сценарии, обработчики событий и URL-адреса в JavaScript в качестве глобального объекта совместно используют один и тот же объект Window. Переменные и функции в JavaScript – это не более чем свойства глобального объекта. Это означает, что функции, объявленные в одном теге <script>, могут вызываться сценариями во всех последующих тегах <script>.

Поскольку обработка события onload не начинается до тех пор, пока не отработают все сценарии, каждый обработчик события onload обладает возможностью обращения к любым функциям и переменным, объявленным в сценариях документа.

Всякий раз, когда в окно броузера загружается новый документ, объект Window переводится в состояние по умолчанию: любые свойства и функции, объявленные сценариями из предыдущего документа, удаляются и восстанавливаются все переопределенные стандартные системные свойства. Каждый документ начинается с «чистого листа». Сценарии могут уверенно полагаться на это обстоятельство – они не унаследуют измененное окружение, оставшееся от предыдущего документа. Кроме того, это означает, что все переменные и функции, определяемые сценарием, будут существовать только до того момента, пока текущий документ не замещен новым.

Срок жизни *свойств* объекта `Window` совпадает со сроком жизни документа, который содержит JavaScript-код и объявляет эти свойства. Объект `Window` обладает более продолжительным временем жизни – он существует ровно столько, сколько существует соответствующее ему окно броузера. Ссылка на объект `Window` остается действительной независимо от того, сколько документов загружалось и выгружалось. Это верно только для веб-приложений, которые имеют несколько окон или фреймов. В данном случае JavaScript-код из одного окна может использовать ссылку на другое окно или фрейм, причем эта ссылка останется работоспособной, даже если в другое окно или фрейм будет загружен новый документ.

13.5.6. Модель управления потоками исполнения в клиентском JavaScript

Ядро языка JavaScript не имеет механизма одновременного исполнения нескольких потоков управления, и клиентский язык JavaScript не добавляет такой возможности. JavaScript-код на стороне клиента исполняется в единственном потоке управления. Синтаксический разбор документа останавливается, пока загружается и исполняется сценарий, а веб-броузер прекращает откликаться на действия пользователя на время исполнения обработчика события.

Исполнение в единственном потоке существенно упрощает разработку сценариев: можно писать программный код, пребывая в полной уверенности, что два обработчика событий никогда не запустятся одновременно. Можно манипулировать содержимым документа, точно зная, что никакой другой поток исполнения не попытается изменить его в то же самое время.

Однако исполнение в единственном потоке накладывает определенные требования, т. е. сценарии и обработчики событий в JavaScript не должны исполняться слишком долго. Если сценарий производит объемные и интенсивные вычисления, это вызовет задержку во время загрузки документа, и пользователь не увидит его содержимое, пока сценарий не закончит свою работу. Если продолжительные по времени операции выполняются в обработчике события, броузер может оказаться неспособным откликаться на действия пользователя, заставляя его думать, что программа «зависла».[1]

Если приложение должно выполнять достаточно сложные вычисления, вызывающие заметные задержки, то перед выполнением таких вычислений следует дать документу возможность полностью загрузиться. Кроме того, полезно предупредить пользователя, что будут производиться длительные вычисления, в процессе которых броузер может не откликаться на его действия. Если есть такая возможность, длительные вычисления следует разбить на несколько подзадач, используя такие методы, как `setTimeout()` и `setInterval()`, для запуска подзадач в фоновом режиме с одновременным обновлением индикатора хода вычислений, предоставляющего обратную связь с пользователем (см. главу 14).

---

[1] В некоторых броузерах, таких как Firefox, имеются средства предотвращения атак типа отказа в обслуживании и случайных зацикливаний. Благодаря этим средствам в случае, когда сценарий или обработчик события исполняется длительное время, перед пользователем выводится окно запроса, позволяющее прервать исполнение зациклившегося кода.

13.5.7. Манипулирование документом в процессе загрузки

В процессе загрузки и синтаксического анализа документа программный код JavaScript-сценария, расположенный в теге <script>, имеет возможность вставить содержимое в документ с помощью метода window.write(). Другие способы манипулирования документом, в которых используются приемы и методы DOM-программирования и которые представлены в главе 15, в тегах <script> могут быть доступны или не доступны.

На первый взгляд большинство броузеров предоставляют сценариям возможность манипулировать любыми элементами документа, которые расположены перед тегом <script>. Некоторые JavaScript-программисты исходят из этого предположения. Однако ни в одном стандарте такое положение вещей не регламентируется, и опытные JavaScript-программисты знают, что если не определено обратное, манипулирование элементами документа из сценариев, размещенных в тегах <script>, может вызывать проблемы (возможно лишь иногда, лишь в некоторых броузерах или лишь тогда, когда происходит перезагрузка документа или возврат к предыдущей странице после щелчка на кнопке Назад).

Единственное, что точно известно об этой темной области, – это то, что безопасно можно манипулировать документом лишь после возникновения события onload, и это обстоятельство учитывается при разработке большинства JavaScript-приложений – в них событие onload служит «сигналом» для выполнения всех необходимых модификаций документа. Вспомогательная подпрограмма, выполняющая регистрацию обработчиков события onload, представлена в примере 17.7.

Когда документ содержит большие изображения или много изображений, синтаксический разбор главного документа может завершиться задолго до того, как будут загружены все изображения и произойдет событие onload. В этом случае может потребоваться начать модификации документа до возникновения события onload. Один из способов (наиболее безопасный из обсуждаемых) заключается в том, чтобы разместить программный код в конце документа. Для IE программный код можно поместить в тег <script> с установленными атрибутами defer и src, а для Firefox – оформить его в виде недокументированного обработчика события DOMContentLoad, которое возникает после разбора содержимого документа, но до загрузки всех внешних объектов, таких как изображения.

Другая темная область модели исполнения JavaScript-кода касается вопроса, могут ли обработчики событий вызываться до того, как документ полностью загружен? До сих пор в нашем обсуждении модели исполнения JavaScript-кода мы придерживались мнения, что все обработчики событий всегда вызываются только после того, как отработают все сценарии. Так обычно и происходит, но никакие стандарты не требуют этого. Если документ очень объемный или загрузка документа происходит через медленное сетевое соединение, броузер может отобразить часть документа и позволить пользователю взаимодействовать с ним (и соответственно запускать обработчики событий) до того, как исполнены все сценарии и вызван обработчик события onload. Если в такой ситуации обработчик события обратится к функции, которая еще не определена, он потерпит неудачу. Впрочем, на практике это случается достаточно редко, потому предпринимать дополнительных усилий для реализации разного рода защитных мер не требуется.

13.6. Совместимость на стороне клиента

Веб-броузер – это универсальная платформа для исполнения приложений, а JavaScript – язык программирования, на котором эти приложения разрабатываются. К счастью, язык JavaScript относится к разряду стандартизованных и широко поддерживаемых – все современные веб-броузеры поддерживают стандарт ECMAScript v3. Чего, впрочем, нельзя сказать о самой платформе. Конечно, все веб-броузеры могут отображать HTML-документы, но они отличаются друг от друга полнотой поддержки других стандартов, таких как CSS и DOM. И хотя все современные броузеры включают совместимые реализации интерпретатора JavaScript, они имеют отличия в прикладном программном интерфейсе (Application Programming Interface, API), доступном для клиентского JavaScript-кода.

Проблемы совместимости – это просто неприятный факт из жизни программистов, использующих клиентский язык JavaScript. Разрабатываемый и распространяемый вами JavaScript-код может исполняться на различных версиях разных броузеров и в разных операционных системах. Рассмотрим наиболее часто встречающиеся комбинации операционных систем и броузеров: Internet Explorer в Windows и Mac OS[1]; Firefox для Windows, Mac OS и Linux; Safari для Mac OS и Opera для Windows, Mac OS и Linux. Если у вас появится желание реализовать поддержку всех этих броузеров, плюс две предыдущие версии каждого из них, умножьте эти девять комбинаций броузера и ОС на три, всего получится 27 комбинаций броузера, версии и ОС. Единственный способ убедиться, что ваши веб-приложения будут безошибочно исполняться в любой из 27 комбинаций, – проверить каждую комбинацию. Это титанический труд, но на практике тестирование часто производится пользователями уже после развертывания приложения.

Прежде чем в процессе разработки приложения перейти к фазе тестирования, необходимо написать программный код. Поэтому при программировании на языке JavaScript знание существующих несовместимостей в броузерах является чрезвычайно важным для создания совместимого программного кода. К сожалению, составление списка всех известных несовместимостей между производителями, версиями и платформами является непомерно сложной задачей. Это далеко выходит за рамки темы данной книги и моих собственных познаний, до сих пор еще ни разу не предпринимались попытки разработать полномасштабные наборы тестов для клиентского JavaScript. Некоторую информацию о совместимости броузеров можно найти в Интернете, причем два сайта я нахожу наиболее полезными:

http://www.quirksmode.org/dom/

Это сайт независимого веб-разработчика Питера-Пауля Коха (Piter-Paul Koch). Его таблицы совместимости с DOM отражают уровень соответствия различных броузеров стандартам W3C DOM.

http://webdevout.net/browser_support.php

Это сайт Дэвида Хаммонда (David Hammond). Он напоминает сайт *quirksmode.org*, но здесь вы найдете более полные и более свежие (на момент написа-

[1] Версия IE для Mac OS постепенно сходит со сцены, и это благо, поскольку данный броузер заметно отличается от версии IE для Windows.

ния этих строк) таблицы совместимости. В дополнение к совместимости с DOM здесь также приводятся оценки соответствия броузеров стандартам HTML, CSS и ECMAScript.

Конечно, выяснить о существовании несовместимости – это лишь первый шаг. В следующих подразделах демонстрируются приемы, используемые для обхода несовместимостей, с которыми вы можете столкнуться.

13.6.1. Происхождение несовместимости

При JavaScript-программировании на стороне клиента всегда приходилось сталкиваться с проблемой несовместимости. Знание истории даст вам понимание происходящего, что, безусловно, пригодится в работе. Ранние дни веб-программирования были отмечены так называемой «войной броузеров» между Netscape и Microsoft. Это был период бурного развития броузеров и API-интерфейсов клиентского JavaScript, часто в несовместимых направлениях. Проблемы несовместимости в этот период проявлялись наиболее остро, и некоторые сайты для их преодоления просто сообщали своим посетителям, какой броузер они должны использовать.

Война броузеров завершилась, когда корпорация Microsoft заняла доминирующее положение на рынке, а веб-стандарты, такие как CSS и DOM, стали приобретать все большее влияние. В период стабильности (или застоя), продолжавшийся, пока броузер Netscape медленно трансформировался в Firefox, Microsoft внесла в свой броузер несколько улучшений. Поддержка стандартов в обоих броузерах достигла высокого уровня или, по крайней мере, достаточного, чтобы обеспечивать совместимость будущим веб-приложениям.

К моменту написания этих строк мы, похоже, становимся свидетелями нового взрыва нововведений в броузерах. Например, все основные броузеры ныне поддерживают возможность отправки HTTP-запросов, что составляет основу новой архитектуры веб-приложений под названием Ajax (подробности см. в главе 20). Корпорация Microsoft ведет работы над созданием версии 7 своего броузера Internet Explorer, в которой должны быть решены многие проблемы безопасности и совместимости с CSS. В IE 7 обычный пользователь найдет множество изменений, но очевидно, что в этом броузере не будет нарушено никаких новых стандартов, составляющих основу веб-разработки. Однако в других броузерах такие нарушения уже наблюдаются. Например, Safari и Firefox поддерживают тег `<canvas>`, предназначенный для создания графических изображений на стороне клиента (подробности см. в главе 22). Консорциум производителей броузеров (в котором, что примечательно, корпорация Microsoft не представлена), известный как WHATWG (*whatwg.org*), работает над стандартизацией тега `<canvas>` и многих других расширений HTML и DOM.

13.6.2. Несколько слов о «современных броузерах»

Тема клиентского JavaScript весьма изменчива, особенно если учесть, что мы вступаем в фазу интенсивного развития. По этой причине в данной книге я старался избегать каких-либо утверждений о конкретных версиях конкретных броузеров. Любые такие утверждения, скорее всего, устареют еще до того, как книга увидит свет. Печатное издание не может обновляться достаточно быстро, что-

бы служить руководством по проблемам совместимости, которые затрагивают современные броузеры.

Поэтому вы часто будете видеть, что я специально подстраховываюсь, используя достаточно расплывчатую фразу «современные броузеры» (или «все современные броузеры, за исключением IE»). К моменту написания этих строк в набор «современных броузеров» входили: Firefox 1.0, Firefox 1.5, IE 5.5, IE 6.0, Safari 2.0, Opera 8 и Opera 8.5. Это не гарантирует, что каждое сделанное в книге утверждение о «современных броузерах» в равной степени будет верным для каждого из этих конкретных броузеров. Тем не менее это дает вам возможность знать, какие броузеры считались современными во время работы над этой книгой.

13.6.3. Проверка особенностей

Проверка особенностей (иногда называемая *проверкой функциональных возможностей*) – это очень мощная методика, позволяющая справиться с проблемами совместимости. Особенность, или функциональная возможность, которую вы собираетесь использовать, может поддерживаться не всеми броузерами, поэтому необходимо включать в свои сценарии программный код, который будет проверять факт поддержки данной особенности. Если требуемая особенность не поддерживается на текущей платформе, тогда можно будет либо не использовать эту особенность на данной платформе, либо разработать альтернативный программный код, одинаково работоспособный на всех платформах.

В следующих главах вы часто будете видеть, что та или иная особенность проверяется снова и снова. Например, в главе 17 приводится программный код, который выглядит примерно следующим образом:

```
if (element.addEventListener) { // Проверить наличие метода W3C перед вызовом
    element.addEventListener("keydown", handler, false);
    element.addEventListener("keypress", handler, false);
}
else if (element.attachEvent) { // Проверить наличие метода IE перед вызовом
    element.attachEvent("onkeydown", handler);
    element.attachEvent("onkeypress", handler);
}
else { // В противном случае использовать универсальный прием
    element.onkeydown = element.onkeypress = handler;
}
```

В главе 20 описывается еще один подход к проверке особенностей: перебирать доступные альтернативы, пока не будет обнаружена та, которая не генерирует исключение! А после обнаружения работоспособной альтернативы она запоминается для последующего использования. Вот как выглядит фрагмент из примера 20.1:

```
// Список функций создания объекта XMLHttpRequest, которые следует опробовать
HTTP._factories = [
    function() { return new XMLHttpRequest(); },
    function() { return new ActiveXObject("Msxml2.XMLHTTP"); },
    function() { return new ActiveXObject("Microsoft.XMLHTTP"); }
];

// Когда обнаружится работоспособный метод, запомнить его здесь
HTTP._factory = null;
```

```
// Создать и вернуть новый объект XMLHttpRequest.
//
// При первом обращении пробовать вызывать функции из списка, пока не будет
// обнаружена та, которая возвращает непустое значение и не генерирует
// исключение. Когда будет обнаружена работоспособная функция, запомнить
// ее для последующего использования.
HTTP.newRequest = function() { /* тело функции опущено */ }
```

Для проверки версии DOM, поддерживаемой броузером, используется несколько устаревший способ проверки особенностей, который довольно часто еще можно встретить в существующем программном коде. Обычно его можно встретить в DHTML-коде, и выглядит он примерно так:

```
if (document.getElementById) { // Если поддерживается W3C DOM API,
    // исполнить DHTML-код, использующий W3C DOM API
}
else if (document.all) {       // Если поддерживается IE 4 API,
    // исполнить DHTML-код, использующий IE 4 API
}
else if (document.layers) { // Если поддерживается Netscape 4 API,
    // исполнить DHTML-эффект (максимум, что нам доступно)
    // с использованием Netscape 4 API
}
else { // В противном случае DHTML не поддерживается,
    // поэтому следует предоставить статичную альтернативу DHTML
}
```

Подобный подход считается устаревшим, потому что все современные броузеры поддерживают стандарт W3C DOM и его функцию `document.getElementById()`.

Самое главное, что дает проверка особенностей, – это программный код, который не привязан к конкретным броузерам или их версиям. Этот прием работает со всеми броузерами, существующими ныне, и должен продолжить работать с будущими версиями броузеров независимо от того, какой набор особенностей они реализуют. Однако следует отметить, что при этом производители броузеров не должны определять свойства и методы, не обладающие полной функциональностью. Если бы корпорация Microsoft определила метод `addEventHandler()`, реализовав спецификации W3C лишь частично, это привело бы к нарушениям работоспособности большого числа сценариев, в которых перед вызовом `addEventHandler()` реализован механизм проверки особенностей.

Свойство `document.all`, показанное в следующем примере, заслуживает отдельного упоминания. Массив `document.all` впервые появился в Microsoft IE 4. Он позволил JavaScript-коду обращаться ко всем элементам документа и стал предвестником новой эры разработки клиентских сценариев. Тем не менее он так и не был стандартизован и был заменен методом `document.getElementById()`. В настоящее время он по-прежнему встречается в сценариях и часто (хотя и неправильно) служит для того, чтобы выяснить, исполняется сценарий под управлением IE или нет:

```
if (document.all) {
    // Сценарий исполняется в IE
}
else {
```

```
    // Сценарий исполняется в каком-то другом броузере
}
```

Поскольку до сих пор существует большое число сценариев, использующих свойство `document.all`, в броузер Firefox была добавлена поддержка этого свойства, чтобы Firefox мог исполнять те фрагменты программного кода, которые ранее были доступны только для IE. Поскольку присутствие свойства `all` часто служит для определения типа броузера, Firefox имитирует отсутствие этого свойства. Таким образом, даже при том, что Firefox поддерживает свойство `document.all`, инструкция `if` в следующем фрагменте сценария ведет себя так, как если бы свойство `all` не существовало, благодаря чему этот сценарий выводит диалоговое окно с надписью «Firefox».

```
if (document.all) alert("IE"); else alert("Firefox");
```

Данный пример демонстрирует, что механизм проверки особенностей не работает, если броузер активно сопротивляется этому! Кроме того, этот пример показывает, что веб-разработчики не единственные, кого мучает проблема совместимости. Производители броузеров также вынуждены идти на всякие хитрости с целью обеспечения совместимости.

13.6.4. Проверка типа броузера

Методика проверки особенностей прекрасно подходит для определения поддерживаемых функциональных возможностей броузера. Ее можно использовать, например, чтобы выяснить, какая модель обработки событий поддерживается, W3C или IE. В то же время, иногда может потребоваться обойти те или иные ошибки, свойственные конкретному типу броузеров, когда нет достаточно простого способа определить наличие этих ошибок. В этом случае бывает необходимо разработать программный код, который должен исполняться только в броузерах определенного производителя, определенного номера версии или в конкретной операционной системе (либо в некоторой комбинации всех трех признаков).

На стороне клиента сделать это можно с помощью объекта `Navigator`, о котором рассказывается в главе 14. Программный код, который определяет производителя и версию броузера, часто называют *анализатором броузера* (*browser sniffer*), или *анализатором клиента* (*client sniffer*). Простой анализатор такого типа приводится в примере 14.3. Методика определения типа клиента широко использовалась на ранних этапах развития Всемирной паутины, когда Netscape и IE имели серьезные отличия и были несовместимы. Ныне ситуация с совместимостью стабилизировалась, анализ типа клиента утратил свою актуальность и проводится лишь в тех случаях, когда это действительно необходимо.

Примечательно, что определение типа клиента может быть выполнено также на стороне сервера, благодаря чему веб-сервер на основе строки идентификации броузера, которая передается серверу в заголовке `User-Agent`, может выяснить, какой JavaScript-код требуется отсылать.

13.6.5. Условные комментарии в Internet Explorer

На практике вы можете обнаружить, что большинство несовместимостей, которые необходимо учитывать при разработке клиентских сценариев, обусловлены спецификой броузера IE. Вследствие этого иногда возникает необходимость соз-

давать программный код отдельно для IE и отдельно для всех остальных броузеров. Хотя обычно нужно стараться избегать применения нестандартных расширений, присущих конкретному типу броузера, броузер IE поддерживает возможность создания условных комментариев в JavaScript-коде, что может оказаться полезным.

В следующем примере демонстрируется, как выглядят условные комментарии в HTML. Примечательно, что вся хитрость заключается в комбинации символов, закрывающих комментарий.

```
<!--[if IE]>
Эти строки фактически находятся внутри HTML-комментария.
Они будут отображаться только в IE.
<![endif]-->

<!--[if gte IE 6]>
Эта строка будет отображена только в IE 6 или более поздних версиях.
<![endif]-->

<!--[if !IE]> <-->
Это обычное HTML-содержимое, но IE не будет отображать его
из-за комментариев, что расположены выше и ниже.
<!--> <![endif]-->

Это обычное содержимое, которое будет отображаться всеми броузерами.
```

Условные комментарии также поддерживаются интерпретатором JavaScript в IE, а программисты, знакомые с языком C/C++, найдут их похожими на инструкции препроцессора `#ifdef/#endif`. Условные JavaScript-комментарии в IE начинаются с комбинации символов `/*@cc_on` и завершаются комбинацией `@*/`. (Префиксы «cc» и «cc_on» происходят от фразы «condition compilation», т. е. «условная компиляция».) Следующий условный комментарий содержит программный код, который может быть исполнен только в IE:

```
/*@cc_on
    @if (@_jscript)

        // Следующий код находится внутри JS-комментария, но IE исполнит его.
        alert("In IE");

    @end
 @*/
```

Внутри условных комментариев могут указываться ключевые слова `@if`, `@else` и `@end`, предназначенные для отделения программного кода, который должен исполняться интерпретатором JavaScript в IE по определенному условию. В большинстве случаев вам достаточно будет использовать показанное в предыдущем фрагменте условие `@if (@_jscript)`. JScript – это название интерпретатора JavaScript, которое было дано ему в Microsoft, а переменная `@_jscript` в IE всегда имеет значение `true`.

При грамотном чередовании условных и обычных JavaScript-комментариев можно определить, какой блок программного кода должен исполняться в IE, а какой во всех остальных броузерах:

```
/*@cc_on
    @if (@_jscript)
```

```
        // Этот блок кода находится внутри условного комментария,
        // который также является обычным JavaScript-комментарием. В IE этот блок
        // будет выполнен, а в других броузерах - нет.
        alert('Вы пользуетесь Internet Explorer');
    @else*/
        // Этот блок уже не находится внутри JavaScript-комментария, но по-прежнему
        // находится внутри условного комментария IE. Вследствие этого данный
        // блок кода будет выполнен всеми броузерами, за исключением IE.
        alert('Вы не пользуетесь Internet Explorer');
    /*@end
@*/
```

Условные комментарии в HTML и JavaScript совершенно не стандартизованы, но иногда они могут оказаться полезными в обеспечении совместимости с IE.

13.7. Доступность

Всемирная паутина представляет собой замечательный инструмент распространения информации, и JavaScript-сценарии могут сделать эту информацию максимально доступной. Однако JavaScript-программисты должны проявлять осторожность: слишком просто написать такой JavaScript-код, который сделает невозможным восприятие информации для пользователей с ограниченными возможностями.

Пользователи с ослабленным зрением применяют такие «вспомогательные технологии», как программы чтения с экрана, когда слова, выводимые на экран, преобразуются в речевые аналоги. Некоторые программы чтения с экрана способны распознавать JavaScript-код, другие лучше работают, когда режим исполнения JavaScript-сценариев отключен. Если вы разрабатываете сайт, который требует исполнения JavaScript-кода на стороне клиента для отображения информации, вы ограничиваете доступность своего сайта для пользователей подобных программ чтения с экрана. (Кроме того, вы ограничиваете доступность своего сайта для всех тех, кто просматривает Интернет с помощью мобильных устройств, таких как сотовые телефоны, не поддерживающие JavaScript, а также для тех, кто преднамеренно отключил режим исполнения JavaScript-сценариев в броузере.) Главная цель JavaScript заключается в *улучшении* представления информации, а не собственно в ее представлении. Основное правило JavaScript-программирования заключается в том, что веб-страница, в которую встроен JavaScript-код, должна оставаться работоспособной (хотя бы ограниченно), даже когда интерпретатор JavaScript отключен.

Другое важное замечание относительно доступности касается пользователей, которые могут работать с клавиатурой, но не могут (или не хотят) применять указывающие устройства, такие как мышь. Если программный код ориентирован на события, возникающие от действий мышью, вы ограничиваете доступность страницы для тех, кто не пользуется мышью. Веб-броузеры позволяют задействовать клавиатуру для перемещения и активации веб-страниц, то же самое должен позволять делать JavaScript-код. Одновременно с этим не следует писать программный код, который ориентирован исключительно на ввод с клавиатуры, иначе страница окажется недоступной для тех, у кого нет клавиатуры, например для пользователей наладонных компьютеров или сотовых телефонов. Как демонстрируется в главе 17, наряду с поддержкой событий, зависящих от типа

устройства, таких как onmouseover или onmousedown, JavaScript обладает поддержкой событий, от типа устройства не зависящих, таких как onfocus и onchange. Для достижения максимальной доступности следует отдавать предпочтение событиям, не зависящим от типа устройства.

Создание максимально доступных веб-страниц – нетривиальная задача, не имеющая четких решений. Во время написания этих строк не прекращаются споры о том, как с помощью JavaScript сделать веб-страницы не менее, а более доступными. Полное обсуждение вопросов доступности далеко выходит за рамки темы этой книги. Однако поиск по Интернету даст вам массу информации по этой теме, причем большая ее часть находится в форме рекомендаций из авторитетных источников. Не следует забывать, что как приемы JavaScript-программирования на стороне клиента, так и теория доступности продолжают развиваться, и соответствующие рекомендации относительно доступности не всегда за ними успевают.

13.8. Безопасность в JavaScript

Проблема обеспечения безопасности в Интернете – это очень обширная и сложная тема. В данном разделе рассматриваются вопросы безопасности клиентского JavaScript-кода.

13.8.1. Чего не может JavaScript

Наличие интерпретаторов JavaScript в веб-броузерах означает, что загружаемая веб-страница может вызвать на исполнение произвольный JavaScript-код. Безопасные веб-броузеры, а наиболее распространенные современные броузеры обеспечивают достаточный уровень безопасности, ограничивают теми или иными способами возможность исполнять сценарии, мешая злонамеренному программному коду получить доступ к конфиденциальным данным, изменить персональные данные пользователя или поставить под угрозу безопасность системы.

JavaScript – это первая линия обороны против злонамеренного кода, поэтому некоторые функциональные возможности этим языком преднамеренно не поддерживаются. Например, клиентский JavaScript-код не предоставляет никакого способа записи или удаления файлов и каталогов на клиентском компьютере. Без объекта File и функций доступа к файлам JavaScript-программа не может удалять данные или разводить вирусы в пользовательской системе.

Вторая линия обороны – это наложение ограничений на некоторые поддерживаемые функциональные возможности. Например, клиентский JavaScript-код способен организовать обмен с веб-серверами по протоколу HTTP и может даже загружать данные с FTP-серверов, а также серверов других типов. Однако язык JavaScript не предоставляет универсальных сетевых примитивов и не может открыть сокет или принять запрос на соединение от другого хоста.

Следующий список включает в себя другие функциональные возможности, которые могут ограничиваться. Следует отметить, что это – не окончательный список. Различные броузеры накладывают разные ограничения, причем многие из этих ограничений могут устанавливаться пользователем:

- JavaScript-программа может открыть новое окно броузера, но вследствие того, что многие рекламодатели злоупотребляют этой возможностью, большин-

ство броузеров позволяют ограничить эту возможность так, чтобы всплывающие окна могли появиться только в ответ на действия пользователя, такие как щелчок мыши.

- JavaScript-программа не может закрыть другое окно броузера без подтверждения пользователя, если только программа сама не открыла это окно. Это не дает злонамеренным сценариям вызвать метод `self.close()`, чтобы закрыть окно броузера и выйти из программы.
- JavaScript-программа не может скрыть адрес ссылки, который появляется в строке состояния, когда указатель мыши оказывается на ссылке. (В прошлом такая возможность была, и обычно она использовалась для предоставления дополнительной информации о ссылке. Многочисленные случаи жульничества и злоупотребления заставили производителей броузеров от нее отказаться.)
- Сценарий не может открыть слишком маленькое окно (размеры которого по одной стороне меньше 100 пикселов) или чрезмерно уменьшить размеры окна. Аналогично сценарий не может переместить окно за пределы экрана или создать окно, превышающее размеры экрана. Это не дает сценариям открывать окна, которые пользователь не может видеть или легко может не заметить; такие окна могут содержать сценарии, продолжающие работать после того, как пользователь решил, что они завершились. Кроме того, сценарий не может создавать окна без заголовка, потому что такое окно может имитировать системное диалоговое окно и обманом заставить пользователя ввести, например, секретный пароль.
- Свойство `value` HTML-элемента `FileUpload` не может быть установлено. Если бы это свойство было доступно, сценарий мог бы установить его значение равным любому желаемому имени файла и заставить форму загрузить на сервер содержимое любого указанного файла (например, файла паролей).
- Сценарий не может прочитать содержимое документов с других серверов, отличных от сервера, откуда был получен сам документ с данным сценарием. Аналогичным образом сценарий не может зарегистрировать обработчики событий в документах, полученных с других серверов. Это предотвращает возможность подсматривания данных, вводимых пользователем (таких как комбинации символов, составляющих пароль) в других страницах. Это ограничение известно как *политика общего происхождения* (*same-origin policy*) и более подробно описывается в следующем разделе.

13.8.2. Политика общего происхождения

Политика общего происхождения накладывает ограничения на содержимое Всемирной паутины, с которым JavaScript-код может взаимодействовать. Обычно эта политика вступает в игру, когда на одной веб-странице располагается несколько фреймов, включающих теги `<iframe>`, или когда открываются другие окна броузера. В этом случае политика общего происхождения ограничивает возможность JavaScript-кода из одного окна взаимодействовать с другими фреймами или окнами. В частности, сценарий может читать только свойства окон и документов, имеющих общее с самим сценарием происхождение (о том, как использовать JavaScript для работы с несколькими окнами и фреймами, рассказывается в разделе 14.8).

Кроме того, политика общего происхождения действует при работе по протоколу HTTP с помощью объекта `XMLHttpRequest`. Этот объект позволяет JavaScript-сценариям, исполняющимся на стороне клиента, отправлять произвольные HTTP-запросы, но только тому веб-серверу, откуда был загружен документ, содержащий сценарий (подробнее об объекте `XMLHttpRequest` рассказывается в главе 20).

Происхождение документа определяется исходя из протокола, хоста и номера порта для URL-адреса, с которого был загружен документ. Документы, загружаемые с других веб-серверов, имеют другое происхождение. Документы, загруженные с разных портов одного и того же хоста, также имеют другое происхождение. Наконец, документы, загруженные по протоколу HTTP, по происхождению отличаются от документов, загруженных по протоколу HTTPS, даже если загружены с одного и того же веб-сервера.

Важно понимать, что происхождение самого сценария не имеет никакого отношения к политике общего происхождения: значение имеет происхождение документа, в который встраивается сценарий. Предположим, что сценарий из домена A включается (с помощью атрибута `src` тега `<script>`) в веб-страницу из домена B. Этот сценарий будет иметь полный доступ ко всему содержимому этого документа. Если этот сценарий откроет второе окно и загрузит в него документ из домена B, он также будет иметь полный доступ к содержимому этого второго документа. Но если сценарий откроет третье окно и загрузит в него документ из домена C (или даже из домена A), в дело вступит политика общего происхождения и ограничит сценарий в доступе к этому документу.

Политика общего происхождения на самом деле применяется не ко всем свойствам всех объектов в окне, имеющем другое происхождение, но она применяется ко многим из них, в частности, практически ко всем свойствам объекта `Document` (подробности см. в главе 15). Кроме того, разные производители броузеров реализуют эту политику немного по-разному. (Например, броузер Firefox 1.0 допускает вызов метода `history.back()` из другого окна, а IE 6 – нет.) В любом случае можно считать, что любое окно, содержащее документ, полученный с другого сервера, для ваших сценариев закрыто. Если сценарий открыл такое окно, он может закрыть его, но он никаким способом не может «заглянуть внутрь» окна.

Политика общего происхождения необходима для того, чтобы не допустить кражу внутренней информации. Без этого ограничения злонамеренный сценарий (возможно, загруженный в броузер, расположенный в защищенной брандмауэром корпоративной сети) мог бы открыть пустое окно, в надежде обмануть пользователя и заставить его задействовать это окно для поиска файлов в локальной сети. После этого злонамеренный сценарий мог бы прочитать содержимое этого окна и отправить его обратно на свой сервер. Политика общего происхождения предотвращает возможность возникновения такого рода ситуаций.

Тем не менее в других ситуациях политика общего происхождения оказывается слишком строгой. Это создает особые проблемы для крупных веб-сайтов, на которых может функционировать несколько серверов. Например, сценарий с сервера *home.example.com* мог бы на вполне законных основаниях читать свойства документа, загруженного с *developer.example.com*, а сценариям с *orders.example.com* может потребоваться прочитать свойства из документов с *catalog.example.com*. Чтобы поддерживать такие крупные веб-сайты, можно использовать свойство `domain`: объекта `Document`. По умолчанию свойство `domain` содержит имя

сервера, с которого был загружен документ. Это свойство можно установить только равным строке, являющейся допустимым доменным суффиксом первоначального значения. Другими словами, если значение `domain` первоначально было равно строке `"home.example.com"`, то можно установить его равным `"example.com"`, но не `"home.example"` или `"ample.com"`. Кроме того, значение свойства `domain` должно содержать, по крайней мере, одну точку, чтобы его нельзя было установить равным `"com"` или другому имени домена верхнего уровня.

Если два окна (или фрейма) содержат сценарии, установившие одинаковые значения свойства `domain`, политика общего происхождения для этих двух окон ослабляется, и каждое из окон может читать значения свойств другого окна. Например, взаимодействующие сценарии в документах, загруженных с серверов *orders.example.com* и *catalog.example.com*, могут установить свойства `document.domain` равными `"example.com"`, тем самым указывая на общность происхождения документов и разрешая каждому из документов читать свойства другого.

13.8.3. Взаимодействие с модулями расширения и элементами управления ActiveX

Несмотря на то что в ядре языка JavaScript и базовой объектной модели на стороне клиента отсутствуют средства для работы с сетевым окружением и файловой системой, необходимые в основном для злонамеренного программного кода, тем не менее ситуация не такая простая, как кажется на первый взгляд. В большинстве броузеров JavaScript-код используется как «механизм исполнения» других программных компонентов, таких как элементы управления ActiveX в IE и модули расширения в других броузерах. Тем самым в распоряжении у сценариев на стороне клиента оказываются мощные средства. В главе 20 представлены примеры, в которых для организации взаимодействия по протоколу HTTP применяется элемент управления ActiveX, а в главах 19 и 22 для хранения информации на стороне клиента и отображения улучшенной графики используются модули расширения Java и Flash.

Вопросы безопасности приобретают особый смысл, поскольку элементы управления ActiveX и Java-апплеты могут иметь, например, низкоуровневый доступ к сетевым возможностям. Защитная «песочница» для Java не позволяет апплетам взаимодействовать с серверами, отличными от того, откуда получен апплет; тем самым закрывается брешь в системе безопасности. Но остается основная проблема: если модуль расширения может управляться из сценария, необходимо полное доверие не только системе безопасности броузера, но и системе безопасности самого модуля расширения. На практике модули расширения Java и Flash, похоже, не имеют проблем с безопасностью и не вызывают появление этих проблем в клиентском JavaScript-коде. Однако элементы управления ActiveX имеют более пестрое прошлое. Броузер IE обладает возможностью доступа из сценариев к самым разным элементам управления ActiveX, которые являются частью операционной системы Windows и которые раньше уже были источниками проблем безопасности. Однако к моменту написания этих строк эти проблемы были решены.

13.8.4. Межсайтовый скриптинг

Термин *межсайтовый скриптинг* (*cross-site scripting*), или XSS, относится к области компьютерной уязвимости, когда атакующий внедряет HTML-теги или сце-

нарии в документы на уязвимом веб-сайте. Организация защиты от XSS-атак – обычное дело для веб-разработчиков, занимающихся созданием серверных сценариев. Однако программисты, разрабатывающие клиентские JavaScript-сценарии, также должны знать о XSS-атаках и предпринимать меры защиты от них.

Веб-страница считается уязвимой для XSS-атак, если она динамически создает содержимое документа на основе пользовательских данных, не прошедших предварительную обработку по удалению встроенного HTML-кода. В качестве тривиального примера рассмотрим следующую веб-страницу, которая использует JavaScript-сценарий, чтобы приветствовать пользователя по имени:

```
<script>
var name = decodeURIComponent(window.location.search.substring(6)) || "";
document.write("Привет " + name);
</script>
```

Во второй строке сценария вызывается метод `window.location.search.substring`, с помощью которого извлекается часть адресной строки, начинающаяся с символа `?`. Затем с помощью метода `document.write()` добавляется динамически сгенерированное содержимое документа. Этот сценарий предполагает, что обращение к веб-странице будет производиться с помощью примерно такого URL-адреса:

```
http://www.example.com/greet.html?name=Давид
```

В этом случае будет выведен текст «Привет Давид». Но что произойдет, если страница будет запрошена с использованием следующего URL-адреса:

```
http://www.example.com/greet.html?name=%3Cscript%3Ealert('Давид')%3C/script%3E
```

С таким содержимым URL-адреса сценарий динамически сгенерирует другой сценарий (коды `%3C` и `%3E` – это угловые скобки)! В данном случае вставленный сценарий просто отобразит диалоговое окно, которое не представляет никакой опасности. Но представьте себе такой случай:

```
http://siteA/greet.html?name=%3Cscript src=siteB/evil.js%3E%3C/script%3E
```

Межсайтовый скриптинг потому так и называется, что в атаке участвует более одного сайта. Сайт B (или даже сайт C) включает специально сконструированную ссылку (подобную только что показанной) на сайт A, в которой содержится сценарий с сайта B. Сценарий *evil.js* размещается на сайте злоумышленника B, но теперь этот сценарий оказывается внедренным в сайт A и может делать все, что ему заблагорассудится с содержимым сайта A. Он может стереть страницу или вызвать другие нарушения в работе сайта (например, отказать в обслуживании, о чем рассказывается в следующем разделе). Это может отрицательно сказаться на посетителях сайта A. Гораздо опаснее, что такой злонамеренный сценарий может прочитать содержимое cookies, хранящихся на сайте A (возможно содержащих учетные номера или другие персональные сведения), и отправить эти данные обратно на сайт B. Внедренный сценарий может даже отслеживать нажатия клавиш и отправлять эти данные на сайт B.

Универсальный способ предотвращения XSS-атак заключается в удалении HTML-тегов из всех данных сомнительного происхождения, прежде чем использовать их для динамического создания содержимого документа. Чтобы исправить эту проблему в показанном ранее файле *greet.html*, нужно добавить следующую строку в сценарий, которая призвана удалять угловые скобки, окружающие тег `<script>`:

```
name = name.replace(/</g, "&lt;").replace(/>/g, "&gt;");
```

Межсайтовый скриптинг представляет собой уязвимость, глубоко уходящую корнями в архитектуру Всемирной паутины. Необходимо осознавать всю глубину этой уязвимости, но дальнейшее ее обсуждение далеко выходит за рамки темы данной книги. В Интернете есть немало ресурсов, которые помогут вам организовать защиту от атак подобного рода. Наиболее важный из них принадлежит группе компьютерной «скорой помощи» CERT Advisory: *http://www.cert.org/advisories/CA-2000-02.html*.

13.8.5. Атаки типа отказа в обслуживании

Политика общего происхождения и другие меры безопасности прекрасно защищают данные клиента от посягательств со стороны злонамеренного программного кода, но не могут предотвратить атак типа отказа в обслуживании. Если вы посетите злонамеренный веб-сайт и ваш броузер получит JavaScript-сценарий, в котором в бесконечном цикле вызывается метод `alert()`, выводящий диалоговое окно, вам придется использовать, например, команду `kill` в операционной системе Unix или диспетчер задач в Windows, чтобы завершить работу броузера.

Злонамеренный сайт может также попытаться загрузить центральный процессор компьютера бесконечным циклом с бессмысленными вычислениями. Некоторые типы броузеров (такие как Firefox) автоматически определяют наличие циклов с продолжительным временем работы и предоставляют пользователю возможность их прерывать. Это защищает от случайного зацикливания сценария, но злонамеренный программный код для обхода этой защиты может использовать прием на основе метода `window.setInterval()`. Подобная атака нагружает систему клиента, вызывая огромный расход памяти.

Универсального способа предотвращения таких атак в веб-броузерах не существует. На самом деле это не является общей проблемой Всемирной паутины, т. к. никаких данных на злонамеренный сайт в этом случае не передается!

13.9. Другие реализации JavaScript во Всемирной паутине

В дополнение к клиентскому языку JavaScript имеются и другие реализации языка JavaScript, имеющие отношение ко Всемирной паутине. В данной книге эти реализации не обсуждаются, но вам следует знать об их существовании, чтобы не путать с клиентским JavaScript:

Пользовательские сценарии

Пользовательские сценарии – это новейшее достижение, позволяющее пользователю добавлять сценарии к HTML-документам, прежде чем они будут отображены броузером. После этого веб-страница получает возможность управляться не только ее автором, но и посетителем веб-сайта. Самым известным примером пользовательского сценария является расширение броузера Firefox – Greasemonkey (*http://greasemonkey.mozdev.org*). Программное окружение, предоставляемое пользовательским сценариям, похоже, но не идентично клиентскому программному окружению. В этой книге не рассказывает-

ся о том, как написан пользовательский сценарий Greasemonkey, но изучение принципов клиентского JavaScript-программирования можно считать предпосылкой к изучению пользовательского JavaScript-программирования.

SVG

SVG (Scalable Vector Graphics – масштабируемая векторная графика) – это основанный на XML графический формат, допускающий внедрение JavaScript-сценариев. Как мы выяснили, клиентский JavaScript-код может взаимодействовать с HTML-документом, в который он внедрен. Аналогичным образом JavaScript-код, встроенный в SVG-файл, может взаимодействовать с XML-элементами этого документа. Материал, излагаемый в главах 15 и 17, имеет некоторое отношение и к SVG, но его недостаточно, поскольку объектная модель SVG-документа несколько отличается от объектной модели HTML-документа.

Спецификацию формата SVG можно найти на сайте *http://www.w3.org/TR/SVG*. В приложении B к этой спецификации находится определение DOM SVG. В главе 22 есть пример клиентского JavaScript-кода, встроенного в HTML-документ и создающего SVG-документ внутри HTML-документа. Поскольку JavaScript-код находится вне SVG-документа, это пример обычного клиентского JavaScript-кода, а не JavaScript-кода, встроенного в SVG.

XUL

XUL (XML User interface Language) – это основанный на грамматике XML язык, предназначенный для описания пользовательских интерфейсов. Графический интерфейс пользователя веб-броузера Firefox создан на основе XUL-документов. Подобно SVG, грамматика XUL может использоваться в JavaScript-сценариях. Как и в случае с SVG, материал, излагаемый в главах 15 и 17, имеет некоторое отношение и к XUL, однако JavaScript-код в XUL-документах имеет доступ к совсем другим объектам и прикладным интерфейсам, являясь субъектом иной модели безопасности, нежели клиентский JavaScript-код. Подробнее о XUL можно узнать на сайтах *http://www.mozilla.org/projects/xul* и *http://www.xulplanet.com*.

ActionScript

ActionScript – это язык программирования, подобный JavaScript (он следует все той же спецификации ECMAScript, но развивался в направлении объектно-ориентированного подхода) и используемый в анимационных Flash-роликах. Большая часть материала по основам JavaScript из первой части этой книги пригодна для изучения ActionScript-программирования. Формат Flash никакого отношения не имеет ни к XML, ни к HTML, а прикладные интерфейсы Flash вообще никак не связаны с темой этой книги. Однако в главах 19, 22 и 23 имеются примеры, демонстрирующие, как с помощью клиентского JavaScript-кода можно управлять Flash-роликами. В этих примерах можно найти маленькие фрагменты ActionScript-кода, но основное внимание в них уделяется использованию обычного клиентского JavaScript-кода для взаимодействия с ActionScript-кодом.

14

Работа с окнами броузера

В главе 13 был описан объект `Window` и отмечена центральная роль, которую этот объект играет в клиентском JavaScript-коде. Мы видели, что объект `Window` является глобальным объектом для клиентских JavaScript-программ. В этой главе будут рассмотрены свойства и методы объекта `Window`, позволяющие управлять броузером, его окнами и фреймами.

Здесь рассказывается о том, как:

- Зарегистрировать JavaScript-код для однократного или многократного исполнения в будущем
- Получить URL-адрес документа, отображаемого в окне, и выделить аргументы запроса из этого URL-адреса
- Заставить броузер загрузить и отобразить новый документ
- Сообщить броузеру о необходимости вернуться на предыдущую или следующую страницу из списка ранее посещавшихся страниц и управлять другими функциями броузера, такими как печать документа
- Открывать новые окна броузера, манипулировать ими и закрывать их
- Выводить простейшие диалоговые окна
- Определять тип броузера, в котором идет исполнение JavaScript-кода, и получать другие сведения о клиентском программном окружении
- Выводить произвольный текст в строке состояния окна броузера
- Обрабатывать неперехваченные ошибки, возникшие в окне броузера
- Писать JavaScript-код, призванный взаимодействовать с несколькими окнами и фреймами

Следует отметить, что в этой главе много говорится об окнах броузера, но ничего о *содержимом*, отображаемом в этих окнах. В самом начале развития JavaScript возможности взаимодействия с содержимым документа были весьма ограниченными, а приемы работы с окнами, описываемые в этой главе, были достаточно новыми и необычными. Сегодня, когда существует возможность в полной мере

управлять содержимым документов (см. главу 15), тема программирования броузера уже не кажется такой захватывающей. Кроме того, некоторые приемы, демонстрируемые в этой главе, работают уже не так, как раньше, из-за появившихся ограничений в области безопасности. Другие приемы по-прежнему работают, но их востребованность у веб-дизайнеров снизилась, потому они практически вышли из употребления.

Хотя на сегодняшний день эта глава в значительной степени утратила свою актуальность, излагаемый здесь материал еще может быть востребован, и я не рекомендовал бы пропускать ее. Глава организована так, что наиболее важные сведения находятся в начале главы. Ближе к концу описываются менее важные или редко используемые приемы. Лишь один важный и сложный раздел, в котором описываются приемы организации взаимодействия JavaScript-кода с несколькими окнами и фреймами, приводится в конце главы, а сама глава заканчивается полезным примером.

14.1. Таймеры

Одной из важнейших характеристик любого программного окружения является возможность запланировать исполнение программного кода в некоторый момент времени в будущем. Ядро языка JavaScript не предоставляет такой возможности, но в клиентском языке JavaScript такая возможность предусмотрена в виде глобальных функций `setTimeout()`, `clearTimeout()`, `setInterval()` и `clearInterval()`. Хотя в действительности эти функции ничего не делают с объектом `Window`, они описываются в этой главе, потому что объект `Window` является глобальным объектом, а данные функции являются методами этого объекта.

Метод `setTimeout()` объекта `Window` планирует запуск функции через определенное число миллисекунд. Метод `setTimeout()` возвращает значение, которое может быть передано методу `clearTimeout()`, позволяющему отменить запланированный ранее запуск функции.

Метод `setInterval()` похож на `setTimeout()`, за исключением того, что он автоматически заново планирует повторное исполнение. Подобно `setTimeout()`, метод `setInterval()` возвращает значение, которое может быть передано методу `clearInterval()`, позволяющему отменить запланированный запуск функции.

Методам `setTimeout()` и `setInterval()` в виде первого аргумента предпочтительнее передавать функцию, но допускается предавать строку JavaScript-кода. В этом случае программный код будет исполнен (один раз или несколько) через заданный интервал времени. В старых броузерах, таких как IE 4, возможность передачи функций не поддерживается, потому необходимо передавать методам непосредственно JavaScript-код в виде строки.

Методы `setTimeout()` и `setInterval()` могут использоваться в самых разных ситуациях. Если необходимо отобразить всплывающую подсказку, когда пользователь задерживает указатель мыши на некотором элементе документа на полсекунды или дольше, можно запланировать вывод подсказки с помощью метода `setTimeout()`. Если указатель мыши перемещается дальше без задержки, можно отменить вывод подсказки с помощью метода `clearTimeout()`. Порядок использования метода `setTimeout()` будет продемонстрирован позднее в примере 14.7. Всякий раз, когда возникает необходимость в выполнении анимации того или

иного рода, обычно используется метод setInterval(), с помощью которого планируется периодический запуск программного кода, реализующего анимацию. Этот прием демонстрируется в примерах 14.4 и 14.6.

Один интересный способ регистрации функции, реализуемый методом setTimeout(), заключается в том, чтобы запланировать ее запуск через 0 миллисекунд. При этом функция вызывается не сразу, а «как только такая возможность появляется». На практике метод setTimeout() планирует запуск функции лишь после того, как будут обработаны все события, ожидающие обработки, и завершится обновление текущего состояния документа. Даже обработчики событий (см. главу 17), которые пытаются получить или модифицировать содержимое документа (см. главу 15), иногда вынуждены использовать этот трюк, чтобы отложить исполнение своего программного кода на тот момент, когда состояние документа стабилизируется.

Справочную информацию по этим функциям вы найдете в четвертой части книги в разделе, в котором описывается объект Window.

14.2. Объекты Location и History

В этом разделе обсуждаются объекты Location и History окна. Эти объекты предоставляют доступ к URL-адресу текущего документа и позволяют загрузить новый документ либо заставляют броузер вернуться назад (или перейти вперед) к ранее просматривавшимся документам.

14.2.1. Анализ URL

Свойство location окна (или фрейма) является ссылкой на объект Location и представляет URL-адрес документа, отображаемого в данный момент в текущем окне. Свойство href объекта Location – это строка, содержащая полный текст URL-адреса. Метод toString() объекта Location возвращает значение свойства href, поэтому вместо конструкции location.href можно писать просто location.

Другие свойства этого объекта, такие как protocol, host, pathname и search, определяют отдельные части URL-адреса (полное описание объекта Location приводится в четвертой части книги).

Свойство search объекта Location представляет особый интерес. Оно содержит часть URL-адреса, следующую за вопросительным знаком, если таковая имеется, включая сам знак вопроса. Обычно эта часть URL-адреса является строкой запроса. Вопросительный знак в URL-адресе – это средство для встраивания аргументов в URL-адрес. Хотя эти аргументы обычно предназначены для CGI-сценариев, исполняющихся на сервере, нет причины, по которой они не могли бы также использоваться в страницах, содержащих JavaScript-код. В примере 14.1 показано определение универсальной функции getArgs(), позволяющей извлекать аргументы из свойства search URL-адреса.

Пример 14.1. Извлечение аргументов из URL-адреса

```
/*
 * Эта функция выделяет в URL-адресе разделенные амперсандами пары аргументов
 * name=value из строки запроса. Она сохраняет эти пары в свойствах объекта
 * и возвращает этот объект. Порядок использования:
```

```
 *
 * var args = getArgs();  // Извлечь аргументы из URL
 * var q = args.q || "";  // Использовать аргумент, если определен,
 *                        // или значение по умолчанию
 * var n = args.n ? parseInt(args.n) : 10;
 */
function getArgs( ) {
    var args = new Object();
    var query = location.search.substring(1);     // Получить строку запроса
    var pairs = query.split("&");                 // Разбить по амперсандам
    for(var i = 0; i < pairs.length; i++) {
        var pos = pairs[i].indexOf('=');          // Отыскать пару "name=value"
        if (pos == -1) continue;                  // Не найдено - пропустить
        var argname = pairs[i].substring(0,pos); // Извлечь имя
        var value = pairs[i].substring(pos+1);   // Извлечь значение
        value = decodeURIComponent(value);        // Преобразовать, если нужно
        args[argname] = value;                    // Сохранить в виде свойства
    }
    return args;                                  // Вернуть объект
}
```

14.2.2. Загрузка нового документа

Даже при том, что свойство `location` объекта `Window` ссылается на объект `Location`, существует возможность присвоить этому свойству строковое значение. В этом случае броузер интерпретирует строку как URL-адрес и предпринимает попытку загрузить и отобразить документ с этим URL-адресом. Например, присвоить строку URL-адреса свойству `location` можно следующим образом:

```
// Если броузер не поддерживает функцию Document.getElementById, выполнить
// переход к статической странице, которая не использует эту функцию.
if (!document.getElementById) location = "staticpage.html";
```

Примечательно, что строка URL-адреса, записанная в свойство `location` в этом примере, представляет относительный адрес. Относительные URL-адреса интерпретируются относительно страницы, в которой они появляются, точно так же, как если бы они использовались в гиперссылке.

В примере 14.7, который приводится в конце этой главы, для загрузки нового документа также используется свойство `location`.

Интересно, что в объекте `Window` отсутствует метод, с помощью которого можно было бы заставить броузер загрузить и отобразить новый документ. Исторически сложилось так, что для загрузки новых страниц поддерживается только прием с присваиванием строки URL-адреса свойству `location` окна. Однако объект `Location` содержит два метода, предназначенные для аналогичных целей. Метод `reload()` заново загружает текущую отображаемую страницу с веб-сервера. Метод `replace()` загружает и отображает страницу по заданному URL-адресу. Однако вызов этого метода для данного URL-адреса отличается от присваивания этого URL свойству `location` окна. Когда вызывается `replace()`, указанный URL-адрес заменяет текущий URL-адрес в списке истории просмотра, а не создает новую запись. Следовательно, если для перекрытия одного документа другим вызывается метод `replace()`, кнопка Назад не вернет пользователя обратно к исход-

ному документу, как это произойдет при загрузке нового документа путем присваивания URL-адреса свойству location. Для сайтов, использующих фреймы и отображающих много временных страниц (возможно, сгенерированных серверными сценариями), применение метода replace() часто оказывается полезным, поскольку временные страницы не сохраняются в списке истории и от кнопки Назад пользователь может добиться больше толка.

И наконец, не путайте свойство location объекта Window, ссылающееся на объект Location, со свойством location объекта Document, которое просто представляет собой доступную только для чтения строку без каких-либо особенностей, присущих объекту Location. Свойство document.location — это синоним свойства document.URL, которое является более предпочтительным именем для этого свойства (т. к. позволяет избежать потенциальной путаницы). В большинстве случаев document.location совпадает с location.href. Однако когда происходит перенаправление на стороне сервера, document.location содержит загруженный URL-адрес, а location.href — изначально запрошенный URL-адрес.

14.2.3. Объект History

Свойство history объекта Window ссылается на объект History данного окна. Объект History изначально разрабатывался для ведения истории просмотра страниц в окне в виде массива недавно открывавшихся URL-адресов. Однако этот замысел оказался неудачным; по серьезным причинам, относящимся к безопасности и секретности, сценарию почти никогда нельзя предоставлять доступ к списку веб-сайтов, ранее посещенных пользователем. Поэтому реально элементы массива объекта History практически никогда недоступны для сценариев.

Хотя элементы массива недоступны, объект History поддерживает три метода. Методы back() и forward() позволяют перемещаться вперед и назад по истории просмотра данного окна (или фрейма), заменяя текущий отображаемый документ ранее просматривавшимся. Аналогичные события происходят, когда пользователь щелкает в броузере на кнопках Назад и Вперед. Третий метод, go(), принимает целочисленный аргумент и пропускает заданное число страниц вперед (если аргумент положительный) или назад (отрицательный) в списке истории. Использование методов back() и forward() объекта History демонстрируется в примере 14.7 в конце этой главы.

Броузеры Netscape и Mozilla поддерживают также методы back() и forward() в самом объекте Window. Эти непереносимые методы выполняют те же действия, что и кнопки броузера Назад и Вперед. При использовании фреймов метод window.back() по своему действию может отличаться от метода history.back().

14.3. Объекты Window, Screen и Navigator

Иногда сценариям бывает необходимо получить информацию об окне, рабочем столе или броузере, в котором исполняется сценарий. В этом разделе описываются свойства объектов Window, Screen и Navigator, позволяющие определить такие параметры, как размер окна броузера, размер рабочего стола и версию веб-броузера. Эта информация дает возможность подстроить поведение сценария под существующее окружение.

14.3.1. Геометрия окна

Большинство броузеров (за исключением Internet Explorer) поддерживают простой набор свойств объекта `Window`, с помощью которых можно получить сведения о размерах окна и его положении:

```
// Полные размеры окна броузера на рабочем столе
var windowWidth = window.outerWidth;
var windowHeight = window.outerHeight;

// Положение окна броузера на рабочем столе
var windowX = window.screenX
var windowY = window.screenY

// Размеры клиентской области окна, в которой отображается содержимое
// документа. Это размер окна минус высота строки меню,
// панелей инструментов, полос прокрутки и т.п.
var viewportWidth = window.innerWidth;
var viewportHeight = window.innerHeight;

// Эти значения определяют величину вертикального и горизонтального смещения
// Используются для перехода между координатами документа и координатами окна
// Эти значения указывают, какая часть документа находится в верхнем левом углу экрана
var horizontalScroll = window.pageXOffset;
var verticalScroll = window.pageYOffset;
```

Обратите внимание: эти свойства доступны только для чтения. Методы, которые позволяют перемещать окно, изменять его размеры или прокручивать содержимое, описаны в этой главе далее. Кроме того, следует отметить, что имеется несколько систем координат, о существовании которых знать совершенно необходимо. *Экранные координаты* определяют положение окна броузера на рабочем столе и измеряются относительно верхнего левого угла рабочего стола. *Оконные координаты* определяют положение внутри клиентской области окна броузера и измеряются относительно верхнего левого угла клиентской области окна. *Координаты документа* определяют положение внутри HTML-документа и измеряются относительно верхнего левого угла документа. Если документ по размерам превышает клиентскую область окна (что случается достаточно часто), координаты в документе и оконные координаты не совпадают, и при переходе между этими системами координат необходимо учитывать величины смещений. Подробнее о системах координат рассказывается в главах 15 и 16.

Как уже упоминалось, свойства объекта `Window`, о которых только что говорилось, отсутствуют в Internet Explorer. По каким-то причинам свойства, описывающие геометрию окна в IE, определены как свойства тега `<body>` HTML-документа. Хуже того, когда документ с объявлением `<!DOCTYPE>` отображается в IE 6, эти свойства перемещаются в объект `document.documentElement`, а не в `document.body`.

Подробности вы найдете в примере 14.2. Здесь приводится объявление объекта `Geometry` с методами, которые позволяют определять размер клиентской области окна, величину смещения и экранные координаты переносимым способом.

Пример 14.2. Переносимый способ определения геометрии окна

```
/**
 * Geometry.js: переносимые функции определения геометрии окна и документа
 *
```

```
 * Этот модуль определяет функции получения геометрических характеристик
 * окна и документа
 *
 * getWindowX/Y()            : возвращают положение окна на экране
 * getViewportWidth/Height() : возвращают размеры клиентской области окна
 * getDocumentWidth/Height() : возвращают размеры документа
 * getHorizontalScroll()     : возвращает смещение по горизонтали
 * getVerticalScroll()       : возвращает смещение по вертикали
 *
 * Обратите внимание: не существует переносимого способа определить общие
 * размеры окна броузера, поэтому отсутствуют функции getWindowWidth/Height()
 *
 * ВАЖНО: Этот модуль должен включаться в тег <body> документа, а не в тег <head>
 */
var Geometry = {};

if (window.screenLeft === undefined) { // Для IE и других
    Geometry.getWindowX = function() { return window.screenLeft; };
    Geometry.getWindowY = function() { return window.screenTop; };
}
else if (window.screenX) { // Для Firefox и других
    Geometry.getWindowX = function() { return window.screenX; };
    Geometry.getWindowY = function() { return window.screenY; };
}

if (window.innerWidth) { // Все броузеры, кроме IE
    Geometry.getViewportWidth = function() { return window.innerWidth; };
    Geometry.getViewportHeight = function() { return window.innerHeight; };
    Geometry.getHorizontalScroll = function() { return window.pageXOffset; };
    Geometry.getVerticalScroll = function() { return window.pageYOffset; };
}
else if (document.documentElement && document.documentElement.clientWidth) {
    // Эти функции предназначены для IE 6 и документов с объявлением DOCTYPE
    Geometry.getViewportWidth =
        function() { return document.documentElement.clientWidth; };
    Geometry.getViewportHeight =
        function() { return document.documentElement.clientHeight; };
    Geometry.getHorizontalScroll =
        function() { return document.documentElement.scrollLeft; };
    Geometry.getVerticalScroll =
        function() { return document.documentElement.scrollTop; };
}
else if (document.body.clientWidth) {
    // Эти функции предназначены для IE4, IE5 и IE6 без объявления DOCTYPE
    Geometry.getViewportWidth =
        function() { return document.body.clientWidth; };
    Geometry.getViewportHeight =
        function() { return document.body.clientHeight; };
    Geometry.getHorizontalScroll =
        function() { return document.body.scrollLeft; };
    Geometry.getVerticalScroll =
        function() { return document.body.scrollTop; };
}

// Следующие функции возвращают размеры документа.
```

```
// Они не имеют отношения к окну, но бывает удобно иметь их.
if (document.documentElement && document.documentElement.scrollWidth) {
    Geometry.getDocumentWidth =
        function() { return document.documentElement.scrollWidth; };
    Geometry.getDocumentHeight =
        function() { return document.documentElement.scrollHeight; };
}
else if (document.body.scrollWidth) {
    Geometry.getDocumentWidth =
        function() { return document.body.scrollWidth; };
    Geometry.getDocumentHeight =
        function() { return document.body.scrollHeight; };
}
```

14.3.2. Объект Screen

Свойство `screen` объекта `Window` ссылается на объект `Screen`, предоставляющий информацию о размере экрана пользователя и доступном количестве цветов. Свойства `width` и `height` задают размер экрана в пикселах. Они могут использоваться, например, для выбора размеров изображений, включаемых в документ.

Свойства `availWidth` и `availHeight` задают реально доступный размер экрана; из них исключается пространство, требуемое для таких графических элементов, как панель задач. В броузере Firefox и подобных ему (но не в IE) у объекта `Screen` имеются еще два свойства – `availLeft` и `availTop`. Эти свойства определяют координаты первой доступной позиции на экране. Если, например, создается сценарий, который открывает новые окна броузера (о чем рассказывается далее в этой главе), эти свойства могут использоваться для позиционирования окна в центр рабочего стола.

Работу с объектом `Screen` иллюстрирует пример 14.4 далее в этой главе.

14.3.3. Объект Navigator

Свойство `navigator` объекта `Window` ссылается на объект `Navigator`, содержащий общую информацию о веб-броузере, такую как версия и список отображаемых форматов данных. Объект `Navigator` назван «в честь» Netscape Navigator, но он также поддерживается в Internet Explorer. (Кроме того, IE поддерживает свойство `clientInformation` как нейтральный синоним для `navigator`. К сожалению, другие броузеры свойство с таким именем не поддерживают.)

В прошлом объект `Navigator` обычно использовался сценариями для определения типа броузера – Internet Explorer или Netscape. Однако такой подход к определению типа броузера сопряжен с определенными проблемами, т. к. требует постоянного обновления с появлением новых броузеров или новых версий существующих броузеров. Ныне более предпочтительным считается метод на основе *проверки функциональных возможностей*. Вместо того чтобы делать какие-либо предположения о броузерах и их возможностях, гораздо проще прямо проверить наличие требуемой функциональной возможности (например, метода). Например, в следующем примере демонстрируется, как выполняется проверка функциональных возможностей при регистрации методов обработки событий (эта тема подробно рассматривается в главе 17).

```
if (window.addEventListener) {
    // Если метод addEventListener() поддерживается, использовать его.
    // Это случай совместимых со стандартами броузеров, таких как
    // Netscape, Mozilla и Firefox.
}
else if (window.attachEvent) {
    // Иначе, если существует метод attachEvent(), использовать его.
    // Это относится к IE и другим имитирующим его нестандартным броузерам.
}
else {
    // Иначе ни один из методов недоступен.
    // Это характерно для старых броузеров, не поддерживающих DHTML.
}
```

Однако иногда определение типа броузера может представлять определенную ценность. Один из таких случаев – возможность обойти ошибку, свойственную определенному типу броузера определенной версии. Объект `Navigator` позволяет решать такие задачи.

Объект `Navigator` имеет пять свойств, предоставляющих информацию о версии работающего броузера:

`appName`

Название веб-броузера. В IE это строка `"Microsoft Internet Explorer"`, в Firefox и других броузерах, в основе которых лежит программный код Netscape (таких как Mozilla или собственно Netscape), значением этого свойства является строка `"Netscape"`.

`appVersion`

Номер версии и/или другая информация о версии броузера. Обратите внимание: этот номер следует рассматривать как внутренний номер версии, поскольку он не всегда соответствует номеру, отображаемому для пользователя. Так, Netscape 6 и последовавшие за ним версии Mozilla и Firefox сообщают о себе номер версии 5.0. Кроме того, все версии IE от 4 до 6 сообщают о себе номер версии 4.0, что указывает на совместимость с базовой функциональностью броузеров 4-го поколения.

`userAgent`

Строка, которую броузер посылает в HTTP-заголовке `USER-AGENT`. Это свойство обычно содержит всю ту информацию, которая содержится в свойствах `appName` и `appVersion`, а также может содержать дополнительные сведения. Однако формат представления этой информации не стандартизован, поэтому невозможно организовать разбор этой строки способом, не зависящим от типа броузера.

`appCodeName`

Кодовое имя броузера. Для Netscape используется кодовое имя «Mozilla». Для совместимости IE делает то же самое.

`platform`

Аппаратная платформа, на которой работает броузер. Это свойство было добавлено в JavaScript 1.2.

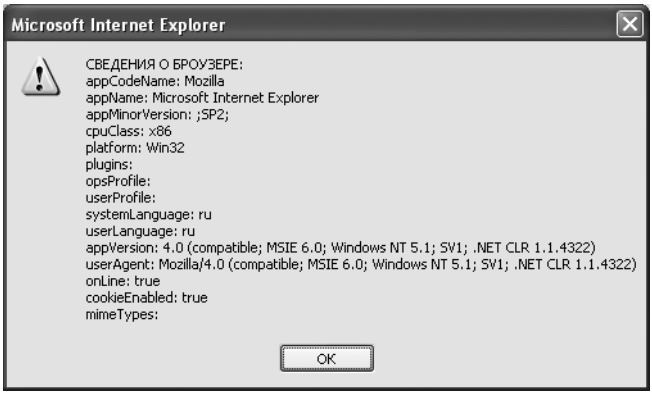

***Рис. 14.1.** Свойства объекта Navigator*

Следующие строки JavaScript-кода выводят значения всех свойств объекта `Navigator` в диалоговом окне:

```
var browser = "СВЕДЕНИЯ О БРОУЗЕРЕ:\n";
for(var propname in navigator) {
    browser += propname + ": " + navigator[propname] + "\n"
}
alert(browser);
```

Диалоговое окно, представленное на рис. 14.1, выводится при запуске этого сценария в IE 6.

Как видно из рис. 14.1, свойства объекта `Navigator` иногда содержат более сложную информацию, чем та, которая нас интересует. Например, обычно достаточно знать лишь первые цифры из свойства `appVersion`. Для извлечения из объекта `Navigator` только необходимой информации о броузере часто используются методы `parseInt()` и `String.indexOf()`. В примере 14.3 показан программный код, обрабатывающий свойства объекта `Navigator` и сохраняющий их в объекте с именем `browser`. С обработанными свойствами иметь дело проще, чем с исходными значениями свойств объекта `navigator`. Общий термин для такого кода – *анализатор клиента* (*client sniffer*), и в Интернете можно найти код более сложных и универсальных анализаторов. (Например, *http://www.mozilla.org/docs/web-developer/sniffer/browser_type.html*.) Однако для многих целей прекрасно работают и такие простые фрагменты кода.

Пример 14.3. Определение производителя броузера и номера версии

```
/**
 * browser.js: простейший анализатор клиента
 *
 * Этот модуль объявляет объект с именем "browser", пользоваться которым
 * гораздо проще, чем объектом "navigator".
 */
var browser = {
    version: parseInt(navigator.appVersion),
    isNetscape: navigator.appName.indexOf("Netscape") != -1,
```

```
    isMicrosoft: navigator.appName.indexOf("Microsoft") != -1
};
```

Прежде чем закончить этот раздел, необходимо сделать одно важное замечание: свойства объекта `Navigator` не могут служить основой для надежной идентификации броузера. Например, в Firefox 1.0 свойство `appName` имеет значение `"Netscape"`, а свойство `appVersion` начинается со значения 5.0. В броузере Safari, который не имеет ничего общего с линейкой броузеров проекта Mozilla, это свойство возвращает то же самое значение! В IE 6.0 свойство `appCodeName` имеет значение `"Mozilla"`, а свойство `appVersion` начинается со значения 4.0. Причина такого положения вещей кроется в следующем: в прошлом было создано так много программного кода, анализирующего тип броузера, что производители броузеров не могут позволить себе изменять значения этих свойств, поскольку это приведет к нарушению обратной совместимости. Кстати, это одна из причин, почему анализ типа броузера все больше выходит из употребления и все чаще используются методики на основе проверки функциональных возможностей.

14.4. Методы управления окнами

Объект `Window` определяет несколько методов, предназначенных для высокоуровневого управления самим окном. В следующих разделах рассматривается, как эти методы позволяют открывать и закрывать окна, управлять их положением и размером, запрашивать и передавать фокус ввода, прокручивать содержимое окна. Заканчивается раздел примером, демонстрирующим некоторые из этих возможностей.

14.4.1. Открытие окон

Открывать новые окна броузера можно методом `open()` объекта `Window`.

С помощью метода `window.open()` создаются всплывающие окна, содержащие рекламу, когда пользователь путешествует по Всемирной паутине. Из-за подобных злоупотреблений в большинстве веб-броузеров появились разнообразные системы блокирования всплывающих окон. Обычно вызов метода `open()` приводит к открытию нового окна[1], если это действие инициировано самим пользователем, например щелчком мыши на кнопке или ссылке. JavaScript-сценарий, который попытается открыть новое окно при первой загрузке (или выгрузке) страницы, наверняка потерпит неудачу.

Метод `open()` принимает четыре необязательных аргумента и возвращает объект `Window`, представляющий только что открытое окно. Первый аргумент `open()` – это URL-адрес документа, отображаемого в новом окне. Если этот аргумент отсутствует (либо равен `null` или пустой строке), окно будет пустым.

Второй аргумент `open()` – это имя окна. Как показано далее в данной главе, это имя может использоваться в качестве значения атрибута `target` тега `<form>` или `<a>`. Если указать имя уже существующего окна, `open()` просто вернет ссылку на существующее окно, не открывая нового.

---

[1] Или на выбор новой «вкладки» главного окна броузера, как это названо в Mozilla Firefox. – *Примеч. науч. ред.*

Третий необязательный аргумент open() – это список параметров, задающих размер и элементы графического пользовательского интерфейса окна. Если опустить этот аргумент, окно получает размер по умолчанию и полный набор графических элементов, включая меню, строку состояния, панель инструментов и т. д. Указав этот аргумент, можно явно задать размер окна и набор имеющихся в нем элементов управления. Например, маленькое окно с изменяемым размером, имеющее строку состояния, но не содержащее меню, панели инструментов и адресную строку, можно открыть посредством следующей строки JavaScript-кода:

```
var w = window.open("smallwin.html", "smallwin",
                    "width=400,height=350,status=yes,resizable=yes");
```

Обратите внимание: когда указывается третий аргумент, любые явно не заданные элементы управления отсутствуют. Полный набор доступных элементов и их имен приведен в описании метода Window.open() в четвертой части книги. По ряду причин, связанных с проблемами безопасности, броузеры накладывают ограничения на характеристики, которые можно передать методу. Так, например, невозможно открыть слишком маленькое окно или открыть его за пределами видимой области экрана; кроме того, некоторые броузеры не допускают возможности создания окон без строки состояния. Чем больше способов обмана пользователей придумают спамеры, мошенники и другие жители темной стороны Всемирной паутины, тем больше ограничений будет накладываться на метод open().

Указывать четвертый аргумент open() имеет смысл, только если второй аргумент представляет собой имя существующего окна. Этот аргумент – логическое значение, определяющее, должен ли URL-адрес, указанный в первом аргументе, заменить текущую запись в истории просмотра окна (true) или требуется создать новую запись (false). Последний вариант выбирается по умолчанию.

Возвращаемое методом open() значение является объектом Window, представляющим только что созданное окно. Этот объект в JavaScript-коде позволяет сослаться на новое окно так же, как исходный объект Window ссылается на окно, в котором работает ваш код. А как насчет обратной ситуации? Что если JavaScript-код в новом окне захочет обращаться к открывшему его окну? Свойство opener объекта Window ссылается на окно, из которого было открыто текущее окно. Если окно было создано пользователем, а не JavaScript-сценарием, значение свойства opener равно null.

14.4.2. Закрытие окон

Новое окно открывается при помощи метода open() и закрывается при помощи метода close(). Если мы создали объект Window, то закрыть его можно инструкцией:

```
w.close();
```

JavaScript-код, работающий внутри данного окна, может закрыть его так:

```
window.close();
```

Снова обратите внимание на явное использование идентификатора window для устранения неоднозначности между методом close() объекта Window и методом close() объекта Document.

Большинство броузеров разрешают программисту автоматически закрывать только те окна, которые были созданы его собственным JavaScript-кодом. Если

сценарий попытается закрыть любое другое окно, появится диалоговое окно с запросом к пользователю подтвердить (или отменить) закрытие окна. Эта предосторожность не дает неосмотрительным создателям сценариев писать код, закрывающий главное окно броузера пользователя.

Объект `Window` продолжает существовать и после закрытия представляемого им окна. Однако не следует использовать какие-либо его свойства или методы, исключая проверку свойства `closed`. Это свойство равно `true`, если окно было закрыто. Помните, что пользователь может закрывать любые окна в любое время, поэтому, чтобы избежать ошибок, полезно периодически проверять, открыто ли окно, с которым вы пытаетесь работать.

14.4.3. Геометрия окна

Объект `Window` определяет методы, с помощью которых можно перемещать окна и изменять их размеры. Обращение к этим методам обычно не приветствуется, поскольку считается, что пользователь должен иметь исключительный контроль над размерами и положением окон на рабочем столе. Современные броузеры обычно содержат параметр, с помощью которого можно запретить JavaScript-сценариям перемещать окна или изменять их размеры, поэтому всегда следует быть готовым к тому, что у значительного числа пользователей этот параметр активирован. Кроме того, чтобы воспрепятствовать исполнению злонамеренных сценариев в окнах небольшого размера или расположенных за пределами видимой области экрана, которые пользователь может не заметить, броузеры обычно ограничивают возможность перемещать окна за пределы экрана или делать их слишком маленькими. Если после всего сказанного у вас еще осталось желание перемещать окна или изменять их размеры, тогда продолжайте читать.

Метод `moveTo()` перемещает левый верхний угол окна в точку с указанными координатами. Похожим образом метод `moveBy()` перемещает окно на указанное количество пикселов влево или вправо, вверх или вниз. Методы `resizeTo()` и `resizeBy()` изменяют размер окна на абсолютное или относительное значение. Подробнее об этом говорится в четвертой части книги.

14.4.4. Фокус ввода и видимость

Методы `focus()` и `blur()` также предоставляют средства высокоуровневого управления окном. Вызов `focus()` запрашивает у системы фокус ввода для окна, а `blur()` освобождает фокус. Кроме того, метод `focus()` гарантирует, что окно будет видимым, перенося его в начало стека окон. Когда новое окно открывается с помощью метода `window.open()`, броузер автоматически создает окно в начале стека окон. Но если второй аргумент задает имя уже существующего окна, метод `open()` не делает автоматически окно видимым. Поэтому часто за вызовом `open()` следует вызов `focus()`.

14.4.5. Прокрутка

Объект `Window` содержит также методы, прокручивающие документ внутри окна или фрейма. Метод `scrollBy()` прокручивает документ на указанное количество пикселов влево или вправо, вверх или вниз, а метод `scrollTo()` – на абсолютную позицию. Он перемещает документ таким образом, что точка документа с ука-

занными координатами отображается в левом верхнем углу области документа в окне.

В современных броузерах HTML-элементы документа (см. главу 15) имеют такие свойства, как offsetLeft и offsetTop, которые содержат координаты X и Y элемента (в разделе 16.2.3 описываются методы, которые могут использоваться для определения координат любого элемента). После того как координаты элемента определены, с помощью метода scrollTo()можно прокрутить содержимое окна так, что любой конкретный элемент переместится в левый верхний угол окна.

Другой способ прокрутки заключается в обращении к методу focus() элемента документа (например, поля ввода или кнопки), в результате элементу передается фокус ввода. Как побочный эффект операции передачи фокуса ввода документ прокручивается так, чтобы элемент с фокусом ввода стал видимым. Обратите внимание: это не означает, что элемент обязательно переместится в левый верхний угол окна, метод лишь гарантирует, что элемент станет видимым.

Большинством современных броузеров поддерживается еще один удобный метод прокрутки: вызов метода scrollIntoView() для любого HTML-элемента делает этот элемент видимым. Данный метод пытается расположить указанный элемент как можно ближе к верхней границе окна, но это, конечно же, не относится к элементам, расположенным достаточно близко к концу документа. Метод scrollIntoView() реализован не так широко, как метод focus(), зато работает со всеми HTML-элементами, а не только с теми, которые способны принимать фокус ввода. Подробнее об этом методе можно прочитать в четвертой части книги.

Последний способ прокрутки окна из сценариев заключается в определении якорных элементов в виде тегов <a name=> в тех позициях, к которым может потребоваться прокрутить документ. После этого можно использовать имена якорных элементов для записи в свойство hash объекта Location. Например, если в документе имеется якорный элемент с именем «top» в начале документа, тогда вернуться к началу документа можно будет следующим образом:

```
window.location.hash = "#top";
```

Прием с использованием именованных якорных элементов расширяет возможности навигации в пределах документа. Кроме того, он делает позицию в документе видимой в адресной строке броузера, позволяет устанавливать закладки и возвращаться к предыдущей позиции с помощью кнопки Назад, что может быть весьма привлекательным.

В то же время загромождение списка истории просмотра именованными якорями, сгенерированными сценариями, в некоторых ситуациях можно рассматривать как досадную помеху. Чтобы прокрутить документ к именованному якорному элементу (в большинстве броузеров) без создания новой записи в списке истории, следует использовать метод Location.replace():

```
window.location.replace("#top");
```

14.4.6. Пример использования методов объекта Window

Пример 14.4 демонстрирует порядок использования методов open(), close() и moveTo() объекта Window, а также некоторые другие обсуждавшиеся нами приемы работы с окнами. В примере создается новое окно и затем с помощью метода

setInterval() **задаются интервалы повторяющихся вызовов функции, перемещающей это окно по экрану. Размер экрана определяется с помощью объекта** Screen **и потом на основе этих данных осуществляется отскок окна при достижении им любого края экрана.**

Пример 14.4. Создание и перемещение окна

```
<script>
var bounce = {
    x:0, y:0, w:200, h:200, // Положение окна и его размеры
    dx:5, dy:5,             // Скорость перемещения
    interval: 100,          // Частота обновления в миллисекундах
    win: null,              // Создаваемое окно
    timer: null,            // Возвращаемое значение метода setInterval()

    // Запуск анимации
    start: function() {
        // Вначале окно располагается в центре экрана
        bounce.x = (screen.width - bounce.w)/2;
        bounce.y = (screen.height - bounce.h)/2;

        // Создать окно, которое будет перемещаться по экрану
        // URL javascript: - простейший способ вывести короткий документ
        // Последний аргумент определяет размеры окна
        bounce.win = window.open('javascript:"<h1>ОТСКОК!</h1>"', "",
                                 "left=" + bounce.x + ",top=" + bounce.y +
                                 ",width=" + bounce.w + ",height=" +bounce.h+
                                 ",status=yes");

        // Использовать setInterval() для вызова метода nextFrame() через
        // каждый установленный интервал времени. Сохранить возвращаемое
        // значение, чтобы иметь возможность остановить анимацию
        // вызовом clearInterval().
        bounce.timer = setInterval(bounce.nextFrame, bounce.interval);
    },

    // Остановить анимацию
    stop: function() {
        clearInterval(bounce.timer);                    // Прервать работу таймера
        if (!bounce.win.closed) bounce.win.close();     // Закрыть окно
    },

    // Отобразить следующий кадр. Вызывается методом setInterval()
    nextFrame: function() {

        // Если пользователь закрыл окно – прекратить работу
        if (bounce.win.closed) {
            clearInterval(bounce.timer);
            return;
        }

        // Имитировать отскок, если была достигнута правая или левая граница
        if ((bounce.x+bounce.dx > (screen.availWidth - bounce.w)) ||
            (bounce.x+bounce.dx < 0)) bounce.dx = -bounce.dx;

        // Имитировать отскок, если была достигнута верхняя или нижняя граница
        if ((bounce.y+bounce.dy > (screen.availHeight - bounce.h)) ||
            (bounce.y+bounce.dy < 0)) bounce.dy = -bounce.dy;
```

```
        // Обновить координаты окна
        bounce.x += bounce.dx;
        bounce.y += bounce.dy;

        // Переместить окно в новую позицию
        bounce.win.moveTo(bounce.x,bounce.y);

        // Отобразить текущие координаты в строке состояния
        bounce.win.defaultStatus = "(" + bounce.x + "," + bounce.y + ")";
    }
}
</script>
<button onclick="bounce.start()">Старт</button>
<button onclick="bounce.stop()">Стоп</button>
```

14.5. Простые диалоговые окна

Объект `Window` обладает тремя методами для представления пользователю простейших диалоговых окон. Метод `alert()` выводит сообщение и ожидает, пока пользователь закроет диалоговое окно. Метод `confirm()` предлагает пользователю щелкнуть на кнопке OK или Отмена для подтверждения или отмены операции. Метод `prompt()` предлагает пользователю ввести строку.

Хотя эти методы вызова диалоговых окон чрезвычайно просты в использовании, правила хорошего тона требуют, чтобы они применялись как можно реже и только в случае насущной необходимости. Диалоговые окна, подобные этим, не являются распространенной парадигмой в веб-дизайне и в настоящее время применяются все реже и реже благодаря поддержке в веб-броузерах средств изменения содержимого самого документа. Большинство пользователей считают, что диалоговые окна, выводимые методами `alert()`, `confirm()` и `prompt()`, противоречат обычной практике. Единственный вариант, когда имеет смысл обращаться к этим методам, – это отладка. JavaScript-программисты часто вставляют вызов метода `alert()` в программный код, пытаясь диагностировать возникающие проблемы (альтернативный способ отладки представлен в примере 15.9).

Обратите внимание: текст, отображаемый в диалоговых окнах, – это обычный неформатированный текст. Его можно форматировать только пробелами, переводами строк и различными знаками пунктуации.

Некоторые броузеры отображают слово «JavaScript» в заголовке или верхнем левом углу всех диалоговых окон, создаваемых методами `alert()`, `confirm()` и `prompt()`. Хотя дизайнеров этот факт раздражает, его стоит рассматривать как особенность, а не как ошибку; слово «JavaScript» находится там, чтобы пользователю было ясно происхождение диалогового окна и помешать созданию кода «троянских коней», имитирующего системные диалоговые окна и обманом заставляющего пользователей вводить свои пароли и выполнять другие действия, которые им выполнять не следует.

Методы `confirm()` и `prompt()` являются *блокирующими*, т. е. они не возвращают управление, пока пользователь не закроет выводимые ими диалоговые окна.[1] Это значит, что когда выводится одно из этих окон, программный код прекращает

[1] Обычно такие окна называют модальными. – *Примеч. науч. ред.*

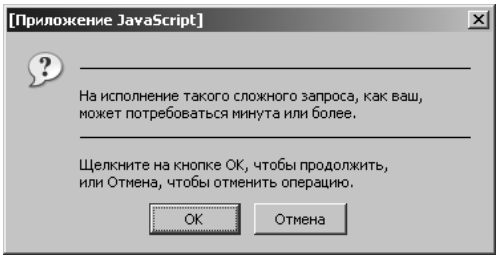

***Рис. 14.2.** Диалоговое окно confirm( )*

исполнение, и текущий загружаемый документ, если таковой существует, прекращает загружаться до тех пор, пока пользователь не отреагирует на запрос. Такому поведению методов нет альтернативы: возвращаемое ими значение – это введенные пользователем данные, поэтому они просто должны дождаться реакции пользователя перед тем, как вернуть значение. В большинстве броузеров метод `alert()` также является блокирующим и ожидает от пользователя закрытия диалогового окна.

Один из типичных вариантов использования метода `confirm()` приводится в примере 14.5, который создает диалоговое окно, показанное на рис. 14.2.

Пример 14.5. Использование метода confirm( )

```
function submitQuery() {
    // Это текст вопроса, который выводится перед пользователем.
    // Форматирование выполнено только с помощью символов
    // подчеркивания и перевода строки.
    var message = "\n\n\n\n" +
        "_____\n\n" +
        "На исполнение такого сложного запроса, как ваш,\n" +
        "может потребоваться минута или более.\n" +
        "_____\n\n\n" +
        "Щелкните на кнопке OK, чтобы продолжить,\n" +
        "или Отмена, чтобы отменить операцию.";

        // Запросить разрешение на выполнение операции
        // и прервать ее, если разрешение не будет получено
        if (!confirm(message)) return;

        /* Здесь находится программный код, выполняющий запрос */
}
```

14.6. Строка состояния

Веб-броузеры обычно отображают в нижней части любого окна *строку состояния*, предназначенную для вывода сообщений пользователю. Когда пользователь, например, наводит указатель мыши на гиперссылку, броузер обычно показывает URL-адрес, на который эта ссылка указывает.

Чтобы задать текст, который броузер должен вывести в строке состояния, в старых броузерах можно использовать свойство `status`. Это свойство обычно применяется для вывода в строке состояния описания документа в удобочитаемом ви-

де, когда пользователь наводит указатель мыши на гиперссылку. Сделать это можно примерно следующим образом:

```
Смущены? Попробуйте
<a href="help.html" onmouseover="status='Переход к справке!'; return true;">
обратиться к разделу справки</a>.
```

Когда указатель мыши оказывается на этой ссылке, исполняется JavaScript-код в обработчике события `onmouseover`. В результате в свойство `status` окна записывается текст и затем возвращается значение `true`, сообщая броузеру, что он не должен предпринимать собственное действие, выполняемое по умолчанию (отображать URL-адрес гиперссылки).

Этот фрагмент в современных броузерах уже не работает. Подобный программный код слишком часто применялся для преднамеренного обмана пользователя путем подмены целевого адреса (например, с целью мошенничества), что привело к наложению запрета на изменение свойства `status` в современных броузерах.

Объект `Window` обладает также свойством `defaultStatus`, позволяющим вывести текст в строке состояния, если броузер сам ничего другого в ней не выводит (например, URL-адрес гиперссылки). Это свойство работоспособно лишь в некоторых броузерах (например, в Firefox 1.0 возможность записи в свойство `defaultStatus` отсутствует, равно как и в свойство `status`).

Исторически свойство `defaultStatus` использовалось для создания анимационных эффектов в строке состояния. В прежние времена, когда содержимое документа еще не было доступно сценариям, но были доступны свойство `defaultStatus` и метод `setInterval()`, веб-разработчики часто поддавались искушению, создавая разнообразные кричащие и сбивающие с толку анимационные эффекты в стиле бегущей строки. К счастью, эти дни канули в лету. Тем не менее необходимость использовать строку состояния иногда все-таки возникает, причем даже совместно с методом `setInterval()`, как это демонстрируется в примере 14.6.

Пример 14.6. Анимационный эффект в строке состояния, выполненный со вкусом

```
<script>
var WastedTime = {
    start: new Date( ),          // Запомнить время начала
    displayElapsedTime: function() {
        var now = new Date();    // Получить текущее время
        // Подсчитать число прошедших минут
        var elapsed = Math.round((now - WastedTime.start)/60000);
        // И попробовать отобразить их в строке состояния
        window.defaultStatus = "Прошло " + elapsed + " минут.";
    }
}
// Обновлять строку состояния раз в минуту
setInterval(WastedTime.displayElapsedTime, 60000);
</script>
```

14.7. Обработка ошибок

Свойство `onerror` объекта `Window` – это особенный обработчик. Если присвоить этому свойству функцию, она будет вызываться во всех случаях, когда в окне воз-

никает ошибка – эта функция становится обработчиком ошибок для данного окна. (Обратите внимание: для тех же целей было бы достаточно определить глобальную функцию onerror(), т. к. это эквивалентно присваиванию функции свойству onerror объекта Window. Однако прием с определением функции onerror() в IE работать не будет.)

Обработчику ошибок передается три аргумента. Первый аргумент – это сообщение, описывающее произошедшую ошибку. Это может быть что-то вроде «отсутствует оператор в выражении», «свойство self доступно только для чтения» или «свойство myname не определено». Второй аргумент – это строка, содержащая URL-адрес документа с JavaScript-кодом, приведшим к ошибке. Третий аргумент – это номер строки в документе, где произошла ошибка. Обработчик ошибок может применять эти аргументы для разных целей. Типичный обработчик ошибок может показать сообщение пользователю, записать его в журнал или потребовать игнорирования ошибки. До появления JavaScript 1.5 обработчик onerror() мог использоваться как замена конструкции try/catch (см. главу 6) для обработки исключений.

Помимо этих трех аргументов, важную роль играет значение, возвращаемое обработчиком onerror(). Броузеры в случае возникновения ошибки обычно выводят сообщение в диалоговом окне или строке состояния. Если обработчик onerror() возвращает true, это говорит системе о том, что ошибка обработана и никаких дальнейших действий не требуется; другими словами, система не должна выводить собственное сообщение об ошибке.

За последние годы способ обработки JavaScript-кодом ошибок в броузерах изменился. Ранее, когда язык JavaScript еще был в диковинку, а броузеры были совсем еще юными, для них было обычным делом выводить диалоговые окна всякий раз, когда в сценарии возникала ошибка. Эти окна несли информацию, полезную для разработчика, но сбивали с толку конечного пользователя. Чтобы уберечь конечного пользователя от появления подобных диалоговых окон, в окончательных версиях веб-страниц (многие веб-страницы порождают ошибки в ходе исполнения JavaScript-сценариев, по крайней мере, в некоторых броузерах) можно просто определить обработчик ошибок, который ничего не выводит:

```
// Не беспокоить пользователя сообщениями об ошибках
window.onerror = function() { return true; }
```

С ростом объемов плохо продуманного и несовместимого JavaScript-кода в Интернете ошибки стали обычным делом, в результате броузеры стали регистрировать возникающие ошибки ненавязчивым образом. Это улучшило положение конечных пользователей, но усложнило жизнь разработчикам, которым теперь приходится открывать окно JavaScript-консоли (например, в Firefox), чтобы увидеть, возникали ли какие-либо ошибки. Чтобы упростить процесс отладки, можно воспользоваться примерно таким обработчиком ошибок:

```
// Вывести сообщение об ошибке в виде диалогового окна, но не более 3 раз
window.onerror = function(msg, url, line) {
    if (onerror.num++ < onerror.max) {
        alert("ОШИБКА: " + msg + "\n" + url + ":" + line);
        return true;
    }
}
```

```
onerror.max = 3;
onerror.num = 0;
```

14.8. Работа с несколькими окнами и фреймами

Большинство веб-приложений исполняются в единственном окне, хотя при этом могут открывать маленькие вспомогательные окна. Тем не менее вполне допустимо создавать приложения, которые используют в своей работе два или более фрейма или окна, обеспечив взаимодействие между этими фреймами или окнами с помощью JavaScript-кода. В этом разделе рассказывается, как это реализовать на практике.[1]

Прежде чем приступить к обсуждению темы создания веб-приложений с несколькими окнами или фреймами, есть смысл еще раз вспомнить положения политики общего происхождения, описываемые в разделе 13.8.2. Данная политика позволяет JavaScript-сценарию взаимодействовать с содержимым только тех документов, которые получены с того же самого сервера, что и документ с этим сценарием. Любые попытки прочитать содержимое или свойства документа, полученного с другого веб-сервера, окажутся неудачными. Это значит, например, что можно написать такую программу на языке JavaScript, которая будет индексировать собственный веб-сайт и составлять список ссылок в документах, представленных на этом сайте. Однако невозможно расширить возможности этой программы таким образом, чтобы она могла следовать по этим ссылкам и индексировать другие сайты: попытки получить список ссылок из документов, расположенных за пределами сайта, окажутся неудачными. Программный код, который не работает из-за ограничений, накладываемых политикой общего происхождения, вы найдете в примере 14.7.

14.8.1. Отношения между фреймами

Мы уже видели, что метод `open()` объекта `Window` возвращает новый объект `Window`, представляющий только что созданное окно. Также мы видели, что это новое окно имеет свойство `opener`, ссылающееся на первоначальное окно. Так два окна могут ссылаться друг на друга, и каждое из них может читать свойства и вызывать методы другого. То же самое возможно для фреймов. Любой фрейм в окне может ссылаться на любой другой фрейм при помощи свойств `frames`, `parent` и `top` объекта `Window`.

JavaScript-код в любом окне или фрейме может сослаться на собственное окно или фрейм с помощью свойств `window` или `self`. Поскольку каждое окно или фрейм – это глобальный объект для содержащегося в них программного кода, совершенно не обязательно использовать свойство `window` или `self` для ссылки на сам глобальный объект. То есть если требуется сослаться на метод или свойство глобального объекта (хотя из соображений стилевого оформления это может

[1] На ранних этапах развития JavaScript веб-приложения с множеством фреймов и окон были обычным явлением. Ныне в соответствии с общепринятыми принципами веб-дизайна фреймы использовать не рекомендуется (это не относится к *плавающим фреймам*, которые называются *iframes*), благодаря чему все реже можно встретить веб-сайты, где присутствуют взаимодействующие друг с другом окна.

быть полезно), использовать префикс window или self при обращении к методам или свойствам глобального объекта не обязательно.

Любое окно имеет свойство frames. Это свойство ссылается на массив объектов Window, каждый из которых представляет содержащийся внутри окна фрейм. (Если окно не содержит фреймов, массив frames[] пуст, и значение frames.length равно нулю.) Следовательно, окно (или фрейм) может ссылаться на свой первый подфрейм как на элемент frames[0], на второй подфрейм – как на элемент frames[1] и т. д. Аналогично JavaScript-код, работающий в окне, может следующим образом ссылаться на третий подфрейм второго фрейма этого окна:

```
frames[1].frames[2]
```

Каждое окно имеет также свойство parent, ссылающееся на объект Window, в котором это окно содержится. Следовательно, первый фрейм в окне может сослаться на смежный с ним фрейм (второй фрейм того же окна) так:

```
parent.frames[1]
```

Если окно является окном верхнего уровня, а не фреймом, свойство parent просто ссылается на само окно:

```
parent == self; // Для любого окна верхнего уровня
```

Если фрейм находится внутри другого фрейма, содержащегося в окне верхнего уровня, то он может сослаться на окно верхнего уровня так: parent.parent. Однако в качестве универсального сокращения имеется свойство top: независимо от глубины вложенности фрейма его свойство top ссылается на содержащее его окно самого верхнего уровня. Если объект Window представляет окно верхнего уровня, top просто ссылается на само окно. Для фреймов, непосредственно принадлежащих окну верхнего уровня, свойство top совпадает со свойством parent.

Фреймы обычно создаются с помощью тегов <frameset> и <frame>. Однако в HTML 4 может также использоваться тег <iframe>, создающий в документе плавающий фрейм. Для JavaScript фреймы, созданные с помощью тега <iframe>, – это то же самое, что фреймы, созданные с помощью тегов <frameset> и <frame>. Все, о чем говорилось ранее, применимо к обоим видам фреймов.

Рисунок 14.3 иллюстрирует эти отношения между фреймами и показывает, как код, работающий в одном фрейме, может ссылаться на любой другой фрейм посредством свойств frames, parent и top. Здесь окно броузера содержит два фрейма – один над другим. Второй фрейм (больший, он расположен снизу) сам содержит три подфрейма, расположенных бок о бок.

14.8.2. Имена окон и фреймов

Второй (необязательный) аргумент обсуждавшегося ранее метода Window.open() – это имя только что созданного окна. Создавая фрейм с помощью тега <frame>, можно с помощью атрибута name задать его имя. Важным основанием именования окон и фреймов является то, что их имена могут затем использоваться в качестве значений атрибута target тегов <a> и <form>. Это значение сообщает броузеру, где вы хотите видеть результат активизации ссылки или подтверждения формы.

Например, если у вас имеется два окна, одно с именем table_of_contents, а другое – mainwin, в окне table_of_contents может быть следующий HTML-код:

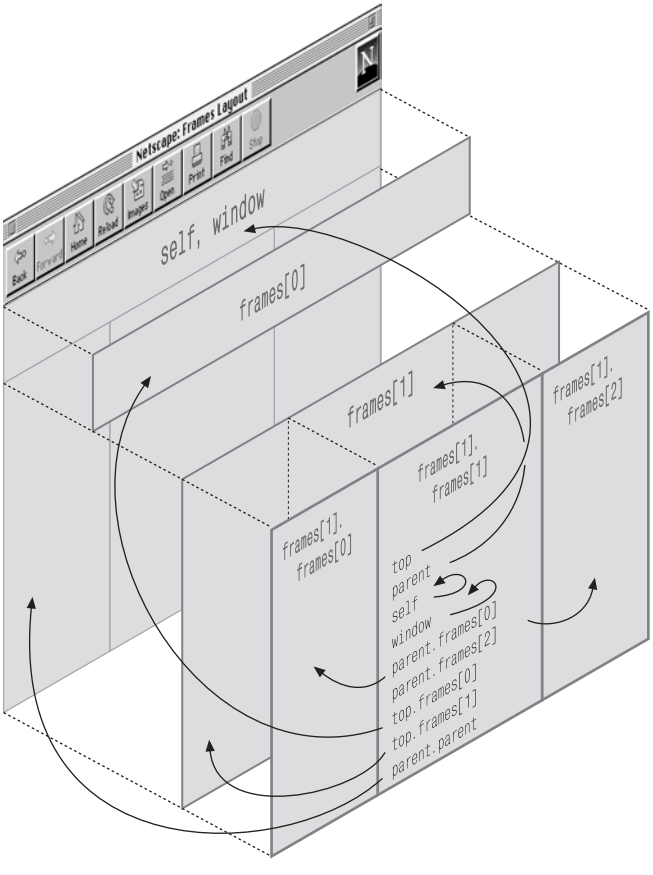

***Рис. 14.3.** Отношения между фреймами*

```
<a href="chapter01.html" target="mainwin">Глава 1. Введение</a>
```

Когда пользователь щелкает на этой гиперссылке, броузер загружает указанный URL-адрес, но вывод осуществляется не в окно, в котором находится ссылка, а в окно с именем `mainwin`. Если окно с именем `mainwin` отсутствует, щелчок на ссылке создает новое окно с этим именем, и документ с указанным URL-адресом загружается в это окно.

Атрибуты `target` и `name` являются частью HTML-кода и работают без вмешательства JavaScript, но есть и связанные с JavaScript причины присваивания имен фреймам. Мы видели, что в любом объекте `Window` имеется массив `frames[]`, содержащий ссылки на все фреймы окна (или фрейма) независимо от того, есть у них имена или нет. Однако если фрейму дано имя, ссылка на этот фрейм также сохраняется в новом свойстве родительского объекта `Window`. Имя нового свойства совпадает с именем фрейма. Следовательно, можно создать фрейм с помощью следующего HTML-кода:

```
<frame name="table_of_contents" src="toc.html">
```

После этого на данный фрейм можно ссылаться из другого смежного с ним фрейма:

```
parent.table_of_contents
```

Такой код проще читать и понимать, чем код, в котором индекс массива жестко закодирован (и вы зависите от него), что неизбежно в случае безымянного фрейма:

```
parent.frames[1]
```

В примере 14.7 в конце этой главы программный код ссылается на фреймы по именам с использованием только что описанного приема.

14.8.3. JavaScript во взаимодействующих окнах

В главе 13 уже говорилось, что объект Window выступает в качестве глобального объекта для клиентского JavaScript-кода, а окно – в качестве контекста исполнения для всего содержащегося в нем JavaScript-кода. Это относится и к фреймам: каждый фрейм представляет собою независимый контекст исполнения JavaScript-кода. Каждый объект Window является отдельным глобальным объектом, поэтому в каждом окне определено собственное пространство имен и свой набор глобальных переменных. Если смотреть с точки зрения работы с несколькими фреймами или окнами, то глобальные переменные уже не кажутся такими глобальными!

Несмотря на то что каждое окно или фрейм определяет независимый контекст исполнения JavaScript-кода, это не значит, что код, исполняющийся в одном окне, изолирован от кода в других окнах. Код, исполняющийся в одном фрейме, имеет в вершине своей цепочки областей видимости объект Window, отличный от того, который имеет код, исполняющийся в другом фрейме. Однако код из обоих фреймов исполняется одним и тем же интерпретатором JavaScript в одной и той же среде. Как мы видели, фрейм может ссылаться на любой другой фрейм с помощью свойств frames, parent и top. Поэтому, хотя JavaScript-код в разных фреймах исполняется с различными цепочками областей видимости, тем не менее код в одном фрейме может обращаться к переменным и функциям, определенным в коде другого фрейма, и использовать их.

Предположим, что код во фрейме A определяет переменную i:

```
var i = 3;
```

Это переменная представляет собой свойство глобального объекта, т. е. свойство объекта Window. Код во фрейме A может явно ссылаться на эту переменную как на свойство с помощью любого из двух выражений:

```
window.i
self.i
```

Теперь предположим, что у фрейма A имеется смежный фрейм B, который пытается установить значение переменной i, определенной в коде фрейма A. Если фрейм B просто присвоит значение переменной i, он лишь успешно создаст новое свойство собственного объекта Window. Поэтому он должен явно сослаться на свойство i смежного объекта с помощью следующего кода:

```
parent.frames[0].i = 4;
```

Вспомните, что ключевое слово function, определяющее функцию, объявляет переменную так же, как ключевое слово var. Если JavaScript-код во фрейме A объявляет функцию f, эта функция определяется только внутри фрейма A. Код во фрейме A может вызывать функцию f следующим образом:

```
f();
```

Однако код во фрейме B должен ссылаться на f, как на свойство объекта Window фрейма A:

```
parent.frames[0].f();
```

Если код во фрейме B часто вызывает эту функцию, можно присвоить ее переменной фрейма B, так чтобы было удобнее ссылаться на функцию:

```
var f = parent.frames[0].f;
```

Теперь код во фрейме B может вызывать функцию как f() точно так же, как код фрейма A.

Разделяя подобным образом функции между фреймами или окнами, очень важно помнить о правилах лексического контекста. Функции исполняются в том контексте, в котором они определены, а не в том, из которого они вызываются. Следовательно, продолжая предыдущий пример, если функция f ссылается на глобальные переменные, поиск этих переменных выполняется в свойствах фрейма A, даже когда функция вызывается из фрейма B.

Если не обращать на это особого внимания, могут получаться программы, ведущие себя неожиданным и запутанным образом. Предположим, что вы определили в секции <head> документа, содержащего несколько фреймов, следующую функцию, думая, что она поможет вам при отладке:

```
function debug(msg) {
    alert("Отладочное сообщение от фрейма: " + name + "\n" + msg);
}
```

JavaScript-код в каждом из ваших фреймов может ссылаться на эту функцию так: top.debug(). Однако при ее вызове функция будет искать переменную name в контексте окна верхнего уровня, в котором определена функция, а не в контексте фрейма, из которого она вызвана. В результате отладочные сообщения всегда будут содержать имя окна верхнего уровня, а не имя фрейма, посылающего сообщение, как это предполагалось.

Помните, что конструкторы – это тоже функции, поэтому, когда вы определяете класс объектов с функцией-конструктором и связанным с ним объектом-прототипом, этот класс оказывается определен только для одного окна. Вспомним класс Complex, который мы определили в главе 9, и рассмотрим следующий HTML-документ с несколькими фреймами:

```
<head>
<script src="Complex.js"></script>
</head>
<frameset rows="50%,50%">
    <frame name="frame1" src="frame1.html">
    <frame name="frame2" src="frame2.html">
</frameset>
```

JavaScript-код в файлах *frame1.html* и *frame2.html* не может создать объект `Complex` с помощью примерно такого выражения:

```
var c = new Complex(1,2); // Не работает ни из одного фрейма
```

Он должен явно ссылаться на функцию-конструктор:

```
var c = new top.Complex(3,4);
```

В качестве альтернативы код в любом фрейме может определять собственные переменные для более удобного обращения к функции-конструктору:

```
var Complex = top.Complex;
var c = new Complex(1,2);
```

В отличие от пользовательских конструкторов, предопределенные конструкторы оказываются автоматически определенными во всех окнах. Однако следует заметить, что каждое окно имеет независимую копию конструктора и независимую копию объекта-прототипа конструктора. Например, каждое окно имеет собственную копию конструктора `String()` и объекта `String.prototype`. Поэтому если вы создадите новый метод для работы с JavaScript-строками и сделаете его методом класса `String`, присвоив его объекту `String.prototype` текущего окна, все строки в этом окне смогут использовать новый метод, однако этот новый метод будет недоступен строкам, определенным в других окнах. Обратите внимание: не имеет значения, в каком из окон содержится ссылка на строку; имеет значение только окно, в котором фактически создана строка.

14.9. Пример: панель навигации во фрейме

Эта глава заканчивается примером, демонстрирующим несколько наиболее важных приемов работы с окнами, которые были здесь описаны:

- Запрос текущего URL-адреса с помощью свойства `location.herf` и загрузка нового документа установкой нового значения в свойство `location`.
- Использование методов `back()` и `forward()` объекта `History`.
- Использование метода `setTimeout()` для отложенного вызова функции.
- Открытие нового окна броузера методом `window.open()`.
- Использование JavaScript-кода из одного фрейма для взаимодействия с другим фреймом.
- Демонстрация ограничений, накладываемых политикой общего происхождения.

Пример 14.7 представляет собой сценарий и несложную HTML-форму, предназначенную для работы с документом в другом фрейме. В одном фрейме создается простая панель навигации, которая используется для управления содержимым другого фрейма. Панель навигации включает в себя кнопки Назад и Вперед, а также текстовое поле, в которое вводится URL-адрес. Панель навигации можно видеть в нижней части окна броузера на рис. 14.4.

В теге `<script>` примера определяются функции, а кнопки и текстовое поле для ввода URL-адреса – в теге `<form>`. Для вызова функций используются обработчики событий щелчков на кнопках. Хотя до сих пор обработчики событий для HTML-форм еще не обсуждались, для понимания данного примера это несущественно.

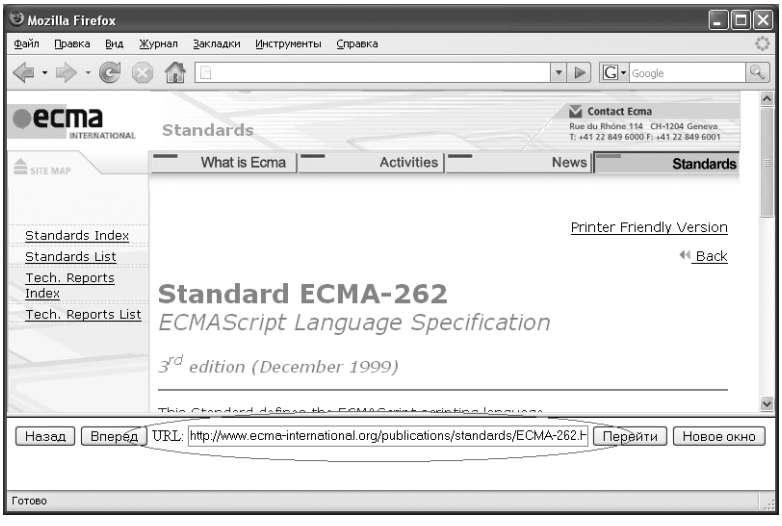

***Рис. 14.4.** Панель навигации*

Из примера 14.7 вы узнаете, как используются объекты `History` и `Location`, функции `setTimeout()` и `Window.open()`. Увидите, как JavaScript-код из фрейма с панелью навигации ссылается на другой фрейм по имени. Кроме того, вам встретятся блоки `try/catch` в тех местах, где согласно положениям политики общего происхождения могут генерироваться исключения.

Пример 14.7. Панель навигации

```
<!--
    Этот файл реализует панель навигации, предназначенную для фрейма
    в нижней части окна. Включите ее в набор фреймов следующим образом:

    <frameset rows="*,75">
        <frame src="about:blank" name="main">
        <frame src="navigation.html">
    </frameset>

    Программный код из этого файла управляет содержимым фрейма с именем "main"
-->
<script>
// Эта функция вызывается щелчком на кнопке Назад в панели навигации
function back() {
    // Для начала следует очистить поле ввода URL-адреса в форме
    document.navbar.url.value = "";

    // Затем с помощью объекта History главного фрейма нужно вернуться
    // назад, если этому не противоречит политика общего происхождения
    try { parent.main.history.back( ); }
    catch(e) {
        alert("Вызов History.back() заблокирован " +
              "политикой общего происхождения: " + e.message);
    }

    // Отобразить URL-адрес документа, к которому был выполнен переход,
```

```
    // если это получилось. Вызов updateURL() откладывается,
    // чтобы свойство location.href успело обновиться.
    setTimeout(updateURL, 1000);
}

// Эта функция вызывается щелчком на кнопке Вперед в панели навигации.
function forward() {
    document.navbar.url.value = "";
    try { parent.main.history.forward(); }
    catch(e) {
        alert("Вызов History.forward() заблокирован " +
              "политикой общего происхождения: "+e.message);
    }
    setTimeout(updateURL, 1000);
}

// Следующая частная функция вызывается функциями back() и forward()
// для обновления текстового поля URL-адреса в форме. Обычно политика общего
// происхождения запрещает изменение свойства location в главном фрейме.
function updateURL() {
    try { document.navbar.url.value = parent.main.location.href; }
    catch(e) {
        document.navbar.url.value = "<Политика общего происхождения " +
                                    "блокирует доступ к URL>";
    }
}

// Вспомогательная функция: если URL не начинается с префикса "http://", добавить его.
function fixup(url) {
    if (url.substring(0,7) != "http://") url = "http://" + url;
    return url;
}

// Эта функция вызывается щелчком на кнопке Перейти в панели навигации,
// а также при подтверждении пользователем формы
function go() {
    // И загружает документ с заданным URL-адресом в главный фрейм.
    parent.main.location = fixup(document.navbar.url.value);
}

// Открывает новое окно и отображает в нем URL-адрес, заданный пользователем
function displayInNewWindow() {
    // Открыть обычное неименованное полноценное окно, для чего достаточно
    // определить аргумент URL. После того как окно будет открыто,
    // панель навигации потеряет контроль над ним.
    window.open(fixup(document.navbar.url.value));
}
</script>

<!-Далее следует форма с обработчиками событий,
    которые вызывают определенные ранее функции -->
<form name="navbar" onsubmit="go(); return false;">
    <input type="button" value="Назад" onclick="back();">
    <input type="button" value="Вперед" onclick="forward();">
    URL: <input type="text" name="url" size="50">
    <input type="button" value="Перейти" onclick="go();">
    <input type="button" value="Новое окно" onclick="displayInNewWindow();">
</form>
```

15

Работа с документами

Клиентский JavaScript предназначен для того, чтобы превращать статические HTML-документы в интерактивные веб-приложения. Работа с содержимым веб-страниц – это главное предназначение JavaScript. Данная глава является наиболее важной во второй части – здесь рассказывается о том, как это делается.

Каждое окно (или фрейм) веб-броузера отображает HTML-документ. Объект `Window`, представляющий окно, имеет свойство `document`, которое ссылается на объект `Document`. Этот объект `Document` и является темой обсуждения данной главы, которая начинается с изучения свойств и методов самого объекта `Document`. Но эта интересная тема – лишь начало.

Более интересными, чем объект `Document`, являются объекты, которые представляют *содержимое* документа. HTML-документы могут содержать текст, изображения, гиперссылки, элементы форм и т. д. JavaScript-сценарии могут обращаться ко всем объектам, которые представляют элементы документа, и манипулировать ими. Прямой доступ к объектам, представляющим содержимое документа, дает широчайшие возможности, но одновременно означает определенные сложности.

Объектная модель документа (DOM) – это прикладной программный интерфейс (API), определяющий порядок доступа к объектам, из которых состоит документ. Консорциумом W3C был выработан стандарт DOM, достаточно полно поддерживаемый всеми современными броузерами. К сожалению, такое положение дел имело место не всегда. В реальности история развития клиентского JavaScript-программирования – это история развития DOM (причем не всегда в согласованных направлениях). В первые годы существования Всемирной паутины ведущим производителем броузеров была компания Netscape, и именно она определяла прикладные интерфейсы для разработки клиентских сценариев. Броузеры Netscape 2 и 3 поддерживали упрощенную версию модели DOM, которая предоставляла возможность доступа только к отдельным элементам, таким как ссылки, изображения и элементы форм. Эта устаревшая спецификация DOM была воспринята всеми производителями броузеров и формально включена в стандарт консорциума W3C как DOM Level 0. Данная спецификация до сих пор поддерживается во всех броузерах, а потому мы рассмотрим ее в первую очередь.

С появлением Internet Explorer 4 доминирование во Всемирной паутине перешло к Microsoft. В броузере IE 4 была реализована совершенно новая объектная модель документа, которая давала возможность обращаться ко всем элементам документа и взаимодействовать с ними довольно интересными способами. Она даже позволяла изменять текст документа за счет перестановки местами абзацев, если в этом возникала необходимость. Прикладной интерфейс, разработанный в Microsoft, получил название IE 4 DOM. Однако он так и не был стандартизован, поэтому в IE 5 и более поздних версиях был реализован стандарт W3C DOM, при этом поддержка IE 4 DOM была сохранена. Частично модель IE 4 DOM была реализована в других броузерах и по-прежнему используется во Всемирной паутине. Подробнее эта модель обсуждается в конце главы в сравнении с ее стандартной альтернативой.

В Netscape 4 был выбран совершенно иной подход к реализации DOM, в основе которого лежали динамически позиционируемые программируемые элементы, получившие название *слои* (*layers*). Модель Netscape 4 DOM оказалась эволюционным тупиком и поддерживалась только в Netscape 4. При разработке броузеров Mozilla, Firefox и всех остальных, основанных на программном коде Netscape, было решено отказаться от этой модели. В результате модель Netscape 4 DOM в этой редакции книги не рассматривается.

Значительная часть данной главы посвящена описанию стандарта W3C DOM. Следует отметить, что при этом обсуждаются только базовые положения стандарта. Управление содержимым документа – это основная цель клиентского JavaScript-кода, поэтому большинство последующих глав этой книги в действительности можно рассматривать как продолжение этой главы. В главе 16 рассказывается о стандарте W3C DOM в отношении работы с CSS-стилями и таблицами стилей, а в главе 17 – в отношении обработки событий (а также о приемах программирования, унаследованных от IE 4). Глава 18 касается порядка работы с тегами `<img>` HTML-документа и рассказывает о том, как создавать графические изображения на стороне клиента.

В модели DOM Level 0 определяется единственный класс `Document`, и в этой главе имеются многочисленные неформальные ссылки на объект `Document`. Однако стандарт W3C DOM определяет универсальный прикладной интерфейс `Document`, который описывает функциональность документа, в равной степени применимую и к HTML-, и к XML-документам, а также специализированный интерфейс `HTMLDocument`, добавляющий свойства и методы, характерные для HTML-документов. Справочный материал, приводимый в четвертой части книги, следует соглашениям W3C, поэтому если вы ищете свойства HTML-документа, их необходимо искать в разделе с описанием интерфейса `HTMLDocument`. Большая часть функциональных возможностей модели DOM Level 0 относится к HTML-документам, поэтому их описание также следует искать в разделе, посвященном интерфейсу `HTMLDocument`, хотя в этой главе они упоминаются как свойства и методы объекта `Document`.

15.1. Динамическое содержимое документа

Исследование объекта `Document` начнется с метода `write()`, который позволяет записывать содержимое в тело документа. Этот метод относится к унаследованной части DOM, и начиная с самых ранних версий JavaScript метод `document.write()` можно было использовать двумя способами. Первый и самый простой способ –

вывести HTML-текст из сценария в тело документа, анализ которого производится в текущий момент. Рассмотрим следующий фрагмент, где в статический HTML-документ с помощью метода `write()` добавляется информация о текущей дате:

```
<script>
    var today = new Date();
    document.write("<p>Документ открыт: " + today.toString( ));
</script>
```

Необходимо отметить, что вывод текста в формате HTML в текущий документ возможен только в процессе его синтаксического анализа. То есть вызывать метод `document.write()` из программного кода верхнего уровня в теге `<script>` можно только в том случае, если исполнение сценария является частью процесса анализа документа. Если поместить вызов `document.write()` в определение функции и затем вызвать эту функцию из обработчика события, результат окажется неожиданным – фактически этот вызов уничтожит текущий документ и все содержащиеся в нем сценарии! (Причины такого поведения вскоре будут описаны.)

Метод `document.write()` вставляет текст в то место HTML-документа, где находится тег `<script>`, содержащий вызов метода. Если тег `<script>` помечен атрибутом `defer`, он не должен содержать никаких обращений к методу `document.write()`. Атрибут `defer` сообщает веб-броузеру, что исполнение сценария может быть отложено до того момента, когда документ будет полностью загружен. Но когда это произойдет, будет уже слишком поздно вставлять дополнительное содержимое в документ методом `document.write()`, поскольку разбор документа уже закончится.

Использование метода `write()` для создания содержимого документа в процессе его разбора – это широко распространенная практика JavaScript-программирования. Ныне стандарт W3C DOM позволяет вставлять содержимое (с помощью описываемых далее приемов) в любую часть документа уже после того, как закончится его анализ. Тем не менее применение метода `document.write()` по-прежнему является делом вполне обычным.

Кроме того, метод `write()` можно использовать (совместно с методами `open()` и `close()` объекта `Document`) для создания полностью новых документов в других окнах и фреймах. Хотя возможность выполнить запись в текущий документ из обработчика события отсутствует, нет никаких причин, которые могли бы воспрепятствовать выполнить запись в документ в другом фрейме или окне – это может оказаться удобным при создании многооконных веб-приложений или страниц с несколькими фреймами. Например, можно было бы создать всплывающее окно и записать в него некоторый HTML-код следующим образом:

```
// Эта функция открывает всплывающее окно. Она должна вызываться из обработчика события,
// в противном случае всплывающее окно, скорее всего, будет заблокировано
function hello() {
    var w = window.open();              // Создать новое пустое окно
    var d = w.document;                 // Получить ссылку на объект Document
    d.open();                           // Начать новый документ (необязательно)
    d.write("<h1>Привет, МИР!</h1>"); // Вывести содержимое документа
    d.close();                          // Закрыть документ
}
```

Чтобы создать новый документ, прежде всего нужно вызвать метод `open()` объекта `Document`, затем вызвать несколько раз метод `write()`, чтобы вывести содержи-

мое документа, и наконец вызвать метод close() объекта Document, чтобы указать, что работа с документом окончена. Этот последний шаг очень важен – если не закрыть документ, броузер будет продолжать показывать, что идет загрузка документа. Кроме того, броузер может буферизовать только что записанный HTML-текст и не отображать его, пока документ не будет явно закрыт методом close().

В отличие от метода close(), вызывать метод open() не обязательно. Если метод write() вызывается для уже закрытого документа, интерпретатор JavaScript неявно открывает новый HTML-документ, как если бы перед первым вызовом write() стоял вызов open(). Это объясняет происходящее, когда вызов метода document.write() производится из обработчика события в том же самом документе: JavaScript открывает новый документ. В результате текущий документ (и все его содержимое, включая сценарии и обработчики событий) уничтожается. Главное правило, которое следует соблюдать: метод write() никогда не должен вызываться для записи в тот же самый документ из обработчиков событий.

Два последних замечания по поводу метода write(). Во-первых, многие еще не понимают, что метод write() способен принимать более одного аргумента. Когда методу передаются несколько аргументов, они выводятся один за другим, как если бы были объединены в одну строку. Например:

```
document.write("Привет, " + username + " Добро пожаловать на мою страницу!");
```

Этот вызов можно заменить следующим фрагментом:

```
var greeting = "Привет, ";
var welcome = " Добро пожаловать на мою страницу!";
document.write(greeting, username, welcome);
```

Во-вторых, объект Document поддерживает еще один метод – writeln(), который идентичен методу write() за исключением того, что после вывода последнего аргумента добавляет символ перевода строки. Это может оказаться удобным, например, при выводе отформатированного текста в теге <pre>.

Полное описание методов write(), writeln(), open() и close() вы найдете в четвертой части книги в том разделе, в котором описывается объект HTMLDocument.

15.2. Свойства объекта Document

Рассмотрев «старейшие» методы объекта Document, перейдем к его «старейшим» свойствам:

bgColor

Цвета фона документа. Это свойство соответствует атрибуту bgcolor тега <body>.

cookie

Специальное свойство, позволяющее JavaScript-программам читать и писать cookie-файлы. Этому свойству посвящена отдельная глава – глава 19.

domain

Свойство, которое позволяет доверяющим друг другу веб-серверам, принадлежащим одному домену, ослаблять связанные с политикой общего происхождения ограничения на взаимодействие между их веб-страницами (подробности см. в разделе 13.8.2).

`lastModified`

Строка, содержащая дату последнего изменения документа.

`location`

Устаревший синоним свойства `URL`.

`referrer`

URL-адрес документа, содержащего ссылку (если таковая существует), которая привела броузер к текущему документу.

`title`

Текст между тегами `<title>` и `</title>` данного документа.

`URL`

Строка, задающая URL-адрес, с которого был загружен документ. Значение этого свойства совпадает со значением свойства `location.href` объекта `Window` за исключением случая перенаправления на стороне сервера.

Некоторые из этих свойств предоставляют информацию о документе в целом. Следующий фрагмент можно поместить в конец каждого вашего документа, чтобы автоматически предоставлять пользователю дополнительные сведения о документе, которые позволят судить о том, насколько устарел этот документ:

```
<hr><font size="1">
    Документ: <i><script>document.write(document.title);</script></i><br>
    URL: <i><script>document.write(document.URL);</script></i><br>
    Дата последнего обновления:
    <i><script>document.write(document.lastModified);</script></i>
</font>
```

Еще одно интересное свойство – `referrer`. Оно содержит URL-адрес документа, из которого пользователь перешел к текущему документу по ссылке. Это свойство позволяет предотвратить создание глубоких ссылок в недра вашего сайта. Если вы желаете, чтобы все посетители обязательно попадали на вашу домашнюю страницу, можно организовать перенаправление, разместив следующий фрагмент в начале всех страниц, за исключением домашней:

```
<script>
// Если переход выполнен по ссылке из-за пределов сайта,
// выполнить перенаправление на домашнюю страницу
if (document.referrer == "" || document.referrer.indexOf("mysite.com") == -1)
    window.location = "http://home.mysite.com";
</script>
```

Конечно, этот прием не следует рассматривать как серьезную защитную меру. Вполне очевидно, что он не будет работать у пользователей, отключивших в своих веб-броузерах режим исполнения JavaScript-кода.

Последнее интересное свойство объекта `Document` – свойство `bgColor`. Оно соответствует HTML-атрибуту, потому использовать его не рекомендуется. Это свойство упомянуто здесь лишь по историческим причинам – первая клиентская JavaScript-программа изменяла цвет фона документа. Даже очень-очень старые веб-броузеры изменят цвет фона документа, если в свойство `document.bgColor` записать строку, устанавливающую цвет, например `"pink"` или `"#FFAAAA"`.

Полное описание этих старейших свойств объекта Document приводится в четвертой части книги в том разделе, в котором описывается объект HTMLDocument.

Объект Document обладает другими важными свойствами, значениями которых являются массивы объектов документа. Эти коллекции станут темой обсуждения следующего раздела.

15.3. Ранняя упрощенная модель DOM: коллекции объектов документа

В списке свойств объекта Document, который приводился в предыдущем разделе, отсутствуют важные категории свойств – коллекции объектов документа. Эти свойства, представляющие собой массивы, являются сердцем ранней объектной модели документа. С их помощью обеспечивается доступ к некоторым специальным элементам документа:

anchors[]

Массив объектов Anchor, представляющих якорные элементы документа. *Якорный элемент* (*anchor*) – это именованная позиция в документе, которая создается с помощью тега <a> и в которой вместо атрибута href определяется атрибут name. Свойство name объекта Anchor хранит значение атрибута name. Полное описание объекта Anchor вы найдете в четвертой части книги.

applets[]

Массив объектов Applet, представляющих Java-апплеты в документе. Подробно апплеты обсуждаются в главе 23.

forms[]

Массив объектов Form, представляющих элементы <form> в документе. Каждый объект Form обладает собственным свойством-коллекцией с именем elements[], в котором содержатся объекты, представляющий элементы формы. Прежде чем форма будет отправлена, объекты Form вызывают обработчик события onsubmit. Этот обработчик может выполнить проверку правильности заполнения формы на стороне клиента: если он возвратит значение false, броузер отменит операцию отправки формы. Коллекция forms[] – самое важное свойство ранней версии DOM. Формы и элементы форм обсуждаются в главе 18.

images[]

Массив объектов Image, представляющих элементы <img> в документе. Свойство src объекта Image доступно для чтения/записи. Запись строки URL-адреса в это свойство вынуждает броузер прочитать и отобразить новое изображение (в старых версиях броузеров размеры нового изображения должны были совпадать с размерами оригинала). Программирование свойства src объекта Image позволяет организовать листание изображений и простейшие виды анимации. Подробнее об этом рассказывается в главе 22.

links[]

Массив объектов Link, представляющих гипертекстовые ссылки в документе. Гипертекстовые ссылки в языке HTML создаются с помощью тегов <a>, а при создании карт ссылок для изображений – с помощью тегов <area>. Свойство

href объекта Link соответствует атрибуту href тега <a>: в нем хранится строка URL-адреса ссылки. Кроме того, объекты Link обеспечивают доступ к различным элементам URL-адреса через свойства, такие как protocol, hostname и pathname. Благодаря этому объект Link напоминает объект Location, обсуждавшийся в главе 14. Когда указатель мыши наводится на ссылку, объект Link вызывает обработчик события onmouseover, а когда уводится со ссылки – обработчик события onmouseout. Когда производится щелчок мышью на ссылке, объект Link вызывает обработчик события onclick. Если обработчик события вернет false, броузер не выполнит переход по ссылке. Полное описание объекта Link приводится в четвертой части книги.

Как следует из имен этих свойств, они являются коллекциями всех ссылок, изображений, форм и прочего, что имеется в документе. Элементы этих массивов располагаются в том же порядке, в котором они находятся в исходном документе. Например, элемент document.forms[0] ссылается на первый тег <form> в документе, а document.images[4] – на пятый тег <img>.

Объекты, содержащиеся в этих коллекциях ранней версии DOM, доступны для JavaScript-программ, но вы должны понимать, что ни один из них не дает возможности изменить *структуру* документа. Вы можете проверять адреса ссылок и изменять их, читать или записывать значения элементов форм и даже менять местами изображения, но вы не сможете изменить текст документа. Старые броузеры, такие как Netscape 2, 3 и 4, а также IE 3, были не в состоянии переформатировать текст документа после того, как он проанализирован и отображен. По этой причине ранняя версия DOM не позволяла (и не позволяет) вносить изменения, которые могут привести к переформатированию текста. Например, ранняя версия DOM включает в себя API-функцию для добавления новых элементов <option> внутри элемента <select>. Это возможно потому, что HTML-формы отображают элементы <select> как раскрывающиеся меню, а добавление новых пунктов в такие меню не влияет на размещение других элементов формы. В то же время в ранней версии DOM отсутствует API-функция для добавления новых переключателей на форму или новых строк в таблицу, потому что эти изменения требуют переформатирования документа.

15.3.1. Именование объектов документа

Проблема использования числовых индексов при работе с коллекциями объектов документа состоит в том, что незначительные изменения, которые влекут за собой переупорядочивание элементов, могут привести к нарушениям в работе сценариев, опирающихся на исходный порядок следования элементов. Более надежное решение заключается в том, чтобы присваивать имена важным элементам документа и затем обращаться к ним по этим именам. В ранней версии DOM для этих целей можно было задействовать атрибут name форм, элементов форм, изображений, апплетов и ссылок.

Если атрибут присутствует, его значение используется в качестве имени соответствующего объекта. Например, предположим, что HTML-документ содержит следующую форму:

```
<form name="f1"><input type="button" value="Нажми меня"></form>
```

Допустим, что тег <form> является первым таким тегом в документе, тогда из JavaScript-сценария к получившемуся объекту Form можно обратиться любым из трех способов:

```
document.forms[0]    // По номеру формы внутри документа
document.forms.f1    // По имени, как к свойству
document.forms["f1"] // По имени, как к элементу массива
```

Фактически установка атрибута name в тегах <form>, <img> и <applet> (но не в теге <a>) позволяет обращаться к соответствующим объектам Form, Image и Applet (но не к объектам Link и Anchor), как к именованным свойствам объекта Document. То есть к форме можно обратиться так:

```
document.f1
```

Элементы внутри формы также могут иметь имена. Если был определен атрибут name в элементе формы, объект, который представляет этот элемент, становится доступным в качестве свойства соответствующего объекта Form. Предположим, что у нас имеется следующая форма:

```
<form name="shipping">
    ...
    <input type="text" name="zipcode">
    ...
</form>
```

Тогда сослаться на элемент текстового поля ввода в этой форме можно с помощью интуитивно понятного синтаксиса:

```
document.shipping.zipcode
```

В этом месте необходимо сделать последнее замечание об именовании элементов документа в ранней версии DOM. Что произойдет, если два элемента документа имеют в атрибуте name одно и то же значение? Если, например, теги <form> и <img> оба имеют имя «n», тогда свойство document.n превратится в массив, которое будет хранить ссылки на оба элемента.

Обычно вы должны стремиться к тому, чтобы подобная ситуация не повторилась у вас и обеспечить уникальность значений атрибутов name. Однако в одном случае такое положение вещей – вполне обычное дело. Согласно соглашениям для группировки переключателей и флажков на HTML-формах этим элементам необходимо присваивать одинаковые имена. В результате имя становится свойством объекта Form, а значением этого свойства – массив ссылок на различные объекты переключателей или флажков. Подробнее об этом рассказывается в главе 18.

15.3.2. Обработчики событий в объектах документа

Интерактивный HTML-документ и находящиеся в нем элементы должны реагировать на пользовательские события. Мы кратко обсуждали события и их обработчики в главе 13, в которой познакомились с несколькими примерами простых обработчиков. В этой главе имеется намного больше примеров обработчиков событий, т. к. они играют ключевую роль во взаимодействии объектов документа с JavaScript-кодом.

К сожалению, мы должны отложить полноценное обсуждение событий и обработчиков событий до главы 17. А сейчас вспомните, что обработчики событий определяются атрибутами HTML-элементов, такими как `onclick` и `onmouseover`. Значениями этих атрибутов должны быть строки JavaScript-кода, который исполняется всегда, когда с HTML-элементом происходит указанное событие.

Объекты документа, доступные через такие коллекции, как `document.links`, обладают свойствами, соответствующими атрибутам HTML-тегов. Объект `Link`, например, имеет свойство `href`, которое соответствует атрибуту `href` тега `<a>`. То же самое относится и к обработчикам событий. Определить обработчик события `onclick` гиперссылки можно либо с помощью атрибута `onclick` тега `<a>`, либо установив значение свойства `onclick` объекта `Link`. В качестве другого примера рассмотрим атрибут `onsubmit` элемента `<form>`. В JavaScript у объекта `Form` есть соответствующее свойство `onsubmit`. (Помните, что язык HTML нечувствителен к регистру, и атрибуты могут быть записаны в нижнем, верхнем или смешанном регистре. В JavaScript имена всех свойств обработчиков событий должны быть записаны в нижнем регистре.)

В HTML обработчики событий определяются путем присваивания строки, содержащей JavaScript-код, атрибуту-обработчику события. В JavaScript они определяются путем присваивания функции свойству-обработчику события. Рассмотрим следующий тег `<form>` и его обработчик события `onsubmit`:

```
<form name="myform" onsubmit="return validateform();">...</form>
```

В JavaScript вместо строки JavaScript-кода, вызывающей функцию и возвращающей ее результат, можно непосредственно присвоить функцию свойству-обработчику события:

```
document.myform.onsubmit = validateform;
```

Обратите внимание: после имени функции не указаны скобки. Дело в том, что здесь мы не хотим вызывать функцию, а просто присваиваем ссылку на нее.

Полное описание такого способа назначения обработчиков событий приводится в главе 17.

15.3.3. Пример использования ранней версии DOM

В примере 15.1 приводится функция `listanchors()`, которая открывает новое окно и использует метод `document.write()` для вывода списка всех якорных элементов в оригинальном документе. Каждая запись в списке – это ссылка с обработчиком события, выполняющим прокрутку оригинального окна в позицию заданного якорного элемента. Программный код этого примера будет особенно полезен, если вы при создании своих HTML-документов вставляете заголовки разделов, отмеченные якорными элементами:

```
<a name="sect14.6"><h2>Объект Anchor</h2></a>
```

Обратите внимание: функция `listanchors()` пользуется методом `Window.open()`. Как было показано ранее, броузеры обычно блокируют всплывающие окна, если те не создаются в ответ на действия пользователя. Поэтому вызов `listanchors()` лучше вставить в обработчик события щелчка на кнопке или на ссылке, а не вызывать его автоматически по окончании загрузки страницы.

Пример 15.1. Список всех якорных элементов

```
/*
 * listanchors.js: Создает простое оглавление с помощью document.anchors[].
 *
 * Функция listanchors() получает документ в качестве аргумента и открывает
 * новое окно, которое выступает в роли "окна навигации" по этому документу.
 * Новое окно отображает список всех якорных элементов документа.
 * Щелчок на любой записи из списка вызывает прокрутку документа
 * в позицию заданного якорного элемента.
 */
function listanchors(d) {
    // Открыть новое окно
    var newwin = window.open("", "navwin",
                             "menubar=yes,scrollbars=yes,resizable=yes," +
                             "width=500,height=300");
    // Установить заголовок
    newwin.document.write("<h1>Окно навигации: " + d.title + "</h1>");

    // Перечислить все якорные элементы
    for(var i = 0; i < d.anchors.length; i++) {
        // Для каждого якорного элемента нужно получить текст для отображения
        // в списке. В первую очередь надо попытаться получить текст, расположенный
        // между тегами <a> и </a>, с помощью свойства, зависящего от типа броузера.
        // Если текст отсутствует, тогда использовать значение свойства name.
        var a = d.anchors[i];
        var text = null;
        if (a.text) text = a.text;                          // Netscape 4
        else if (a.innerText) text = a.innerText;           // IE 4+
        if ((text == null) || (text == '')) text = a.name; // По умолчанию

        // Теперь вывести этот текст в виде ссылки. Свойство href этой ссылки
        // использоваться не будет: всю работу выполняет обработчик события
        // onclick, он устанавливает свойство location.hash оригинального
        // окна, что вызывает прокрутку окна до заданного якорного элемента.
        // См. описание свойств Window.opener, Window.location,
        // Location.hash и Link.onclick.
        newwin.document.write('<a href="#' + a.name + '"' +
                              ' onclick="opener.location.hash=\'' + a.name +
                              '\'; return false;">');
        newwin.document.write(text);
        newwin.document.write('</a><br>');
    }
    newwin.document.close( ); // Никогда не забывайте закрывать документ!
}
```

15.4. Обзор объектной модели W3C DOM

Рассмотрев раннюю упрощенную модель DOM, теперь обратимся к более мощной и стандартизованной модели W3C DOM, пришедшей ей на смену. Программный интерфейс (API) модели W3C DOM не особенно сложен, но прежде чем перейти к рассмотрению DOM-программирования, необходимо уточнить несколько вещей относительно DOM-архитектуры.

15.4.1. Представление документов в виде дерева

HTML-документы имеют иерархическую структуру вложенных тегов, которая в DOM представлена в виде дерева объектов. Узлы дерева представляют различные типы содержимого документа. В первую очередь, древовидное представление HTML-документа содержит узлы, представляющие элементы или теги, такие как <body> и <p>, и узлы, представляющие строки текста. HTML-документ также может содержать узлы, представляющие HTML-комментарии.[1] Рассмотрим следующий простой HTML-документ:

```
<html>
  <head>
    <title>Sample Document</title>
  </head>
  <body>
    <h1>An HTML Document</h1>
    <p>This is a <i>simple</i> document.
  </body>
</html>
```

DOM-представление этого документа приводится на рис. 15.1.

Тем, кто еще не знаком с древовидными структурами в компьютерном программировании, полезно знать, что они заимствуют терминологию у генеалогических деревьев. Узел, расположенный непосредственно над данным узлом, назы-

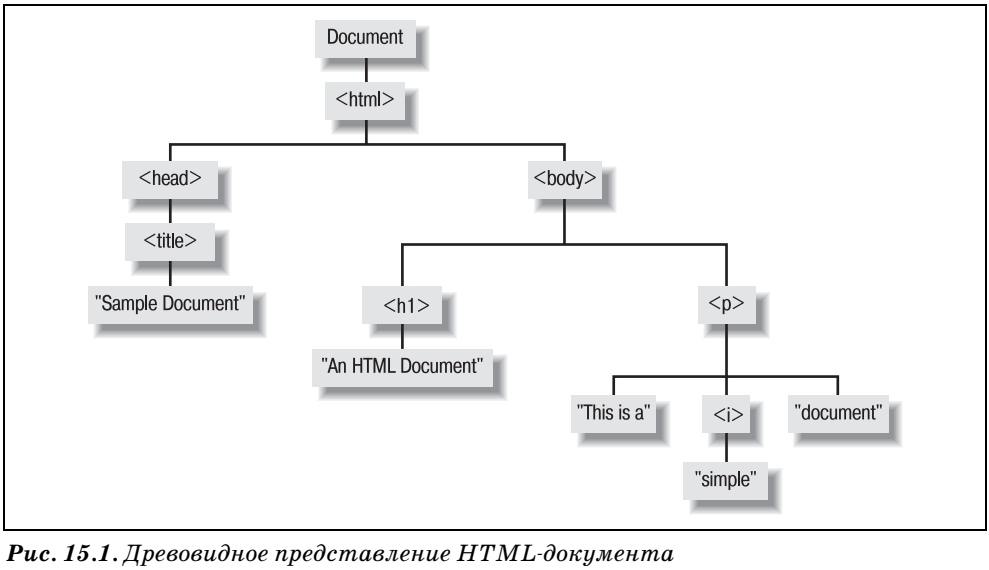

Рис. 15.1. *Древовидное представление HTML-документа*

[1] Модель DOM может также использоваться для представления XML-документов, которые имеют значительно более сложный синтаксис, чем HTML-документы. Древовидное представление таких документов может содержать узлы, являющиеся ссылками на XML-сущности, инструкции по обработке, разделы CDATA и пр. Дополнительные сведения об использовании DOM с XML-документами можно найти в главе 21.

вается *родительским* по отношению к данному узлу. Узлы, расположенные на один уровень ниже другого узла, являются *дочерними* по отношению к данному узлу. Узлы, находящиеся на том же уровне и имеющие того же родителя, называются *братьями*. Узлы, расположенные на любое число уровней ниже другого узла, являются его *потомками*. Родительские, прародительские и любые другие узлы, расположенные выше данного узла, являются его *предками*.

15.4.2. Узлы

Древовидная структура DOM, изображенная на рис. 15.1, представляет собой дерево объектов `Node` различных типов. Интерфейс `Node`[1] определяет свойства и методы для перемещения по дереву и манипуляций им. Свойство `childNodes` объекта `Node` возвращает список дочерних узлов, свойства `firstChild`, `lastChild`, `nextSibling`, `previousSibling` и `parentNode` предоставляют средство обхода узлов дерева. Такие методы, как `appendChild()`, `removeChild()`, `replaceChild()` и `insertBefore()`, позволяют добавлять узлы в дерево документа и удалять их. Далее в этой главе мы встретим примеры применения этих свойств и методов.

15.4.2.1. Типы узлов

Типы узлов в дереве документа представлены специальными подынтерфейсами интерфейса `Node`. У любого объекта `Node` есть свойство `nodeType`, определяющее тип данного узла. Если свойство `nodeType` узла равно, например, константе `Node.ELEMENT_NODE`, значит, объект `Node` является также объектом `Element`, и можно использовать с ним все методы и свойства, определенные интерфейсом `Element`. В табл. 15.1 перечислены чаще всего встречающиеся в HTML-документах типы узлов и значения `nodeType` для каждого из них.

Таблица 15.1. Основные типы узлов

Интерфейс	Константа nodeType	Значение nodeType
Element	`Node.ELEMENT_NODE`	1
Text	`Node.TEXT_NODE`	3
Document	`Node.DOCUMENT_NODE`	9
Comment	`Node.COMMENT_NODE`	8
DocumentFragment	`Node.DOCUMENT_FRAGMENT_NODE`	11
Attr	`Node.ATTRIBUTE_NODE`	2

Корневым узлом DOM-дерева является объект `Document`. Свойство `documentElement` этого объекта ссылается на объект `Element`, представляющий корневой элемент документа. Для HTML-документов это тег `<html>`, явно или неявно присутствующий в документе. (Помимо корневого элемента узел `Document` может иметь другие дочерние элементы, такие как объекты `Comment`.) В HTML-документах, как правило,

[1] Стандарт DOM определяет интерфейсы, а не классы. Те, кто не знаком с термином «интерфейс» в объектно-ориентированном программировании, могут рассматривать его как абстрактный класс. Позднее в этом обзоре модели DOM я более подробно объясню различия между классом и интерфейсом.

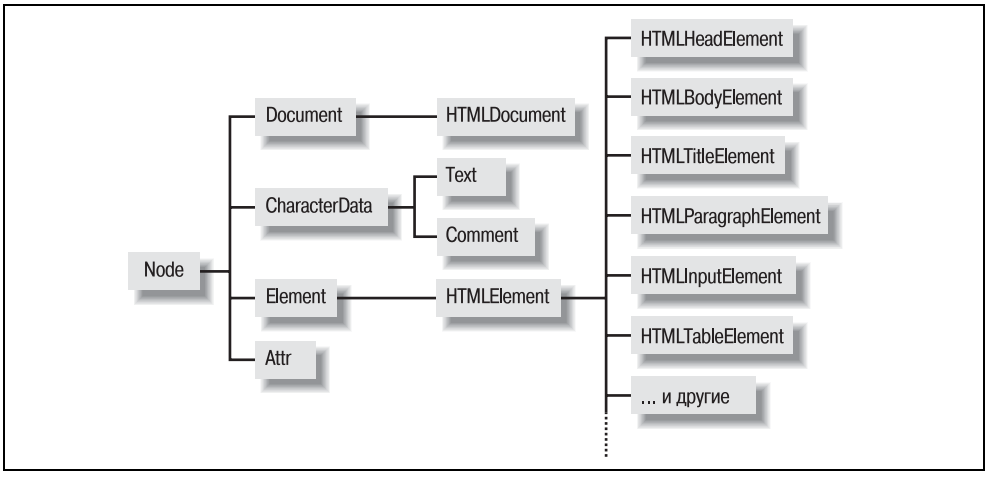

***Рис. 15.2.** Неполная иерархия классов DOM API*

наибольший интерес представляет элемент `<body>`, а не `<html>`, потому для удобства можно пользоваться свойством `document.body` для ссылки на этот элемент.

В DOM-дереве существует лишь один объект `Document`. Большинство узлов дерева – это объекты `Element`, которые представляют такие теги, как `<html>` и `<i>`, а также объекты `Text`, представляющие текстовые строки. Если в документе имеются комментарии, синтаксический анализатор сохраняет их в DOM-дереве в виде объектов `Comment`. На рис. 15.2 приводится неполная иерархия классов для этих и других базовых DOM-интерфейсов.

15.4.2.2. Атрибуты

Атрибуты элемента (такие как `src` и `width` тега `<img>`) могут быть прочитаны, установлены и удалены с помощью методов `getAttribute()`, `setAttribute()` и `removeAttribute()` интерфейса `Element`. Как уже говорилось, стандартные атрибуты HTML-тегов доступны в виде свойств узлов `Element`, представляющих эти теги.

Другой менее удобный способ работы с атрибутами предлагает метод `getAttributeNode()`, который возвращает объект `Attr`, представляющий атрибут и его значение. (Одной из причин выбора этой менее удобной технологии является наличие у интерфейса `Attr` свойства `specified`, позволяющего определять, указан ли данный атрибут в документе явно или для него принимается значение по умолчанию.) Интерфейс `Attr` на рис. 15.2 – отдельный тип узла. Однако следует отметить, что объекты `Attr` отсутствуют в массиве `childNodes[]` элемента и непосредственно не являются частью дерева документа, как узлы `Element` и `Text`. Спецификация DOM позволяет обращаться к узлам `Attr` через массив `attributes[]` интерфейса `Node`, но в Internet Explorer определяется другой несовместимый массив `attributes[]`, что делает невозможным использование этого массива переносимым образом.

15.4.3. DOM HTML API

Стандарт DOM предназначен для работы как с XML-, так и с HTML-документами. Базовый программный интерфейс (API) модели DOM, к которому относятся

интерфейсы Node, Element, Document и другие, относительно универсален и применим к обоим типам документов. Стандарт DOM также включает интерфейсы, специфические для HTML-документов. Как видно на рис. 15.2, HTMLDocument — это специфический для HTML подынтерфейс интерфейса Document, а HTMLElement — специфический для HTML подынтерфейс интерфейса Element. Кроме того, DOM определяет интерфейсы для многих HTML-элементов, относящиеся к конкретным тегам. Эти интерфейсы, такие как HTMLBodyElement и HTMLTitleElement, обычно определяют набор свойств, отражающих атрибуты данного HTML-тега.

Интерфейс HTMLDocument определяет различные свойства документа и методы, поддерживавшиеся броузерами до появления стандарта W3C. В их число входят свойство location, массив forms[] и метод write(), описанные в этой главе ранее.

Интерфейс HTMLDocument определяет свойства id, style, title, lang, dir и className. Эти свойства обеспечивают удобный доступ к значениям атрибутов id, style, title, lang, dir и className, которыми обладают все HTML-теги. (В языке JavaScript слово «class» зарезервировано, поэтому атрибут class в языке JavaScript стал свойством className.) HTML-теги из табл. 15.2 не принимают никаких атрибутов, кроме шести только что перечисленных, и потому полностью представимы интерфейсом HTMLElement.

Таблица 15.2. Простые HTML-теги

<abbr>	<acronym>	<address>	<b>	<bdo>
<big>		<cite>	<code>	<dd>
<dfn>	<dt>		<i>	<kbd>
<noframes>	<noscript>	<s>	<samp>	<small>
	<strike>		<sub>	<sup>
<tt>	<u>	<var>		

Для всех остальных HTML-тегов в части спецификации DOM, относящейся к HTML, определяются специальные интерфейсы. Для многих HTML-тегов эти интерфейсы не делают ничего, кроме предоставления набора свойств, соответствующих HTML-атрибутам. Например, тегу <ul> соответствует интерфейс HTMLUListElement, а для тега <body> есть соответствующий интерфейс HTMLBodyElement. Поскольку эти интерфейсы просто определяют свойства, стандартизованные в HTML, они не документируются в этой книге подробно. Можно спокойно предположить, что объект HTMLElement, представляющий определенный HTML-тег, имеет свойства для каждого из стандартных атрибутов этого тега (соглашения о назначении имен приводятся в следующем разделе).

Примечательно, что стандарт DOM описывает свойства HTML-атрибутов для удобства создателей сценариев. Универсальный способ чтения и установки значений атрибутов предоставляют методы getAttribute() и setAttribute() объекта Element. При работе с атрибутами, которые не являются частью стандартного языка HTML, обязательно должны использоваться эти методы.

Некоторые из интерфейсов, описанных в HTML DOM, определяют дополнительные свойства или методы, отличные от тех, которые соответствуют значениям HTML-атрибутов. Например, интерфейс HTMLInputElement определяет методы focus() и blur(), а также свойство form, а интерфейс HTMLFormElement — методы submit()

и reset(), а также свойство length. Если представление HTML-элемента в JavaScript включает в себя свойства или методы, которые просто являются отражением HTML-атрибутов, такие элементы описываются в четвертой части книги. Однако следует отметить, что в справочном разделе не используются длинные имена, определяемые DOM. Вместо этого с целью упрощения (и сохранения обратной совместимости) эти элементы представлены под более короткими именами, например Anchor, Image, Input, Form, Link, Option, Select, Table или Textarea.

15.4.3.1. Соглашения об именовании для HTML

При работе со специфическими для HTML частями стандарта DOM необходимо иметь в виду некоторые простые соглашения о назначении имен. Прежде всего, следует помнить, что язык HTML нечувствителен к регистру символов, а в JavaScript прописные и строчные символы различаются. Имена свойств, специфических для HTML-интерфейсов, начинаются со строчных букв. Если имя свойства состоит из нескольких слов, первые буквы второго и последующих слов являются прописными. Таким образом, атрибут maxlength тега <input> транслируется в свойство maxLength интерфейса HTMLInputElement.

Когда имя HTML-атрибута конфликтует с ключевым словом JavaScript, для разрешения конфликта к имени добавляется префикс «html». Например, атрибут for тега <label> транслируется в свойство htmlFor интерфейса HTMLLabelElement. Исключение из этого правила составляет атрибут class (который может быть указан для любого HTML-элемента) – он транслируется в свойство className интерфейса HTMLElement.[1]

15.4.4. Уровни и возможности DOM

Имеются две версии, или два «уровня», стандарта DOM. Модель DOM уровня 1 (DOM Level 1) была стандартизована в октябре 1998 года. Она определяет базовые DOM-интерфейсы, такие как Node, Element, Attr и Document, а также различные интерфейсы, специфические для HTML. Модель DOM уровня 2 (DOM Level 2) была стандартизована в ноябре 2000 года. Помимо некоторых изменений в базовых интерфейсах, эта версия DOM была сильно расширена за счет определения стандартных программных интерфейсов (API) для работы с событиями документа и каскадными таблицами стилей (CSS), а также с целью предоставления дополнительных инструментальных средств для работы с непрерывными областями документов.

Стандарт DOM Level 2 стал модульным. Модуль Core, определяющий основную древовидную структуру документа с помощью (среди прочих) интерфейсов Document, Node, Element и Next, – это единственный обязательный модуль. Все остальные модули не обязательны и могут либо поддерживаться, либо нет в зависимости от реализации. Реализация DOM в веб-броузере, очевидно, должна поддерживать модуль HTML, т. к. веб-документы пишутся на языке HTML. Броузеры, поддерживающие таблицы CSS-стилей, обычно поддерживают и модули StyleSheets и CSS, поскольку (как мы увидим в главе 16) CSS-стили играют ключевую

[1] Имя className обманчиво, т. к. помимо указания имени одного класса это свойство (и представляемый им HTML-атрибут) может содержать список имен классов, разделенных пробелами.

роль в DHTML-программировании. Аналогично, поскольку большинство интересных JavaScript-программ требует средств обработки событий, можно предполагать поддержку веб-броузерами модуля Events спецификации DOM. К сожалению, модуль Events лишь недавно был реализован в Microsoft Internet Explorer, и как это будет описано в главе 17, обработка событий в ранней версии DOM, в W3C DOM и в IE DOM выполняется по-разному.

В данной книге описываются модели DOM Level 1 и DOM Level 2; соответствующий справочный материал вы найдете в IV части книги.

В W3C продолжаются работы над расширением стандарта DOM, и были выпущены спецификации уровня 3 (Level 3) для некоторых модулей, включая версию модуля Core. Функциональные возможности, определяемые в модели DOM Level 3, практически не используются в веб-броузерах (хотя в Firefox частичная поддержка имеется) и в этом издании книги не описываются.

Кроме того, иногда вы можете встретить упоминание о модели DOM Level 0. Этот термин не относится к какому-либо формальному стандарту, а служит для неформальной ссылки на общие средства объектных моделей документа, реализованных в Netscape и Internet Explorer до стандартизации консорциумом W3C. То есть термин «DOM Level 0» является синонимом термина «ранняя версия DOM».

15.4.5. Соответствие модели DOM

На момент написания этой книги последние версии современных броузеров, таких как Firefox, Safari и Opera, прекрасно поддерживали стандарт DOM Level 2. Броузер Internet Explorer 6 в основном совместим со стандартом DOM Level 1, и практически не поддерживает стандарт DOM Level 2. Кроме того, из-за неполной поддержки модуля Core Level 2 в нем вообще не поддерживается модуль Events Level 2, который будет обсуждаться в главе 17. Броузеры Internet Explorer 5 и 5.5 имеют значительные пробелы в совместимости, но достаточно хорошо поддерживают ключевые методы стандарта DOM Level 1, чтобы запускать большинство примеров из этой главы.

Количество доступных броузеров теперь слишком велико, а изменения в сфере поддержки стандартов происходят слишком быстро, чтобы даже пытаться в этой книге определенно утверждать, какие средства DOM поддерживает тот или иной броузер. Следовательно, чтобы определить степень соответствия реализации любого конкретного броузера модели DOM, вам придется полагаться на другие источники информации.

Одним из источников информации о соответствии является сама реализация. В «правильной» реализации свойство implementation объекта Document ссылается на объект DOMImplementation, определяющий метод с именем hasFeature(). Посредством этого метода (если он существует) можно получить сведения о поддержке определенного модуля (или характристики) стандарта DOM. Например, определить, поддерживает ли реализация DOM в веб-броузере базовые интерфейсы стандарта DOM Level 1 для работы с HTML-документами, можно с помощью следующего фрагмента:

```
if (document.implementation &&
    document.implementation.hasFeature &&
    document.implementation.hasFeature("html", "1.0")) {
```

```
    // Броузер заявляет о поддержке интерфейсов Core и HTML уровня 1
}
```

Метод `hasFeature()` принимает два аргумента: первый – это имя проверяемого модуля, второй – номер версии в виде строки. Он возвращает `true`, если указанная версия данного модуля поддерживается. В табл. 15.3 перечислены пары «название/номер версии», определенные в стандартах DOM Level 1 и Level 2. Обратите внимание: названия модулей нечувствительны к регистру, поэтому вполне допустимо чередовать прописные и строчные символы в их именах. В четвертом столбце таблицы указано, какие модули требуются для поддержки данного модуля, и следовательно, их наличие подразумевается в случае возвращения методом значения `true`. Например, если метод `hasFeature()` показал, что поддерживается модуль `MouseEvents`, это подразумевает также, что поддерживается модуль `UIEvents`, что, в свою очередь, подразумевает поддержку модулей `Events`, `Views` и `Core`.

Таблица 15.3. Модули, совместимость с которыми можно проверить методом hasFeature( )

Название модуля	Версия	Описание	Подразумевает поддержку
`HTML`	1.0	Интерфейсы `Core` и `HTML` уровня 1	
`XML`	1.0	Интерфейсы `Core` и `XML` уровня 1	
`Core`	2.0	Интерфейсы `Core` уровня 2	
`HTML`	2.0	Интерфейсы `HTML` уровня 2	`Core`
`XML`	2.0	Интерфейсы `XML` уровня 2	`Core`
`Views`	2.0	Интерфейс `AbstractView`	`Core`
`StyleSheets`	2.0	Универсальный обход таблицы стилей	`Core`
`CSS`	2.0	CSS-стили	`Core`, `Views`
`CSS2`	2.0	Интерфейс `CSS2Properties`	`CSS`
`Events`	2.0	Инфраструктура для обработки событий	`Core`
`UIEvents`	2.0	События пользовательского интерфейса (плюс модули `Events` и `Views`)	`Events`, `Views`
`MouseEvents`	2.0	События мыши	`UIEvents`
`HTMLEvents`	2.0	HTML-события	`Events`

В Internet Explorer 6 метод `hasFeature()` возвращает `true` только для модуля `HTML` и версии 1.0. Он не сообщает о соответствии любым другим модулям, перечисленным в табл. 15.3 (хотя, как мы увидим в главе 16, он поддерживает большинство основных применений модуля `CSS2`).

В этой книге документируются интерфейсы, составляющие все DOM-модули, перечисленные в табл. 15.3. Модули `Core` и `HTML` рассматриваются в этой главе, модули `StyleSheets`, `CSS` и `CSS2` – в главе 16, а различные модули, относящиеся к событиям, – в главе 17. Четвертая часть этой книги содержит полное описание всех модулей.

Информации, возвращаемой методом hasFeature(), не всегда можно доверять. Как отмечалось ранее, IE 6 сообщает о соответствии средств HTML уровню 1, хотя в этом соответствии есть некоторые проблемы. В то же время Netscape 6.1 сообщает о несоответствии модулю Core Level 2, хотя этот броузер почти совместим с этим модулем. В обоих случаях нужны более подробные сведения о том, что именно совместимо, а что нет. Однако объем этой информации слишком велик, и она слишком изменчива, чтобы включать ее в печатное издание.

Те, кто активно занимается веб-разработкой, несомненно, уже знают или скоро узнают о многих специфических для броузеров деталях совместимости. Кроме того, в Интернете есть ресурсы, которые могут оказаться полезными. Организация W3C выпустила набор тестов (правда, не совсем полный) для проверки степени поддержки некоторых DOM-модулей, доступный на сайте *http://www.w3c.org/DOM/Test/*. К сожалению, результаты этих тестов для наиболее распространенных броузеров опубликованы не были.

Возможно, в поисках информации о совместимости и соответствии стандартам лучше обратиться к независимым сайтам в Интернете. Один из достойных упоминания сайтов – *http://www.quirksmode.org*; его поддерживает Петер-Пауль Кох (Peter-Paul Koch). Он опубликовал результаты обширных исследований о соответствии броузеров стандартам CSS и DOM. Еще один замечательный сайт – *http://webdevout.net/browser_support.php*; он поддерживается Дэвидом Хаммондом (David Hammond).

15.4.5.1. Соответствие модели DOM броузера Internet Explorer

Поскольку IE – наиболее широко используемый веб-броузер, несколько особых замечаний о его соответствии спецификациям DOM будут здесь вполне уместны. IE 5 и более поздние версии достаточно хорошо поддерживают модули Core и HTML Level 1, чтобы запускать примеры из этой главы, а также ключевые особенности модуля CSS Level 2, чтобы запускать большинство примеров из главы 16. К сожалению, IE версий 5, 5.5 и 6 не поддерживает модуль Events модели DOM Level 2, хотя корпорация Microsoft участвовала в определении этого модуля и имела достаточно времени для его реализации в IE 6. Отсутствие в IE поддержки стандартной модели обработки событий затрудняет создание развитых клиентских веб-приложений.

Хотя IE 6 заявляет (через свой метод hasFeature()) о поддержке интерфейсов Core и HTML стандарта DOM Level 1, фактически эта поддержка неполна. Наиболее вопиющая проблема, с которой вы, скорее всего, столкнетесь, – небольшая, но неприятная: IE не поддерживает константы типов узлов, определяемых в интерфейсе Node. Вспомните, что каждый узел в документе имеет свойство nodeType, задающее тип данного узла. Спецификация DOM также утверждает, что интерфейс Node определяет константы, представляющие каждый из определяемых им типов узлов. Например, константа Node.ELEMENT_NODE представляет узел Element. В IE (по крайней мере, до версии 6 включительно) эти константы просто не существуют.

В этой главе примеры изменены так, чтобы обойти это препятствие. Они содержат целочисленные литералы вместо соответствующих символических констант. Например:

```
if (n.nodeType == 1 /*Node.ELEMENT_NODE*/) // Проверяем, является ли n объектом Element
```

Хороший стиль программирования требует, чтобы в программный код помещались константы, а не жестко определенные целочисленные литералы, и те, кто захочет сделать код переносимым, могут включить в программу следующий код для определения констант, если они отсутствуют:

```
if (!window.Node) {
    var Node = {           // Если объект Node отсутствует, определяем
        ELEMENT_NODE: 1,   // его со следующими свойствами и значениями.
        ATTRIBUTE_NODE: 2, // Обратите внимание, здесь только типы HTML-узлов
        TEXT_NODE: 3,      // Для XML-узлов вам нужно определить
        COMMENT_NODE: 8,   // другие константы.
        DOCUMENT_NODE: 9,
        DOCUMENT_FRAGMENT_NODE: 11
    }
}
```

15.4.6. Независимые от языка DOM-интерфейсы

Хотя стандарт DOM появился благодаря желанию иметь общий прикладной интерфейс (API) для DHTML-программирования, модель DOM интересна не только веб-программистам. На самом деле сейчас этот стандарт наиболее интенсивно используется серверными программами на языках Java и C++ для анализа XML-документов и манипуляции ими. Из-за наличия разнообразных вариантов применения стандарт DOM был определен как независимый от языка. В данной книге описывается только привязка DOM API к JavaScript, но необходимо иметь в виду и некоторые другие моменты. Во-первых, следует отметить, что свойства объекта в привязке к JavaScript обычно соответствуют паре методов `get/set` в привязке к другим языкам. Следовательно, когда программист, пишущий на Java, спрашивает вас о методе `getFirstChild()` интерфейса `Node`, надо понимать, что в JavaScript привязка `Node` API не определяет метода `getFirstChild()`. Вместо этого она просто определяет свойство `firstChild`, и чтение этого свойства в JavaScript эквивалентно вызову метода `getFirstChild()` в Java.

Другая важная особенность привязки DOM API к JavaScript в том, что некоторые DOM-объекты ведут себя как JavaScript-массивы. Если интерфейс определяет метод с именем `item()`, то объекты, реализующие этот интерфейс, ведут себя так же, как доступные только для чтения массивы с числовым индексом. Предположим, что в результате чтения свойства `childNodes` узла получен объект `NodeList`. Отдельные объекты `Node` из списка можно получить двумя способами: во-первых, передав номер нужного узла методу `item()`, и во-вторых, рассматривая объект `NodeList` как массив и обращаясь к нему по индексу. Следующий код иллюстрирует эти две возможности:

```
var n = document.documentElement; // Это объект Node.
var children = n.childNodes;      // Это объект NodeList.
var head = children.item(0);      // Это один из способов использования NodeList.
var body = children[1];           // Но есть более простой способ!
```

Аналогично, если у DOM-объекта есть метод `namedItem()`, передача строки этому методу означает то же самое, что использование строки в качестве индекса массива. Например, следующие строки кода представляют собой эквивалентные средства доступа к элементу формы:

```
var f = document.forms.namedItem("myform");
var g = document.forms["myform"];
var h = document.forms.myform;
```

Несмотря на то что существует возможность обращаться к элементам объекта `NodeList` с использованием нотации массивов, важно помнить, что `NodeList` – это всего лишь объект, подобный массиву, а не настоящий массив (см. раздел 7.8). Объект `NodeList`, например, не имеет метода `sort()`.

Стандарт DOM может использоваться различными способами, поэтому разработчики стандарта определили DOM API таким образом, чтобы не ограничивать возможность реализации API другими разработчиками. В частности, стандарт DOM определяет интерфейсы вместо классов. В объектно-ориентированном программировании класс – это фиксированный тип данных, который должен быть реализован в точном соответствии со своим определением. В то же время интерфейс – это коллекция методов и свойств, которые должны быть реализованы вместе. Следовательно, реализация DOM может определять любые классы, которые считает нужным, но эти классы должны определять методы и свойства различных DOM-интерфейсов.

Такая архитектура имеет несколько важных следствий. Во-первых, имена классов в реализации могут не соответствовать напрямую именам интерфейсов в стандарте DOM (и в этой книге). Во-вторых, один класс может реализовывать более одного интерфейса. Рассмотрим, например, объект `Document`. Этот объект является экземпляром некоторого класса, определенного реализацией веб-броузера. Мы не знаем, какой именно это класс, но знаем, что он реализует интерфейс `Document`; т. е. все методы и свойства, определенные интерфейсом `Document`, доступны нам через объект `Document`. Поскольку веб-броузеры работают с HTML-документами, мы также знаем, что объект `Document` реализует интерфейс `HTMLDocument`, и нам доступны все методы и свойства, определенные этим интерфейсом. Кроме того, если веб-броузер поддерживает CSS и реализует DOM-модуль `CSS`, значит, объект `Document` реализует также DOM-интерфейсы `DocumentStyle` и `DocumentCSS`. И если веб-броузер поддерживает модули `Events` и `Views`, объект `Document` реализует также интерфейсы `DocumentEvent` и `DocumentView`.

Вообще, в IV части книги основное внимание уделяется описанию объектов, с которыми сталкиваются JavaScript-программисты, а не более абстрактных интерфейсов, определяющих API этих объектов. Таким образом, в четвертой части книги со справочными материалами можно найти разделы с описанием объектов `Document` и `HTMLDocument`, но там отсутствуют описания дополнительных интерфейсов, таких как `DocumentCSS` или `DocumentView`. Описания методов, определяемых этими интерфейсами, просто вставлены в раздел с описанием объекта `Document`.

Важно также понимать, что т. к. стандарт DOM определяет интерфейсы, а не классы, он не описывает никаких методов-конструкторов. Если, например, требуется создать новый объект `Text` для вставки в документ, то нельзя просто написать:

```
var t = new Text("это новый текстовый узел"); // Такого конструктора нет!
```

Стандарт DOM не может определять конструкторы, но он определяет в интерфейсе `Document` несколько полезных *методов-фабрик* (*factory methods*) для создания объектов. То есть чтобы создать новый узел `Text` в документе, надо написать:

```
var t = document.createTextNode("это новый текстовый узел");
```

Методы-фабрики, определенные в DOM, имеют имена, которые начинаются со слова «create». Помимо методов-фабрик, определяемых интерфейсом `Document`, несколько таких методов определяется интерфейсом `DOMImplementation` и доступно через свойство `document.implementation`.

15.5. Обход документа

Рассмотрев положения стандарта W3C DOM, можно приступать к использованию DOM API. В этом и следующих разделах демонстрируется, как можно организовать обход дерева элементов документа, а также изменять содержимое документа и добавлять новое содержимое.

Как уже отмечалось, DOM представляет HTML-документ в виде дерева объектов `Node`. Для любой древовидной структуры наиболее частое выполняемое действие – это обход дерева с поочередным просмотром каждого узла. Один из способов показан в примере 15.2. Это JavaScript-функция, рекурсивно просматривающая узел и все дочерние узлы и подсчитывающая количество HTML-тегов (т. е. узлов `Element`), встреченных в процессе обхода. Обратите внимание на свойство `childNodes` текущего узла. Значением этого свойства является объект `NodeList`, ведущий себя (в JavaScript) как массив объектов `Node`. Поэтому функция может перечислить все дочерние узлы данного узла путем циклического перебора элементов массива `childNodes[]`. Функция рекурсивно перечисляет не только все дочерние узлы данного узла, но и все узлы в дереве узлов. Обратите внимание, что эта функция демонстрирует также применение свойства `nodeType` для определения типа каждого узла.

Пример 15.2. Обход узлов документа

```
<head>
<script>
// Этой функции передается DOM-объект Node. Функция проверяет, представляет
// ли этот узел HTML-тег, т. е. является ли узел объектом Element. Она рекурсивно
// вызывает сама себя для каждого дочернего узла, проверяя их таким же образом.
// Функция возвращает общее число найденных ею объектов Element. Если вы вызываете
// эту функцию, передавая ей DOM-объект, она выполнит обход всего DOM-дерева.
function countTags(n) {                          // n - это Node
    var numtags = 0;                             // Инициализируем счетчик тегов
    if (n.nodeType == 1 /*Node.ELEMENT_NODE*/) // Проверяем, является ли n
                                                 // объектом Element
        numtags++;                               // Если это так, увеличиваем счетчик
    var children = n.childNodes;                // Теперь получаем все дочерние элементы n
    for(var i=0; i < children.length; i++) {    // Цикл по всем дочерним элементам
        numtags += countTags(children[i]);      // Рекурсия по всем дочерним элементам
    }
    return numtags;                              // Возвращаем общее число тегов
}
</script>
</head>
<!-- Это пример использования функции countTags() -->
<body onload="alert('Количество тегов в документе: ' + countTags(document))">
Это <i>пример</i> документа.
</body>
```

Обратите внимание: определенная в примере 15.2 функция `countTags()` вызывается из обработчика события `onload`, поэтому она вызвана не будет, пока документ не загрузится целиком. Это обязательное требование при работе с DOM: нельзя обходить дерево документа или манипулировать им до тех пор, пока документ полностью не загружен. (В разделе 13.5.7 детально обсуждаются причины этого ограничения. Дополнительно в примере 17.7 приводится функция, которая позволяет регистрировать обработчики события `onload` нескольких модулей.)

В дополнение к свойству `childNodes` интерфейс `Node` определяет несколько других полезных свойств. Свойства `firstChild` и `lastChild` ссылаются на первый и последний дочерние узлы, а свойства `nextSibling` и `previousSibling` — на ближайшие смежные узлы. (Два узла называются смежными, если имеют один и тот же родительский узел.) Эти свойства предоставляют еще один способ обхода дочерних узлов, который демонстрируется в примере 15.3. В этом примере приводится определение функции `getText()`, которая отыскивает все узлы `Text`, вложенные в указанный узел. Она извлекает и объединяет текстовое содержимое узлов и возвращает полученный результат в виде JavaScript-строки. Потребность в такой функции при программировании с использованием DOM-интерфейсов возникает на удивление часто.

Пример 15.3. Получение текстового содержимого из всех вложенных DOM-узлов

```
/**
 * getText(n): Отыскивает все узлы Text, вложенные в узел n.
 * Объединяет х содержимое и возвращает результат в виде строки.
 */
function getText(n) {
    // Операция объединения строк очень ресурсоемка, потому сначала
    // содержимое текстовых узлов помещается в массив, затем выполняется
    // операция конкатенации элементов массива в одну строку.
    var strings = [];
    getStrings(n, strings);
    return strings.join("");

    // Эта рекурсивная функция отыскивает все текстовые узлы
    // и добавляет их содержимое в конец массива.
    function getStrings(n, strings) {
        if (n.nodeType == 3 /* Node.TEXT_NODE */)
            strings.push(n.data);
        else if (n.nodeType == 1 /* Node.ELEMENT_NODE */) {
            // Обратите внимание, обход выполняется
            // с использованием firstChild/nextSibling
            for(var m = n.firstChild; m != null; m = m.nextSibling) {
                getStrings(m, strings);
            }
        }
    }
}
```

15.6. Поиск элементов в документе

Возможность обхода всех узлов в дереве документа дает нам средство поиска определенных узлов. При программировании с использованием DOM API довольно

часто возникает задача получения определенного узла из документа или списка узлов определенного типа. К счастью, DOM API предоставляет функции, облегчающие решение этой задачи.

Объект Document является корневым элементом для всего DOM-дерева, но он не представляет ни один из HTML-элементов в этом дереве. Свойство document.documentElement ссылается на тег <html>, который выступает в роли корневого элемента документа. Свойство document.body соответствует тегу <body>, который в большинстве случаев представляет больший интерес, чем его родительский тег <html>.

Свойство body объекта Document представляет собой особое удобное свойство, через котрое предпочтительно обращаться к тегу <body> HTML-документа. Однако при отсутствии такого специального свойства мы могли бы обратиться к тегу <body> следующим образом:

```
document.getElementsByTagName("body")[0]
```

Это выражение вызывает метод getElementsByTagName() и выбирает первый элемент полученного массива. Вызов getElementsByTagName() возвращает массив всех элементов <body> в документе. HTML-документы могут содержать только один тег <body>, поэтому мы знаем, что нас интересует первый элемент полученного массива.[1]

Метод getElementsByTagName() может использоваться для получения списка HTML-элементов любого типа. Чтобы, например, найти все таблицы в документе, необходимо сделать следующее:

```
var tables = document.getElementsByTagName("table");
alert("Количество таблиц в документе: " + tables.length);
```

Обратите внимание: поскольку HTML-теги нечувствительны к регистру, строки, передаваемые в getElementsByTagName(), также нечувствительны к регистру. То есть предыдущий код находит теги <table>, даже если в коде они выглядят как <TABLE>. Метод getElementsByTagName() возвращает элементы в том порядке, в котором они расположены в документе. И наконец, если передать методу getElementsByTagName() специальную строку "*", он вернет список всех элементов в порядке их присутствия в документе. (Этот особый вариант не поддерживается в IE 5 и 5.5. См. описание специфического для IE массива Document.all[] в IV части книги.)

Иногда требуется получить не список элементов, а один конкретный элемент документа. Если о структуре документа известно многое, то можно прибегнуть к методу getElementsByTagName(). Так, сделать что-то с четвертым абзацем документа можно при помощи следующего кода:

```
var myParagraph = document.getElementsByTagName("p")[3];
```

Однако, как правило, это не самый лучший (и не самый эффективный) прием, поскольку он в значительной степени зависит от структуры документа – вставка нового абзаца в начало документа нарушит работу кода. Когда требуется манипули-

[1] Формально метод getElementsByTagName() возвращает подобный массиву объект NodeList. В этой книге для обращения к объектам NodeList используется нотация массивов, и неформально я буду называть их массивами.

ровать определенными элементами документа, лучше идти другим путем и определить для этих элементов атрибут id, задающий уникальное (в пределах документа) имя элемента. Тогда элемент можно будет найти по его идентификатору. Например, отметить специальный четвертый абзац документа тегом можно так:

```
<p id="specialParagraph">
```

Теперь легко найти узел этого абзаца посредством следующего JavaScript-кода:

```
var myParagraph = document.getElementById("specialParagraph");
```

Обратите внимание: метод getElementById() не возвращает массив элементов, как метод getElementsByTagName(). Так как значение каждого атрибута id является (или предполагается) уникальным, getElementById() возвращает только один элемент с соответствующим атрибутом id.

Метод getElementById() достаточно важен и довольно часто применяется в DOM-программировании. Обычно он служит для определения вспомогательной функции с более коротким именем:

```
// Если x - это строка, предполагается, что это идентификатор элемента
// и требуется отыскать этот элемент.
// В противном случае предполагается, что x - это уже элемент,
// поэтому нужно просто вернуть его.
function id(x) {
    if (typeof x == "string") return document.getElementById(x);
    return x;
}
```

С помощью аналогичных функций можно реализовать такие методы манипулирования DOM-деревом, которые будут принимать в качестве аргументов элементы или идентификаторы элементов. Для каждого такого аргумента x перед его использованием достаточно будет записать x = id(x). Один широко известный набор инструментальных средств для применения в сценариях[1], написанных на клиентском JavaScript, определяет подобный этому вспомогательный метод, имеющий еще более короткое имя – $().

Оба метода, getElementById() и getElementsByTagName(), относятся к методам объекта Document. Однако объект Element также определяет метод getElementsByTagName(). Этот метод объекта Element ведет себя так же, как метод объекта Document, за исключением того, что возвращает только элементы, являющиеся потомками того элемента, для которого он вызван. Благодаря этому можно, например, сначала использовать метод getElementById() для поиска определенного элемента, а затем – метод getElementsByTagName() для поиска всех потомков данного типа внутри найденного тега, например:

```
// Ищет определенный элемент Table внутри документа
// и подсчитывает количество строк в таблице.
var tableOfContents = document.getElementById("TOC");
var rows = tableOfContents.getElementsByTagName("tr");
var numrows = rows.length;
```

[1] Имеется в виду библиотека Prototype, разработанная Сэмом Стефенсоном (Sam Stephenson) и доступная на сайте *http://prototype.conio.net*.

И наконец, следует отметить, что для HTML-документов объект HTMLDocument определяет также метод getElementsByName(). Этот метод похож на getElementById(), но выполняет поиск элементов по атрибуту name, а не по атрибуту id. Кроме того, поскольку атрибут name не обязательно уникален в пределах документа (например, группы переключателей в HTML-формах обычно имеют одинаковый атрибут name), getElementsByName() возвращает массив элементов, а не одиночный элемент. Пример:

```
// Ищем тег <a name="top">
var link = document.getElementsByName("top")[0];
// Ищем все элементы <input type="radio" name="shippingMethod">
var choices = document.getElementsByName("shippingMethod");
```

В дополнение к выбору элементов по названию тега и идентификатору очень часто бывает удобно иметь возможность отбирать элементы по принадлежности к определенному классу. Атрибуту class в HTML и соответствующему ему свойству className в JavaScript можно присваивать одно или более имен классов (разделенных пробелами). Эти классы предназначены для использования совместно с таблицами CSS-стилей (подробнее об этом см. в главе 16), но это – не единственное их предназначение. Предположим, что в HTML-документ вставляются важные предупреждения, например следующим образом:

```
<div class="warning">
Это предупреждение
</div>
```

Имея такое определение, можно использовать таблицы CSS-стилей, с помощью которых задать цвет, отступы, рамки и другие атрибуты вывода предупреждений этого класса. Но что делать, если необходимо написать JavaScript-сценарий, который мог бы отыскивать теги , являющиеся членами класса «предупреждений», и манипулировать этими тегами? Возможное решение приводится в примере 15.4. Здесь определяется метод getElements(), который позволяет отобрать элементы по имени класса и/или тега. Обратите внимание на ухищрения, используемые при работе со свойством className, – они вызваны тем, что в данном свойстве могут храниться имена нескольких классов. Метод getElements() содержит вложенную функцию isMember(), которая проверяет принадлежность HTML-элемента к заданному классу.

Пример 15.4. Отбор HTML-элементов по имени класса или тега

```
/**
 * getElements(classname, tagname, root):
 * Возвращает массив DOM-элементов, которые являются членами заданного класса,
 * соответствуют тегам с определенным именем и вложены в элемент root.
 *
 * Если аргумент classname не определен, отбор элементов производится
 * без учета принадлежности к какому-то определенному классу.
 * Если аргумент tagname не определен, отбор элементов производится без учета имени тега.
 * Если аргумент root не определен, поиск производится в объекте document.
 * Если аргумент root является строкой, он воспринимается как идентификатор
 * элемента и поиск производится методом getElementsById()
 */
function getElements(classname, tagname, root) {
```

```
    // Если корневой элемент не определен, произвести поиск по всему документу
    // Если это строка, найти сам объект
    if (!root) root = document;
    else if (typeof root == "string") root = document.getElementById(root);

    // Если имя тега не определено, искать без учета имени тега
    if (!tagname) tagname = "*";

    // Искать элементы, вложенные в элемент root и имеющие определенное имя тега
    var all = root.getElementsByTagName(tagname);

    // Если имя класса не определено, вернуть все теги без учета имени класса
    if (!classname) return all;

    // В противном случае отобрать элементы по имени класса
    var elements = [];                    // Создается пустой массив
    for(var i = 0; i < all.length; i++) {
        var element = all[i];
        if (isMember(element, classname)) // Метод isMember() определен далее
        elements.push(element);           // Добавлять члены класса в массив
    }

    // Обратите внимание: всегда возвращается массив, даже если он пустой
    return elements;

    // Определяет принадлежность элемента к заданному классу.
    // Эта функция оптимизирована для случая, когда свойство
    // className содержит единственное имя класса. Но учитывает возможность
    // наличия нескольких имен классов, разделенных пробелами.
    function isMember(element, classname) {
        var classes = element.className;       // Получить список класов
        if (!classes) return false;            // Класс не определен
        if (classes == classname) return true; // Точное совпадение

        // Нет точного совпадения, поэтому если в списке нет пробелов,
        // то этот элемент не является членом класса.
        var whitespace = /\s+/;
        if (!whitespace.test(classes)) return false;

        // В этой точке известно, что элемент принадлежит нескольким
        // классам, поэтому нужно проверить каждый из них.
        var c = classes.split(whitespace);     // Разбить по символам пробелов
        for(var i = 0; i < c.length; i++) {    // Цикл по всем классам
            if (c[i] == classname) return true; // Проверить совпадение
        }

        return false; // Не обнаружено ни одного совпадения
    }
}
```

15.7. Модификация документа

Обход узлов документа может быть полезной функцией, но реальную мощь базовой модели DOM API обеспечивают средства, позволяющие использовать JavaScript для динамического изменения документов. Следующие примеры демонстрируют основные приемы модификации документов и некоторые другие возможности.

Пример 15.5 включает JavaScript-функцию `sortkids()`, примерный документ и HTML-кнопку, которая при щелчке на ней вызывает функцию `sortkids()` и передает ей идентификатор тега `<ul>`. Функция `sotrkids()` отыскивает элементы, дочерние по отношению к заданному, выполняет сортировку, основываясь на текстовом содержимом, и методом `appendChild()` переупорядочивает элементы в документе так, чтобы они шли друг за другом в алфавитном порядке.

Пример 15.5. Сортировка элементов в алфавитном порядке

```
<script>
function sortkids(e) {
    // Это элемент, потомки которого должны быть отсортированы
    if (typeof e == "string") e = document.getElementById(e);

    // Скопировать дочерние элементы (не текстовые узлы) в массив
    var kids = [];
    for(var x = e.firstChild; x != null; x = x.nextSibling)
        if (x.nodeType == 1 /* Node.ELEMENT_NODE */) kids.push(x);

    // Выполнить сортировку массива на основе текстового содержимого
    // каждого дочернего элемента. Здесь предполагается, что каждый
    // потомок имеет единственный вложенный элемент - узел Text
    kids.sort(function(n, m) {     // Функция сравнения для сортировки
        var s = n.firstChild.data; // Текстовое содержимое узла n
        var t = m.firstChild.data; // Текстовое содержимое узла m
        if (s < t) return -1;      // Узел n должен быть выше узла m
        else if (s > t) return 1;  // Узел n должен быть ниже узла m
        else return 0;             // Узлы n и m эквивалентны
    });

    // Теперь нужно перенести узлы-потомки обратно в родительский элемент
    // в отсортированном порядке. Когда в документ вставляется уже
    // существующий элемент, он автоматически удаляется из текущей позиции,
    // в результате операция добавления этих копий элементов автоматически
    // перемещает их из старой позиции. Примечательно, что все текстовые узлы,
    // которые были пропущены, останутся на месте.
    for(var i = 0; i < kids.length; i++) e.appendChild(kids[i]);
}
</script>
<ul id="list"> <!-Этот список будет отсортирован -->
<li>один<li>два<li>три<li>четыре<li>пять
<!-- элементы не в алфавитном порядке -->
</ul>
<!-- кнопка, щелчок на которой запускает сортировку списка -->
<button onclick="sortkids('list')">Сортировать</button>
```

Результаты выполнения примера 15.5 на рис. 15.3 демонстрируют, что после щелчка на кнопке элементы списка сортируются в алфавитном порядке.

Обратите внимание: в примере 15.5 узлы сначала копируются в отдельный массив. Это не только позволяет упростить сортировку, но имеет и другие преимущества. Объекты `NodeList`, которые являются значениями свойства `childNodes` и возвращаются методом `getElementByTagName()`, являются «живыми», т. е. любые изменения в документе немедленно отражаются на объекте `NodeList`. Это может стать источником определенных сложностей, когда добавление или удаление узлов списка производится в процессе обхода этого списка. По этой причине гораз-

***Рис. 15.3.** Список до и после сортировки*

до безопаснее сделать «мгновенный снимок» узлов, скопировав их в настоящий массив, прежде чем выполнять их обход.

Пример 15.5 изменяет структуру документа, переупорядочивая элементы. Пример 15.6 изменяет содержимое документа, изменяя его текст. В этом примере определяется функция `upcase()`, которая рекурсивно обходит все дочерние узлы заданного узла и преобразует символы во всех текстовых узлах, приводя их к верхнему регистру.

Пример 15.6. Преобразование содержимого документа в верхний регистр

```
// Эта функция рекурсивно обходит узел n и всех его потомков, заменяя
// все узлы Text их эквивалентами в верхнем регистре.
function upcase(n) {
    if (n.nodeType == 3 /*Node.TEXT_NODE*/) {
        // Если это узел Text, преобразовать его в верхний регистр.
        n.data = n.data.toUpperCase( );
    }
    else {
        // Если это не узел Text, обойти его потомков
        // и рекурсивно вызвать эту функцию для каждого потомка.
        var kids = n.childNodes;
        for(var i = 0; i < kids.length; i++) upcase(kids[i]);
    }
}
```

В примере 15.6 просто изменяется содержимое свойства `data` каждого встреченного текстового узла. Кроме того, существует возможность добавлять, удалять и заменять текст внутри узла `Text` с помощью методов `appendData()`, `insertData()`, `deleteData()` и `replaceData()`. Эти методы не определены непосредственно в интерфейсе `Text`, а наследуются им от интерфейса `CharacterData`. Дополнительные сведения об этих методах можно найти в описании `CharacterData` в части IV книги.

В примере 15.5 выполнялось переупорядочивание элементов документа, но при этом их родительский элемент оставался тем же самым. Однако следует заметить, что DOM API позволяет свободно перемещать узлы по дереву документа (но только в пределах одного документа). Пример 15.7 демонстрирует это путем

определения функции с именем `embolden()`, заменяющей указанный узел новым элементом (созданным с помощью метода `createElement()` объекта `Document`), представляющим HTML-тег `<b>`, и делающей исходный узел дочерним для нового узла `<b>`. В HTML-документе благодаря этому любой текст в данном узле или в его потомках отображается жирным шрифтом.

Пример 15.7. Замена родителя узла элементом <b>

```
<script>
// Эта функция принимает в качестве аргумента узел n, заменяет его
// в дереве узлом Element, представляющим HTML-тег <b>, а затем делает
// исходный узел дочерним для нового элемента <b>.
function embolden(n) {
    if (typeof n == "string") n = document.getElementById(n);    // Ищем узел
    var b = document.createElement("b"); // Создаем новый элемент <b>
    var parent = n.parentNode;           // Получаем родительский узел
    parent.replaceChild(b, n);           // Заменяем узел тегом <b>
    b.appendChild(n);                    // Делаем узел дочерним тегу <b>
}
</script>

<!-- Пара простых абзацев -->
<p id="p1"><i>Это </i> абзац #1.</p>
<p id="p2"><i>Это </i> абзац #2.</p>
<!-- Кнопка, вызывающая функцию embolden() для первого абзаца (p1) -->
<button onclick="embolden('p1');">Выделить жирным шрифтом</button>
```

15.7.1. Модификация атрибутов

Изменять документы можно не только путем вставки, удаления, изменения родителя или другого переупорядочивания узлов, но и простой установкой значений атрибутов элементов документа. Один из возможных способов – использование метода `element.setAttribute()`. Например:

```
var headline = document.getElementById("headline"); // Найти элемент по имени
headline.setAttribute("align", "center");           // Установить align='center'
```

В DOM-элементах, представляющих HTML-атрибуты, определяются JavaScript-свойства, соответствующие каждому из стандартных атрибутов (даже устаревших, таких как `align`), поэтому получить такой же эффект можно, например, так:

```
var headline = document.getElementById("headline");
headline.align = "center"; // Установить значение атрибута выравнивания.
```

Как показано в главе 16, аналогичным образом огромного разнообразия эффектов можно добиться изменением свойств CSS-стилей HTML-элементов. При этом структура документа и его содержимое остаются неизменными, изменяется лишь его *представление*.

15.7.2. Работа с фрагментами документа

Объект `DocumentFragment` – это специальный тип узла, который отсутствует в самом документе и используется лишь как временный контейнер для хранения последовательности узлов, что позволяет манипулировать этими узлами как

единым объектом. Когда выполняется вставка объекта DocumentFragment в документ (методом appendChild(), insertBefore() или replaceChild() объекта Node), вставляется не сам объект DocumentFragment, а каждый из его потомков.

Создается объект DocumentFragment методом document.createDocumentFragment(). Добавлять узлы в объект DocumentFragment можно методом appendChild() или любым другим, относящимся к объекту Node. Затем, когда все эти узлы будут готовы к вставке в документ, добавляется сам объект DocumentFragment. После операции вставки в документ фрагмент пустеет, и его содержимое нельзя использовать повторно, если предварительно не добавить в него новые узлы-потомки. Этот процесс демонстрируется в примере 15.8. Здесь определяется функция reverse(), которая использует объект DocumentFragment в качестве временного хранилища при изменении порядка следования дочерних узлов на обратный.

Пример 15.8. Использование объекта DocmentFragment

```
// Изменяет порядок следование дочерних узлов на обратный
function reverse(n) {
    // Создать пустой объект DocumentFragment, который будет
    // использоваться как временное хранилище
    var f = document.createDocumentFragment( );

    // Обойти все дочерние узлы в обратном порядке и переместить
    // их во временное хранилище.
    // Последний дочерний узел элемента n станет первым дочерним
    // узлом элемента f, и наоборот.
    // Обратите внимание: добавление узла в f автоматически вызывает
    // его удаление из n.
    while(n.lastChild) f.appendChild(n.lastChild);

    // В заключение переместить дочерние узлы из f обратно в n за один шаг.
    n.appendChild(f);
}
```

15.8. Добавление содержимого в документ

Методы Document.createElement() и Document.createTextNode() создают новые узлы типа Element и Text, а методы Node.appendChild(), Node.insertBefore() и Node.replaceChild() могут использоваться для добавления этих узлов в документ. С помощью указанных методов можно построить DOM-дерево с произвольным содержимым.

В расширенном примере 15.9 определяется функция log() для записи в журнал сообщения и объекта. Кроме того, определяется вспомогательная функция log.debug(), которая представляет собой удобную альтернативу вызовам alert(), используемым при отладке сценариев. В качестве «сообщения» функции log() может передаваться как обычная текстовая строка, так и JavaScript-объект. В первом случае строка просто отображается «как есть», во втором случае, когда в журнал записывается объект, он выводится в виде таблицы с именами свойств объекта и их значениями. В любом из этих двух случаев новое содержимое создается с помощью функций createElement() и createTextNode().

При использовании соответствующих таблиц CSS-стилей (которые включены в пример) вывод функции log() соответствует показанному на рис. 15.4.

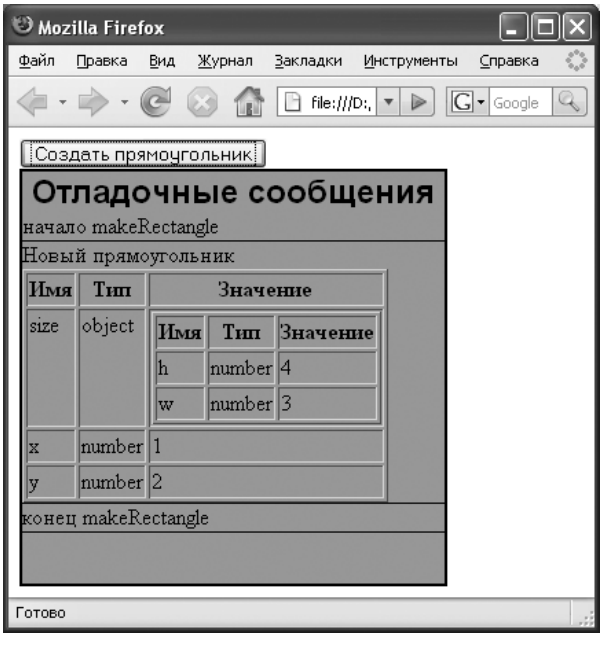

***Рис. 15.4.** Результат исполнения функции log( )*

Хотя пример 15.9 очень большой, он хорошо прокомментирован и заслуживает внимательного изучения. Обратите особое внимание на вызовы методов `createElement()`**,** `createTextNode()` **и** `appendChild()`**. Порядок использования этих методов для создания относительно сложной HTML-таблицы демонстрируется в частной функции** `log.makeTable()`**.**

Пример 15.9. Средства протоколирования в клиентском JavaScript-коде

```
/*
 * Log.js: Средства ненавязчивого протоколирования
 *
 * Данный модуль определяет единственный глобальный символ – функцию log().
 * Протоколирование сообщений производится вызовом этой функции
 * с двумя или тремя аргументами:
 *
 * category: тип сообщения. Это необходимо, чтобы можно было разрешать или
 *   запрещать вывод сообщений различных типов, а также для того, чтобы иметь
 *   возможность оформлять их разными стилями независимо друг от друга.
 *   Подробности далее.
 *
 * message: текст сообщения. Может быть пустой строкой, если функции передается объект
 *
 * object: записываемый в журнал объект. Это необязательный аргумент.
 *   Если он определен, свойства объекта выводятся в форме таблицы.
 *   Любое свойство, значением которого является объект, протоколируется рекурсивно.
 *
 * Вспомогательные функции:
```

```
 *
 * Функции log.debug() и log.warn() – это сервисные функции, которые просто
 *   вызывают функцию log() с жестко предопределенными типами "debug"
 *   и "warning". Довольно просто можно определить функцию, которая подменит
 *   метод alert() и будет вызывать функцию log().
*
 * Включение режима протоколирования
 *
 *   Протоколируемые сообщения по умолчанию *не* отображаются. Разрешить
 *   отображение сообщений того или иного типа можно одним из двух способов.
 *   Первый – создать элемент <div> или другой контейнерный элемент
 *   со значением атрибута id равным "<category>_log". Для вывода сообщений
 *   с категорией "debug" можно вставить в документ следующую строку:
 *
 *     <div id="debug_log"></div>
 *
 *   В этом случае все сообщения данного типа будут добавляться в элемент-
 *   контейнер, для которого могут быть определены свои стили отображения.
 *
 *   Второй способ активировать вывод сообщений определенной категории –
 *   установить значение соответствующего свойства. Например, чтобы разрешить
 *   вывод сообщений категории "debug", следует установить свойство
 *   log.options.debugEnabled = true. После этого создается элемент
 *   <div class="log">, куда и будут добавляться сообщения.
 *   Чтобы запретить отображение протоколируемых сообщений даже
 *   при наличии контейнерного элемента с соответствующим значением
 *   атрибута id следует установить значение свойства:
 *   log.options.debugDisabled=true. Чтобы опять разрешить вывод
 *   сообщений в свойство, соответствующее заданной категории,
 *   следует записать значение false.
 *
 * Оформление сообщений
 *
 *   В дополнение к возможности оформления самого контейнера для сообщений
 *   можно использовать CSS-стили, чтобы оформлять вывод отдельных
 *   сообщений. Каждое сообщение помещается в тег <div> с CSS-классом
 *   <category>_message. Например, сообщения из категории "debug" будут
 *   иметь класс "debug_message"
 *
 * Свойства объекта Log
 *
 *   Порядок протоколирования можно изменять установкой свойств
 *   объекта log.options, как, например, те, которые были описаны ранее
 *   и использовались для разрешения/запрета вывода сообщений отдельных
 *   категорий. Далее приводится перечень доступных свойств:
 *
 * log.options.timestamp: Если в это свойство записано значение true,
 *   к каждому сообщению будут добавлены дата и время.
 *
 * log.options.maxRecursion: Целое число, определяющее глубину вложения
 *   таблиц при отображении сведений об объектах. Если внутри таблиц не должно
 *   быть вложенных таблиц, в это свойство следует записать значение 0
 *
 * log.options.filter: Функция, используемая для определения, какие свойства
```

```
 *   объектов должны отображаться. Функция-фильтр должна принимать имя
 *   и значение свойства и возвращать true, если свойство должно отображаться
 *   в таблице с объектом, и false – в противном случае
 */
function log(category, message, object) {
    // Если заданная категория явно запрещена, ничего не делать
    if (log.options[category + "Disabled"]) return;

    // Отыскать элемент-контейнер
    var id = category + "_log";
    var c = document.getElementById(id);

    // Если контейнер не найден и вывод сообщений этой категории разрешен,
    // создать новый контейнерный элемент.
    if (!c && log.options[category + "Enabled"]) {
        c = document.createElement("div");
        c.id = id;
        c.className = "log";
        document.body.appendChild(c);
    }

    // Если контейнер по-прежнему отсутствует – игнорировать сообщение
    if (!c) return;

    // Если вывод информации о дате/времени разрешен, добавить ее
    if (log.options.timestamp)
        message = new Date() + ": " + (message?message:"");

    // Создать элемент <div>, в который будет записано сообщение
    var entry = document.createElement("div");
    entry.className = category + "_message";

    if (message) {
        // Добавить сообщение в элемент
        entry.appendChild(document.createTextNode(message));
    }

    if (object && typeof object == "object") {
        entry.appendChild(log.makeTable(object, 0));
    }

    // В заключение добавить запись в контейнер
    c.appendChild(entry);
}

// Создает таблицу для отображения свойств заданного объекта
log.makeTable = function(object, level) {
    // Если достигнуто значение, ограничивающее рекурсию, вернуть узел Text.
    if (level > log.options.maxRecursion)
        return document.createTextNode(object.toString());

    // Создать таблицу, которая будет возвращена
    var table = document.createElement("table");
    table.border = 1;

    // Добавить в таблицу заголовки столбцов Имя|Тип|Значение
    var header = document.createElement("tr");
    var headerName = document.createElement("th");
    var headerType = document.createElement("th");
```

```
var headerValue = document.createElement("th");
headerName.appendChild(document.createTextNode("Имя"));
headerType.appendChild(document.createTextNode("Тип"));
headerValue.appendChild(document.createTextNode("Значение"));
header.appendChild(headerName);
header.appendChild(headerType);
header.appendChild(headerValue);
table.appendChild(header);

// Получить имена свойств объекта и отсортировать их в алфавитном порядке
var names = [];
for(var name in object) names.push(name);
names.sort();

// Теперь обойти эти свойства
for(var i = 0; i < names.length; i++) {
    var name, value, type;
    name = names[i];
    try {
        value = object[name];
        type = typeof value;
    }
    catch(e) { // Этого не должно происходить, но случается в Firefox
        value = "<неизвестное значение>";
        type = "не известен";
    };

    // Пропустить свойство, если оно отвергается функцией-фильтром
    if (log.options.filter && !log.options.filter(name, value)) continue;

    // Никогда не отображать исходные тексты функций – это может
    // потребовать слишком много места
    if (type == "function") value = "{/*исходные тексты не выводятся*/}";

    // Создать строку таблицы для отображения имени свойства, типа и значения
    var row = document.createElement("tr");
    row.vAlign = "top";
    var rowName = document.createElement("td");
    var rowType = document.createElement("td");
    var rowValue = document.createElement("td");
    rowName.appendChild(document.createTextNode(name));
    rowType.appendChild(document.createTextNode(type));

    // В случае объекта произвести рекурсивный вызов, чтобы вывести вложенные объекты
    if (type == "object")
        rowValue.appendChild(log.makeTable(value, level+1));
    else
        rowValue.appendChild(document.createTextNode(value));

    // Добавить ячейки в строку, затем строку добавить к таблице
    row.appendChild(rowName);
    row.appendChild(rowType);
    row.appendChild(rowValue);

    table.appendChild(row);
}

// Вернуть таблицу.
return table;
```

```
}
// Создать пустой объект options
log.options = {};

// Вспомогательные функции для вывода сообщений предопределенных типов
log.debug = function(message, object) { log("debug", message, object); };
log.warn = function(message, object) { log("warning", message, object); };

// Раскомментируйте следующую строку, чтобы подменить функцию alert()
// одноименной функцией, использующей функцию log()
// function alert(msg) { log("alert", msg); }
```

Отладочные сообщения, которые приводятся на рис. 15.4, были сгенерированы следующим фрагментом программного кода:

```
<head>
<script src="Log.js"></script>                        <!--подключить log() -->
<link rel="stylesheet" type="text/css" href="log.css"><!--подключить стили -->
</head>
<body>
<script>
function makeRectangle(x, y, w, h) {     // Это отлаживаемая функция
    log.debug("начало makeRectangle");   // Вывод сообщения
    var r = {x:x, y:y, size: { w:w, h:h }};
    log.debug("Новый прямоугольник", r); // Вывод объекта
    log.debug("конец makeRectangle");    // Вывод другого сообщения
    return r;
}
</script>
<!-- этой кнопкой вызывается отлаживаемая функция -->
<button onclick="makeRectangle(1,2,3,4);">Создать прямоугольник</button>
<!-- Это место для вывода сообщений -->
<!-- Вывод сообщений разрешается созданием элемента <div> в документе -->
<div id="debug_log" class="log"></div>
</body>
```

Показанные на рис. 15.4 отладочные сообщения были оформлены средствами CSS-стилей, импортированных в документ с помощью тега `<link>`. При создании рисунка использовались следующие стили:

```
#debug_log { /* Стили оформления контейнера с отладочными сообщениями */
    background-color: #aaa;  /* серый фон */
    border: solid black 2px; /* черная рамка */
    overflow: auto;          /* полосы прокрутки */
    width: 75%;              /* ограничить ширину элемента */
    height: 300px;           /* ограничить размер по вертикали */
}

#debug_log:before { /* Заголовок области вывода сообщений */
    content: "Отладочные сообщения ";
    display: block;
    text-align: center;
    font: bold 18pt sans-serif ;
}

.debug_message { /* Отделять сообщения тонкой горизонтальной линией */
    border-bottom: solid black 1px;
}
```

Больше о CSS вы узнаете в главе 16. Сейчас не очень важно разбираться в этой теме во всех подробностях. В рассматриваемом примере вы можете видеть, как подключенные CSS-стили сказываются на содержимом документа, динамически генерируемом функцией `log()`.

15.8.1. Удобные методы создания узлов

При изучении примера 15.9 вы могли заметить, что для создания содержимого документов приходится вызывать большое число методов: прежде всего, необходимо создать объект `Element`, затем установить его атрибуты, а потом создать узел `Text` и добавить его в объект `Element`. После этого `Element` добавляется в родительский объект `Element` и т. д. Чтобы просто создать элемент `<table>`, установить один атрибут и добавить строку заголовка, в примере 15.9 потребовалось написать 13 строк программного кода. В примере 15.10 приводится определение предназначенной для создания объекта `Element` вспомогательной функции, которая существенно упрощает выполнение повторяющихся операций при DOM-программировании.

Пример 15.10 определяет единственную функцию с именем `make()`. Эта функция создает объект `Element` с заданным именем тега, устанавливает его атрибуты и добавляет к нему дочерний узел. Атрибуты определяются как свойства объекта, а дочерний узел передается как массив. Элементами массива могут быть строки, которые преобразуются в текстовые узлы, или другие объекты типа `Element`, обычно создаваемые с помощью вложенных вызовов функции `make()`.

Функция `make()` обладает очень гибким синтаксисом и допускает два сокращенных варианта вызова. Первый – когда не задается ни одного атрибута; в этом случае аргумент с атрибутами может быть опущен и вместо него передается аргумент с дочерним узлом. Второй – когда существует единственный дочерний узел, который можно передать функции непосредственно, не помещая в массив. Единственное ограничение – эти два способа сокращенной записи вызова не могут использоваться вместе, если единственный дочерний узел не является текстовым узлом, передаваемым в виде строки.

Благодаря функции `make()` целых 13 строк программного кода, которые в примере 15.9 создают элемент `<table>`, могут быть сокращены следующим образом:

```
var table = make("table", {border:1}, make("tr", [make("th", "Имя"),
                                                  make("th", "Тип"),
                                                  make("th", "Значение")]));
```

Но и этот фрагмент можно записать еще короче. В примере 15.10 вслед за функцией `make()` определяется еще одна вспомогательная функция с именем `maker()`. Она принимает имя тега и возвращает вложенную функцию, которая вызывает функцию `make()` с именем заданного тега. Если потребуется создать большое число таблиц, можно будет определить функции для создания таблиц следующим образом:

```
var table = maker("table"), tr = maker("tr"), th = maker("th");
```

После этого программный код создания таблицы с заголовком уместится в единственной строке:

```
var mytable = table({border:1}, tr([th("Имя"), th("Тип"), th("Значение")]));
```

Пример 15.10. Вспомогательные функции создания элементов

```
/**
 * make(tagname, attributes, children):
 *   создает HTML-элемент с заданным именем тега tagname, атрибутами
 *   и дочерними элементами.
 *
 * Аргумент attributes – это JavaScript-объект: имена и значения его свойств - это имена
 * и значения атрибутов. Если атрибуты отсутствуют, а аргумент children
 * представляет собой массив или строку, тогда аргумент attributes
 * можно просто опустить, а значение аргумента children передавать во втором аргументе.
 *
 * Как правило, аргумент children представляет собой массив дочерних
 * элементов, добавляемых к создаваемому элементу. Если элемент не имеет
 * потомков, аргумент children может быть опущен.
 * Если дочерний элемент единственный, его можно передавать непосредственно,
 * не заключая его в массив. (Но если дочерний элемент не является строкой
 * и атрибуты отсутствуют, тогда массив должен использоваться обязательно.)
 *
 * Пример: make("p", ["Это ", make("b", "жирный"), " шрифт."]);
 *
 * Идея взята из библиотеки MochiKit (http://mochikit.com),
 * автор библиотеки – Боб Ипполито (Bob Ippolito)
 */
function make(tagname, attributes, children) {

    // Если было передано два аргумента и аргумент attributes представляет
    // собой массив или строку, значит, на самом деле это аргумент children.
    if (arguments.length == 2 &&
        (attributes instanceof Array || typeof attributes == "string")) {
        children = attributes;
        attributes = null;
    }

    // Создать элемент
    var e = document.createElement(tagname);

    // Установить атрибуты
    if (attributes) {
        for(var name in attributes) e.setAttribute(name, attributes[name]);
    }

    // Добавить дочерний узел, если таковой был определен.
    if (children != null) {
        if (children instanceof Array) {           // Если это массив
            for(var i = 0; i < children.length; i++) { // обойти все элементы
                var child = children[i];
                if (typeof child == "string")      // Текстовый узел
                    child = document.createTextNode(child);
                e.appendChild(child);              // Все остальное – это Node
            }
        }
        else if (typeof children == "string") // Единственный узел-потомок Text
            e.appendChild(document.createTextNode(children));
        else e.appendChild(children);         // Единственный узел-потомок другого типа
    }
```

```
    // Вернуть элемент
    return e;
}

/**
 * maker(tagname): Возвращает функцию, которая вызывает make() для заданного тега.
 * Пример: var table = maker("table"), tr = maker("tr"), td = maker("td");
 */
function maker(tag) {
    return function(attrs, kids) {
        if (arguments.length == 1) return make(tag, attrs);
        else return make(tag, attrs, kids);
    }
}
```

15.8.2. Свойство innerHTML

Хотя консорциум W3C никогда официально не определял свойство `innerHTML` как составную часть модели DOM, тем не менее это свойство узлов `HTMLElement` является настолько важным, что поддерживается всеми современными броузерами. При чтении из этого свойства вы в формате HTML получаете текст, который представляет дочерние узлы элемента. При записи в это свойство броузер запускает синтаксический анализатор HTML-кода для разбора строки и замещает дочерние элементы теми, которые были получены от анализатора.

Описывать HTML-документ в виде строки с текстом в формате HTML обычно удобнее и проще, чем использовать для этих же целей последовательность вызовов `createElement()` и `appendChild()`. Снова вернемся к той части примера 15.9, где создается новый элемент `<table>` и затем к нему добавляется строка заголовка. Благодаря свойству `innerHTML` этот относительно крупный фрагмент программного кода можно переписать следующим образом:

```
var table = document.createElement("table"); // Создать элемент <table>
table.border = 1;                            // Установить атрибут
// Добавить в таблицу заголовок Имя|Тип|Значение
table.innerHTML = "<tr><th>Имя</th><th>Тип</th><th>Значение</th></tr>";
```

Веб-броузеры по определению прекрасно справляются с анализом HTML-кода. Оказывается, что использование свойства `innerHTML` гораздо эффективнее, особенно при анализе больших фрагментов HTML-текста. Однако следует отметить, что операция добавления небольших фрагментов текста в свойство `innerHTML` с помощью оператора `+=` обычно не отличается эффективностью, поскольку требует как сериализации, так и синтаксического анализа.

Свойство `innerHML` было введено компанией Microsoft в IE 4. Оно входит в квартет наиболее важных и часто используемых свойств. Остальные три свойства, `outerHTML`, `innerText` и `outerText`, описанные в конце этой главы, не поддерживаются в Firefox и родственных броузерах.

15.9. Пример: динамическое создание оглавления

В предыдущих разделах было продемонстрировано, как использовать модуль `Core` модели DOM API для обхода документа, выборки элементов из документа, изме-

нения и добавления содержимого документа. Все эти операции собраны воедино в примере 15.11, который автоматически создает оглавление HTML-документа.

В программном коде примера определяется единственный метод `maketoc()` и регистрируется обработчик события `onload`, благодаря чему функция вызывается автоматически после загрузки документа. Метод `maketoc()` обходит документ в поисках тегов `<h1>`, `<h2>`, `<h3>`, `<h4>`, `<h5>` и `<h6>`, которые, как предполагается, отмечают начало разделов документа. Кроме того, метод `maketoc()` отыскивает элемент со значением атрибута `id="toc"` и строит оглавление внутри этого элемента. В ходе этого процесса метод `maketoc()` добавляет номера разделов в заголовки этих разделов, вставляет именованные якорные элементы и затем вставляет в начало каждого раздела ссылку обратно на оглавление. Оглавление, генерируемое функцией `maketoc()`, показано на рис. 15.5.

Функция `maketoc()` может представлять интерес для тех, кто сопровождает и исправляет длинные документы, разбитые на разделы с помощью тегов `<h1>`, `<h2>` и им подобных. Оглавления очень полезны в длинных документах, но если документ редактируется часто, то бывает сложно обеспечивать синхронизацию оглавления с самим документом. Программный код примера 15.11 написан в ненавязчивом стиле: чтобы воспользоваться им, достаточно просто включить модуль в HTML-документ и создать контейнерный элемент для метода `maketoc()`, который и создаст оглавление документа. При желании можно с помощью CSS-таблицы определить стиль оглавления. Вот пример:

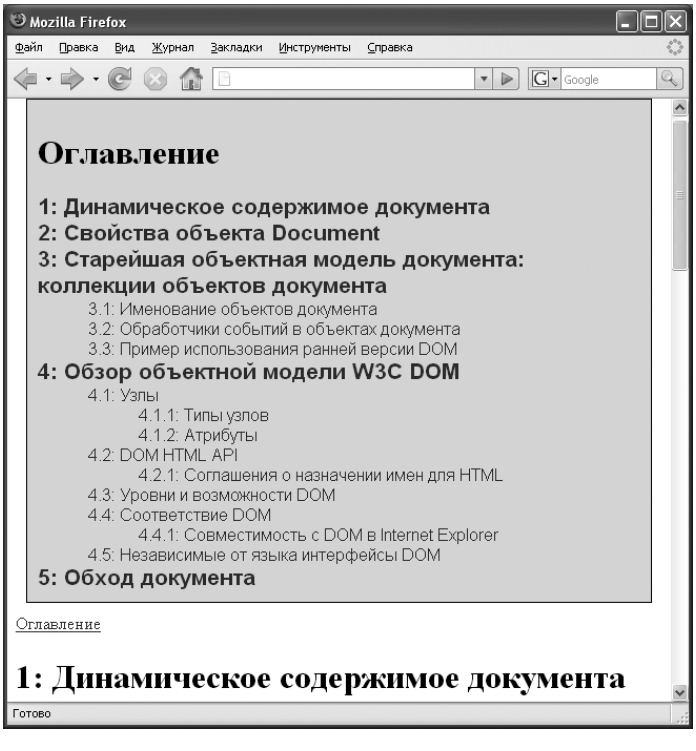

Рис. 15.5. *Динамически созданное оглавление*

```
<script src="TOC.js"></script> <!-- Загрузка функции maketoc() -->
<style>
#toc {/* Следующие стили применяются к контейнерному элементу с оглавлением */
    background: #ddd;            /* фон светло-серого цвета */
    border: solid black 1px;     /* простейшая рамка */
    margin: 10px; padding: 10px; /* отступы */
}
.TOCEntry { font-family: sans-serif; }          /* Пункты выводить шрифтом sans-serif */
.TOCEntry a { text-decoration: none; }           /* Ссылки не подчеркивать */
.TOCLevel1 { font-size: 16pt; font-weight: bold; } /* Пункты первого уровня */
                                                   /* крупным жирным шрифтом */
.TOCLevel2 { font-size: 12pt; margin-left: .5in; } /* Пункты второго уровня с отступом */
.TOCLevel3 { font-size: 12pt; margin-left: 1in; }  /* Пункты третьего уровня */
                                                   /* с отступом */
.TOCBackLink { display: block; }    /* Обратные ссылки в той же строке */
.TOCSectNum:after { content: ":"; } /* Добавлять двоеточие после номера раздела */
</style>
<body>
<div id="toc"><h1>Оглавление</h1></div> <!-- здесь находится оглавление -->
<!--
... здесь находится остальная часть документа ...
-->
```

Далее следует программный код модуля *TOC.js*. Пример 15.11 достаточно длинный, но хорошо прокомментирован и основан на уже знакомых вам приемах. Он заслуживает изучения как практический пример возможностей W3C DOM.

Пример 15.11. Автоматическая генерация оглавления

```
/**
 * TOC.js: создает оглавление документа.
 *
 * В этом модуле определяется единственная функция maketoc(), также он
 * регистрирует обработчик события onload, благодаря чему функция
 * запускается автоматически сразу же после загрузки документа
 * После запуска функция maketoc() сначала просматривает документ в поисках
 * элемента со значением атрибута id="toc". Если такой элемент в документе
 * отсутствует, maketoc() ничего не делает. Если такой элемент обнаруживается,
 * maketoc() обходит документ, отыскивает все теги от <h1> до <h6>
 * и создает оглавление, которое затем добавляется в элемент "toc".
 * Функция maketoc() добавляет номера разделов к каждому заголовку каждого
 * раздела и вставляет перед каждым заголовком обратные ссылки на оглавление
 * Ссылки и якорные элементы с именами, начинающимися с префикса "TOC",
 * создаются функцией maketoc(), т.е. следует избегать
 * использования данного префикса в своих HTML-документах.
 *
 * Формат вывода пунктов оглавления может настраиваться с помощью CSS.
 * Все записи принадлежат к классу "TOCEntry". Кроме того, записи также
 * принадлежат классу, имя которого соответствует уровню заголовка раздела.
 * Для тегов <h1> генерируются пункты с классом "TOCLevel1",
 * Для тегов <h2> - пункты с классом "TOCLevel2", и т.д.
 * Номера разделов вставляются в заголовки, принадлежащие классу "TOCSectNum",
 * а обратные ссылки на оглавление генерируются для заголовков,
 * принадлежащих классу "TOCBackLink".
```

```
 *
 * По умолчанию сгенерированные обратные ссылки содержат текст "Contents".
 * Чтобы изменить этот текст (например, с целью интернационализации),
 * следует записать его в свойство maketoc.backlinkText.
 **/
function maketoc() {
    // Найти контейнер. В случае отсутствия такового просто завершить работу.
    var container = document.getElementById('toc');
    if (!container) return;

    // Обойти документ, добавить в массив все теги <h1>...<h6>
    var sections = [];
    findSections(document, sections);

    // Вставить якорный элемент перед контйнером, чтобы можно было
    // создавать обратные ссылки на него
    var anchor = document.createElement("a"); // Создать узел <a>
    anchor.name = "TOCtop";                   // Установить атрибуты
    anchor.id = "TOCtop";                     // name и id (для IE)
    container.parentNode.insertBefore(anchor, container); // Вставить перед оглавлением
    // Инициализировать массив для отслеживания номеров разделов
    var sectionNumbers = [0,0,0,0,0,0];

    // Обойти в цикле все найденные заголовки разделов
    for(var s = 0; s < sections.length; s++) {
        var section = sections[s];

        // Определить уровень заголовка
        var level = parseInt(section.tagName.charAt(1));
        if (isNaN(level) || level < 1 || level > 6) continue;

        // Увеличить номер раздела для данного уровня
        // и сбросить в ноль номера всех нижележащих подуровней
        sectionNumbers[level-1]++;
        for(var i = level; i < 6; i++) sectionNumbers[i] = 0;

        // Собрать номер раздела для данного уровня,
        // чтобы создать такой номер, как, например, 2.3.1
        var sectionNumber = "";
        for(i = 0; i < level; i++) {
            sectionNumber += sectionNumbers[i];
            if (i < level-1) sectionNumber += ".";
        }

        // Добавить номер и пробел в заголовок.
        // Номер помещается в тег <span>, чтобы можно было влиять на формат вывода.
        var frag = document.createDocumentFragment(); // Для хранения номера и пробела
        var span = document.createElement("span");    // Узел span номера сделать
        span.className = "TOCSectNum";                // доступным для форматирования
        span.appendChild(document.createTextNode(sectionNumber)); // Добавить номер
        frag.appendChild(span);                     // Добавить тег с номером во фрагмент
        frag.appendChild(document.createTextNode(" ")); // Добавить пробел
        section.insertBefore(frag, section.firstChild); // Добавить все в заголовок
        // Создать якорный элемент, который будет отмечать начало раздела.
        var anchor = document.createElement("a");
        anchor.name = "TOC"+sectionNumber;      // Имя элемента, на которое будет ссылка
```

```
            anchor.id = "TOC"+sectionNumber; // В IE для генерации ссылок
                                             // необходимо определить атрибут id
            // Обернуть якорным элементом обратную ссылку на оглавление
            var link = document.createElement("a");
            link.href = "#TOCtop";
            link.className = "TOCBackLink";
            link.appendChild(document.createTextNode(maketoc.backlinkText));
            anchor.appendChild(link);

            // Вставить якорный элемент и ссылку непосредственно перед заголовком раздела
            section.parentNode.insertBefore(anchor, section);

            // Создать ссылку на этот раздел.
            var link = document.createElement("a");
            link.href = "#TOC" + sectionNumber; // Определить адрес ссылки
            link.innerHTML = section.innerHTML; // записать текст заголовка в текст ссылки

            // Поместить ссылку в элемент div, чтобы получить возможность влиять
            // на формат отображения на основе уровня заголовка
            var entry = document.createElement("div");
            entry.className = "TOCEntry TOCLevel" + level; // Для CSS
            entry.appendChild(link);

            // И добавить элемент div в контейнер с оглавлением
            container.appendChild(entry);
        }

        // Этот метод обходит дерево элементов с корнем в элементе n,
        // отыскивает теги с <h1> по <h6> и добавляет их в массив разделов.
        function findSections(n, sects) {
            // Обойти все дочерние узлы элемента n
            for(var m = n.firstChild; m != null; m = m.nextSibling) {
                // Пропустить узлы, которые не являются элементами.
                if (m.nodeType != 1 /* Node.Element_NODE */) continue;
                // Пропустить контейнерный элемент, поскольку он может иметь свой заголовок
                if (m == container) continue;
                // С целью оптимизации пропустить теги <p>, поскольку
                // предполагается, что заголовки не могут появляться внутри
                // абзацев. (Кроме того, можно было бы пропустить списки,
                // теги <pre> и прочие, но тег <p> - самый распространенный.)
                if (m.tagName == "P") continue; // Оптимизация

                // Узел не был пропущен - проверить, не является ли он заголовком.
                // Если это заголовок, добавить его в массив. Иначе просмотреть
                // рекурсивно содержимое узла.
                // Обратите внимание: модель DOM основана на интерфейсах,
                // а не на классах, потому нельзя выполнить проверку
                // на принадлежность (m instanceof HTMLHeadingElement).
                if (m.tagName.length==2 && m.tagName.charAt(0)=="H")
                    sects.push(m);
                else
                    findSections(m, sects);
            }
        }
    }
    // Это текст по умолчанию для обратных ссылок на оглавление
```

```
maketoc.backlinkText = "Contents";

// Зарегистрировать функцию maketoc() как обработчик события onload
if (window.addEventListener) window.addEventListener("load", maketoc, false);
else if (window.attachEvent) window.attachEvent("onload", maketoc);
```

15.10. Получение выделенного текста

Иногда удобно иметь возможность определять, какой участок текста документа выделен пользователем. Хотя эта область слабо стандартизована, тем не менее возможность получать выделенный текст поддерживается во всех современных броузерах. В примере 15.12 показано, как это делается.

Пример 15.12. Получение выделенного текста

```
function getSelectedText() {
    if (window.getSelection) {
        // Этот прием, скорее всего, будет стандартизирован.
        // getSelection() возвращает объект Selection,
        // который в этой книге не описывается.
        return window.getSelection().toString();
    }
    else if (document.getSelection) {
        // Это более старый простейший прием, который возвращает строку
        return document.getSelection( );
    }
    else if (document.selection) {
        // Этот прием используется в IE. В этой книге не описываются
        // ни свойство selection, ни объект TextRange, присутствующие в IE.
        return document.selection.createRange().text;
    }
}
```

Программный код этого примера можно только принять на веру. Объекты `Selection` и `TextRange`, используемые в примере, в данной книге не рассматриваются. Просто на момент написания этих строк их прикладной интерфейс (API) был слишком сложен и к тому же не стандартизован. Однако сама операция получения выделенного текста настолько проста и востребована, что определенно есть смысл проиллюстрировать ее. Она может использоваться, например, в букмарклетах (см. раздел 13.4.1), для организации поиска выделенного текста в поисковых системах или на сайте. Так, следующая HTML-ссылка пытается отыскать выделенный фрагмент текста в виртуальной энциклопедии (Wikipedia). Если поместить в закладку эту ссылку и URL-адрес со спецификатором `javascript:`, закладка превратится в букмарклет:

```
<a href="javascript:
    var q;
    if (window.getSelection) q = window.getSelection().toString();
    else if (document.getSelection) q = document.getSelection();
    else if (document.selection) q = document.selection.createRange().text;
    void window.open('http://en.wikipedia.org/wiki/' + q);
">
Найти выделенный текст в Wikipedia
</a>
```

В примере 15.12 есть одна неточность. Метод `getSelection()` объектов `Window` и `Document` не возвращает выделенный текст, если он находится внутри элементов формы `<input>` или `<textarea>`: он возвращает только тот текст, который выделен в теле самого документа. В то же время свойство `document.selection` в IE возвращает текст, выделенный в любом месте документа.

В Firefox элементы ввода текста определяют свойства `selectionStart` и `selectionEnd`, которые могут использоваться для получения выделенного текста или для выделения фрагментов текста. Например:

```
function getTextFieldSelection(e) {
    if (e.selectionStart != undefined && e.selectionEnd != undefined) {
        var start = e.selectionStart;
        var end = e.selectionEnd;
        return e.value.substring(start, end);
    }
    else return ""; // Не поддерживается данным броузером
}
```

15.11. IE 4 DOM

Хотя броузер IE 4 несовместим с W3C DOM, он поддерживает API с множеством возможностей, аналогичных W3C DOM. Броузер IE 5 и более поздние версии поддерживают IE 4 DOM, некоторые другие броузеры также имеют, по крайней мере частично, совместимость с этой моделью. К моменту написания этих строк броузер IE 4 уже вышел из широкого употребления. При создании новых JavaScript-сценариев уже, как правило, не требуется соблюдать совместимость с IE 4, поэтому значительная часть материала с описанием IE 4 DOM удалена из четвертой части этого издания книги. Однако объем программного кода, соответствующего спецификации IE 4 DOM, еще достаточно велик, поэтому есть смысл хотя бы вкратце ознакомиться с этим прикладным программным интерфейсом.

15.11.1. Обход документа

Стандарт W3C DOM указывает, что во всех объектах `Node`, включая объект `Document` и все объекты `Element`, имеется массив `childNodes[]`, содержащий дочерние узлы данного узла. IE 4 не поддерживает `childNodes[]`, но предоставляет очень похожий на него массив `children[]` в объектах `Document` и `HTMLElement`. Поэтому для обхода всех HTML-элементов в документе IE 4 легко написать рекурсивную функцию, аналогичную показанной в примере 15.2.

Тем не менее между массивом `children[]` в IE 4 и стандартным массивом `childNodes[]` в W3C DOM есть одно существенное различие. В IE 4 нет типа узла `Text`, и в нем строки текста не рассматриваются как дочерние узлы. Следовательно, тег `<p>`, содержащий только обычный текст без разметки, в IE 4 имеет пустой массив `children[]`. Однако как мы скоро увидим, текстовое содержимое тега `<p>` в IE 4 доступно через свойство `innerText`.

15.11.2. Поиск элементов в документе

IE 4 не поддерживает методы `getElementById()` и `getElementsByTagName()` объекта `Document`. Вместо этого у объекта `Document` и у всех элементов документа есть мас-

сив свойств с именем `all[]`. Как следует из имени, этот массив представляет *все* (*all*) элементы документа или все элементы, содержащиеся в другом элементе. Обратите внимание, что массив `all[]` не просто представляет дочерние узлы документа или элемента – он содержит всех потомков независимо от глубины вложенности.

Массив `all[]` может использоваться несколькими способами. Если он индексируется с помощью целого индекса *n*, то возвращает *n*+1-й элемент документа или родительского элемента. Например:

```
var e1 = document.all[0]; // Первый элемент документа
var e2 = e1.all[4];       // Пятый элемент в элементе 1
```

Элементы нумеруются в порядке их расположения в исходном тексте документа. Обратите внимание на одно существенное различие между API IE 4 и стандартом DOM: в IE нет понятия текстовых узлов, поэтому массив `all[]` содержит только элементы документа, но не текст внутри них.

Обычно значительно полезнее иметь возможность ссылаться на элементы документа по именам, чем по номерам. Эквивалентом вызова метода `getElementById()` в IE 4 является индексирование массива `all[]` с помощью строки, а не числа. Когда вы пользуетесь этой возможностью, IE 4 возвращает элемент, у которого атрибут `id` или `name` равен указанному значению. Если имеется более одного такого элемента (что возможно, т. к. часто имеется несколько элементов формы, например переключателей, с одинаковыми значениями атрибута `name`), результатом будет массив этих элементов. Например:

```
var specialParagraph = document.all["special"];
var radioboxes = form.all["shippingMethod"]; // Может вернуть массив
```

JavaScript позволяет также записывать эти выражения, указывая индекс массива как имя свойства:

```
var specialParagraph = document.all.special;
var radioboxes = form.all.shippingMethod;
```

Подобное использование массива `all[]` предоставляет ту же базовую функциональность, что и методы `getElementById()` и `getElementsByName()`. Основное отличие состоит в том, что массив `all[]` объединяет возможности этих двух методов, что может привести к проблемам при непреднамеренном применении одинаковых значений атрибутов `id` и `name` для несвязанных между собой элементов.

У массива `all[]` есть необычный метод `tags()`, который может использоваться для получения массива элементов по имени тега:

```
var lists = document.all.tags("UL"); // Ищет все теги <ul> в документе
var items = lists[0].all.tags("LI"); // Ищет все теги <li> внутри первого <ul>
```

Этот синтаксис IE 4 предоставляет практически ту же функциональность, что и метод `getElementsByTagName()` объектов `Document` и `Element` в DOM. Обратите внимание, что в IE 4 имя тега должно содержать только строчные буквы.

15.11.3. Модификация документов

Как и W3C DOM, IE 4 предоставляет доступ к атрибутам HTML-тегов как к свойствам соответствующих объектов `HTMLElement`. Поэтому можно изменить документ, открытый в IE 4, путем динамического изменения HTML-атрибутов. Если модификация атрибута приводит к изменению размера какого-либо элемента, документ переформатируется, чтобы соответствовать новым размерам элемента. Объект `HTMLElement` в IE 4 определяет также методы `setAttribute()`, `getAttribute()` и `removeAttribute()`. Они аналогичны одноименным методам, определенным в объекте `Element` стандартного прикладного DOM-интерфейса.

Стандарт W3C DOM определяет прикладной интерфейс, позволяющий создавать новые узлы, вставлять узлы в дерево документа, менять родителей для узлов и перемещать узлы внутри дерева. IE 4 не может этого делать. Вместо этого во всех объектах `HTMLElement` в IE 4 определено свойство `innerHTML`. Установка этого свойства равным строке HTML-текста позволяет заменить содержимое элемента чем угодно. Поскольку свойство `innerHTML` представляет собой столь мощное средство, оно реализовано во всех современных броузерах и скорее всего будет включено в стандарт DOM. Порядок использования и описание свойства `innerHTML` приводятся в разделе 15.8.2.

IE 4 определяет также несколько сходных свойств и методов. Свойство `outerHTML` заменяет содержимое элемента и целиком сам элемент указанной HTML-строкой. Свойства `innerText` и `outerText` аналогичны свойствам `innerHTML` и `outerHTML` за исключением того, что рассматривают строку как обычный текст и не анализируют ее как HTML-текст. И наконец, методы `insertAdjacentHTML()` и `insertAdjacentText()` не затрагивают сам элемент, но вставляют новое текстовое содержимое или содержимое в формате HTML рядом (до или после, внутри или снаружи) с элементом. Эти свойства и функции используются не настолько часто, как `innerHTML`, и не реализованы в Firefox.

16

CSS и DHTML

Каскадные таблицы стилей (Cascading Style Sheets, CSS) – это стандарт визуального представления HTML- и XML-документов. Теоретически структуру документа следует задавать путем HTML-разметки, сопротивляясь искушению применять устаревшие HTML-теги, такие как `<font>`, поскольку для задания стилей существуют CSS-таблицы, определяющие, как именно должны отображаться структурные элементы документа. Например, CSS позволяет указать, что заголовки первого уровня, определяемые тегами `<h1>`, должны отображаться в верхнем регистре, шрифтом sans-serif с полужирным начертанием и размером в 24 пункта, выравнивание по центру.

Технология CSS ориентирована на дизайнеров, а также всех тех, кто заботится о точном визуальном отображении HTML-документов. Она интересна программистам, пишущим на клиентском языке JavaScript, т. к. объектная модель документа позволяет при помощи сценариев применять стили к отдельным элементам документа. Совместное применение технологий CSS и JavaScript обеспечивает получение разнообразных визуальных эффектов, не совсем точно называемых динамическим языком HTML (Dynamic HTML, DHTML).[1]

Способность манипулировать CSS-стилями в сценариях позволяет динамически изменять цвет, шрифт и прочие элементы оформления. Еще важнее, что CSS-стили дают возможность устанавливать и изменять размер элементов и даже скрывать или показывать их. Это значит, что технология DHTML может использоваться для создания анимированных переходов, когда, например, содержимое документа «выплывает» из-за границ экрана или структурированный список разворачивается и сворачивается, благодаря чему пользователь может управлять объемом выводимой информации.

Эта глава начинается с обзора CSS. Затем рассказывается о том, как с помощью CSS-стилей задать позицию элементов документа и режим их видимости. После этого описываются приемы манипулирования CSS-стилями в сценариях. Наибо-

---

[1] Во многих сложных DHTML-эффектах используются также приемы обработки событий, которые мы рассмотрим в главе 17.

лее типичный прием при работе со стилями заключается в изменении значения свойства `style` отдельных элементов документа. Реже используются приемы, основанные на косвенном изменении стилей элементов путем определения CSS-классов, применяемых к этим элементам. Достигается это изменением значения свойства `className`. Существует также возможность непосредственного манипулирования таблицами стилей. Заканчивается глава обсуждением механизмов включения и отключения таблиц стилей, а также получения, добавления и удаления правил для таблиц стилей.

16.1. Обзор CSS

Стили в CSS-таблицах задаются в виде разделенных точкой с запятой пар атрибутов, состоящих из имени и значения. Между собой имя и значение разделяются двоеточием. Например, следующий стиль определяет полужирный подчеркнутый текст синего цвета:

```
font-weight: bold; color: blue; text-decoration: underline;
```

Стандарт CSS описывает множество атрибутов стилей. В табл. 16.1 перечислены все атрибуты, кроме тех, которые в настоящее время практически не поддерживаются. Возможно, на данном этапе эти атрибуты и их значения покажутся вам непонятными. Однако когда вы больше узнаете о CSS-стилях и станете применять их в документах и сценариях, эта информация будет вам полезной в качестве справочника. Более полную документацию по CSS можно найти в выпущенных издательством O'Reilly книгах «Cascading Style Sheets: The Definitive Guide»[1] Эрика Мейера (Eric Meyer) и «Dynamic HTML: The Definitive Guide» Денни Гудмена (Danny Goodman). Можно также прочитать спецификацию по адресу *http://www.w3c.org/TR/CSS21/*.

Во втором столбце табл. 16.1 показаны допустимые значения для каждого атрибута стиля. Здесь используется та же грамматика, что и в спецификации CSS. Слова, написанные `моноширинным шрифтом`, являются ключевыми и должны присутствовать в документе в том же виде, в котором они приведены в таблице. Слова, выделенные *курсивом*, описывают тип данных, например *string* (строка) или *length* (длина). Обратите внимание, что тип *length* – это число, за которым следует спецификация единицы измерения, например `px` (пикселы). Описания других типов можно найти в литературе по CSS. Слова, набранные *`моноширинным курсивом`*, определяют набор значений, допустимых для некоторого другого CSS-атрибута. Помимо значений, представленных в таблице, каждый атрибут стиля может иметь значение `inherit`, указывающее, что атрибут должен наследовать значение родительского элемента.

Значения, разделенные символом `|`, являются альтернативными – требуется указать только одно из них. Значения, разделенные символами `||`, представляют собой варианты – необходимо указать хотя бы одно из них, но можно указать и несколько (в любом порядке). Квадратные скобки `[]` предназначены для объединения значений в группы. Звездочка `*` означает, что предыдущее значение или группа может присутствовать ноль или более раз, знак `+` говорит о том, что

[1] Эрик Мейер «CSS – каскадные таблицы стилей. Подробное руководство», 3-е издание. – Пер. с англ. – СПб.: Символ-Плюс, 2008.

предыдущее значение или группа может присутствовать один или более раз, а вопросительный знак ? указывает, что предыдущее значение не обязательно и может присутствовать ноль или более раз. Число в фигурных скобках задает количество повторений. Например, {2} означает что предыдущее значение должно быть повторено дважды, а {1,4} – что предыдущее значение должно присутствовать не менее одного раза и не более четырех раз. (Этот синтаксис повторения может показаться вам знакомым, поскольку соответствует синтаксису регулярных JavaScript-выражений, описываемому в главе 11.)

Таблица 16.1. Атрибуты CSS-стилей и их значения

Имя	Значение
background	[*background-color* || *background-image* || *background-repeat* || *background-attachment* || *background-position*]
background-attachment	scroll | fixed
background-color	***color*** | transparent
background-image	url (***url***) | none
background-position	[[***percentage*** | ***length***]{1,2} | [[top | center | bottom] || [left | center | right]]]
background-repeat	repeat | repeat-x | repeat-y | no-repeat
border	[*border-width* || *border-style* || ***color***]
border-collapse	collapse | separate
border-color	***color***{1,4} | transparent
border-spacing	***length length?***
border-style	[none | hidden | dotted | dashed | solid | double | groove | ridge | inset | outset]{1,4}
border-top border-right border-bottom border-left	[*border-top-width* || *border-style* || [***color*** | transparent]]
border-top-color border-right-color border-bottom-color border-left-color	***color*** | transparent
border-top-style border-right-style border-bottom-style border-left-style	none | hidden | dotted | dashed | solid | double | groove | ridge | inset | outset
border-top-width border-right-width border-bottom-width border-left-width	thin | medium | thick | ***length***

Имя	Значение
border-width	[thin | medium | thick | ***length***]{1,4}
bottom	***length*** | ***percentage*** | auto
caption-side	top | bottom
clear	none | left | right | both
clip	[rect([***length*** | auto]{4})] | auto
color	***color***
content	[***string*** | url(***url***) | ***counter*** | attr(*attribute-name*) | open-quote | close-quote | no-open-quote | no-close-quote]+ | normal
counter-increment	[***identifier integer?***]+ | none
counter-reset	[***identifier integer?***]+ | none
cursor	[[url(***url***) ,]* [auto | crosshair | default | pointer | progress | move | e-resize | ne-resize | nw-resize | n-resize | se-resize | sw-resize | s-resize | w-resize | text | wait | help]]
direction	ltr | rtl
display	inline | block | inline-block | list-item | run-in | table | inline-table | table-row-group | table-header-group | table-footer-group | table-row | table-column-group | table-column | table-cell | table-caption | none
empty-cells	show | hide
float	left | right | none
font	[[*font-style* || *font-variant* || *font-weight*]? *font-size* [/ *line-height*]? *font-family*] | caption | icon | menu | message-box | small-caption | status-bar
font-family	[[***family-name*** | serif | sans-serif | monospace | cursive | fantasy],]+
font-size	xx-small | x-small | small | medium | large | x-large | xx-large | smaller | larger | ***length*** | ***percentage***
font-style	normal | italic | oblique
font-variant	normal | small-caps
font-weight	normal | bold | bolder | lighter | 100 | 200 | 300 | 400 | 500 | 600 | 700 | 800 | 900
height	***length*** | ***percentage*** | auto
left	***length*** | ***percentage*** | auto
letter-spacing	normal | ***length***
line-height	normal | ***number*** | ***length*** | ***percentage***
list-style	[*list-style-type* || *list-style-position* ||м*list-style-image*]
list-style-image	url(***url***) | none

Таблица 16.1 (продолжение)

Имя	Значение
list-style-position	inside | outside
list-style-type	disc | circle | square | decimal | decimal-leading-zero | lowerroman | upper-roman | lower-greek | lower-alpha | lower-latin | upper-alpha | upper-latin | hebrew | armenian | georgian | cjkideographic | hiragana | katakana | hiragana-iroha | katakanairoha | none
margin	[***length*** | ***percentage*** | auto]{1,4}
margin-top	***length*** | ***percentage*** | auto
margin-right	
margin-bottom	
margin-left	
marker-offset	***length*** | auto
max-height	***length*** | ***percentage*** | none
max-width	***length*** | ***percentage*** | none
min-height	***length*** | ***percentage***
min-width	***length*** | ***percentage***
outline	[*outline-color* || *outline-style* || *outline-width*]
outline-color	***color*** | invert
outline-style	none | hidden | dotted | dashed | solid | double | groove | ridge | inset | outset
outline-width	thin | medium | thick | ***length***
overflow	visible | hidden | scroll | auto
padding	[***length*** | ***percentage***]{1,4}
padding-top	***length*** | ***percentage***
padding-right	
padding-bottom	
padding-left	
page-break-after	auto | always | avoid | left | right
page-break-before	auto | always | avoid | left | right
page-break-inside	avoid | auto
position	static | relative | absolute | fixed
quotes	[***string string***]+ | none
right	***length*** | ***percentage*** | auto
table-layout	auto | fixed
text-align	left | right | center | justify
text-decoration	none | [underline || overline || line-through || blink]

Имя	Значение
text-indent	*length* | *percentage*
text-transform	capitalize | uppercase | lowercase | none
top	*length* | *percentage* | auto
unicode-bidi	normal | embed | bidi-override
vertical-align	baseline | sub | super | top | text-top | middle | bottom | text-bottom | *percentage* | *length*
visibility	visible | hidden | collapse
white-space	normal | pre | nowrap | pre-wrap | pre-line
width	*length* | *percentage* | auto
word-spacing	normal | *length*
z-index	auto | *integer*

Стандарт CSS позволяет объединять определенные атрибуты стилей, которые часто задаются вместе, с помощью специальных атрибутов-сокращений. Например, атрибуты font-family, font-size, font-style и font-weight могут быть одновременно установлены с помощью одного атрибута font:

```
font: bold italic 24pt helvetica;
```

Атрибуты margin и padding являются сокращениями атрибутов, задающих поля, отступы и границы отдельной стороны элемента. Поэтому вместо атрибута margin можно задать атрибуты margin-left, margin-right, margin-top и margin-bottom. То же самое относится к атрибуту padding.

16.1.1. Применение правил стиля к элементам документа

Атрибуты стиля применяются к элементам документа несколькими способами. Один из способов состоит в том, чтобы указать их в атрибуте style HTML-тега. Например, поля отдельного абзаца можно установить так:

```
<p style="margin-left: 1in; margin-right: 1in;">
```

Одной из главных задач CSS-стилей является отделение содержимого и структуры документа от его представления. Задание стилей с помощью атрибута style в отдельных HTML-тегах не способствует решению этой задачи (хотя может быть полезным приемом DHTML-программирования). Чтобы добиться разделения структуры и представления, применяются *таблицы стилей* (*stylesheets*), объединяющие всю информацию о стилях в одном месте. Таблица CSS-стилей состоит из набора правил стиля. Каждое правило начинается с селектора, задающего элемент или элементы документа, к которым применяется правило; за селектором следует набор атрибутов стиля и их значений, заключенный в фигурные скобки. Простейший вид правил определяет стиль для одного или нескольких конкретных тегов. Например, вот правило, устанавливающее поля и цвет фона для тега <body>:

```
body { margin-left: 30px; margin-right: 15px; background-color: #ffffff }
```

Следующее правило определяет, что текст внутри заголовков <h1> и <h2> должен быть выровнен по центру:

```
h1, h2 { text-align: center; }
```

Обратите внимание на использование в примере запятой для разделения имен тегов, к которым должны применяться стили. Если запятая отсутствует, селектор задает контекстное правило, применяемое только в том случае, если один тег вложен в другой. Например, следующие правила указывают, что теги <blockquote> выводятся курсивом, но текст внутри тега <i>, находящегося внутри <blockquote>, должен выводиться прямым шрифтом:

```
blockquote { font-style: italic; }
blockquote i { font-style: normal; }
```

Другой тип правил в таблице стилей задает *классы* элементов, к которым должны применяться стили, и селектор в этом случае иной. Класс элемента определяется атрибутом class HTML-тега. Например, следующее правило указывает, что любой тег с атрибутом class="attention" должен отображаться шрифтом полужирного начертания:

```
.attention { font-weight: bold; }
```

Селекторы классов могут объединяться с селекторами имен тегов. Следующее правило указывает, что если в теге <p> есть атрибут class="attention", тег должен отображаться красным цветом (помимо полужирного начертания, как определено предыдущим правилом):

```
p.attention { color: red; }
```

И наконец, таблицы стилей содержат правила, применяемые к отдельным элементам, имеющим заданное значение атрибута id. Следующее правило указывает, что элемент с атрибутом id, равным p1, не должен отображаться:

```
#p1 { visibility: hidden; }
```

Мы встречали атрибут id раньше: он применяется с DOM-функцией getElementById() для получения отдельных элементов документа. Как можно предположить, этот вид правил в таблице стилей может с успехом использоваться для управления стилем отдельного элемента. При наличии такого правила сценарий может менять значение атрибута visibility со значения hidden (скрытый) на visible (видимый), вызывая динамическое появление элемента. Как это делается, показано в этой главе далее.

Стандарт CSS определяет еще целый ряд других селекторов, помимо тех, что были продемонстрированы здесь, и некоторые из них поддерживаются современными броузерами. За дополнительной информацией обращайтесь к спецификации CSS или к справочным руководствам.

16.1.2. Связывание таблиц стилей с документами

Таблицу стилей можно внедрить в HTML-документ, поместив ее между тегами <style> и </style> в заголовке документа, или сохранить в собственном файле, сославшись на файл из HTML-документа с помощью тега <link>. Например:

```
<html>
<head><title>Тестовый документ</title>
<style type="text/css">
body { margin-left: 30px; margin-right: 15px; background-color: #ffffff }
p { font-size: 24px; }
</style>
</head>
<body><p>Проверка, проверка и еще раз проверка</p></body>
</html>
```

Если таблица стилей используется более чем одним документом на веб-сайте, лучше хранить ее в виде отдельного файла без охватывающих HTML-тегов. Этот CSS-файл может быть подключен к HTML-странице. Однако в отличие от тега `<script>`, тег `<style>` не имеет атрибута `src`. Поэтому, чтобы подключить таблицу стилей к HTML-документу, необходимо воспользоваться тегом `<link>`:

```
<html>
<head><title>Тестовый документ</title>
<link rel="stylesheet" href="mystyles.css" type="text/css">
</head>
<body><p>Проверка, проверка и еще раз проверка</p></body>
</html>
```

Тег `<link>` может использоваться для задания альтернативной таблицы стилей. Некоторые броузеры (такие как Firefox) позволяют выбирать из имеющихся альтернатив (в меню с помощью пункта Вид>Стиль страницы). Например, вы могли бы предусмотреть альтернативную таблицу стилей для посетителей сайта, которые предпочитают крупный шрифт и высококонтрастное цветовое оформление:

```
<link rel="alternate stylesheet" href="largetype.css" type="text/css"
      title="Крупный шрифт"> <!-- заголовок отображается в меню -->
```

Если к веб-странице с помощью тега `<style>` подключается особенная таблица стилей, то чтобы включить в эту таблицу общий CSS-файл, можно воспользоваться CSS-директивой `@import`:

```
<html>
<head><title>Тестовый документ</title>
<style type="text/css">
@import "mystyles.css"; /* импорт общей таблицы стилей */
p { font-size: 48px; }  /* переопределение импортированных стилей */
</style>
</head>
<body><p>Проверка, проверка и еще раз проверка</p></body>
</html>
```

16.1.3. Каскад правил

Вспомните, что буква «C» в аббревиатуре CSS обозначает «cascade» (каскадная). Этот термин указывает, что правила стилей, применяемые к конкретному элементу документа, могут быть получены из каскада различных источников. Каждый веб-броузер, обычно имеющий собственные стили, применяемые по умолчанию ко всем HTML-элементам, может разрешить пользователю переопределять эти значения с помощью пользовательской таблицы стилей. Автор документа

может определять таблицы стилей с помощью тегов <style> или внешних файлов, связанных с другими таблицами стилей или импортированных в них. Автор также может определять встроенные стили для индивидуальных элементов с помощью HTML-атрибута style.

Спецификация CSS включает полный набор правил, определяющих, какие правила из каскада имеют приоритет над другими. Если не вдаваться в детали, просто запомните, что пользовательская таблица стилей переопределяет таблицу стилей броузера, применяемую по умолчанию, авторская таблица стилей переопределяет пользовательскую таблицу стилей, а встроенные стили переопределяют все. Исключение из этого общего правила состоит в том, что пользовательские атрибуты стилей, значения которых включают модификатор !important, переопределяют авторские стили. Если в таблице стилей к элементу применяется более одного правила, то стили, определенные по наиболее конкретному правилу, переопределяют конфликтующие стили, заданные по менее конкретным правилам. Правила, задающие атрибут id элемента, являются наиболее конкретными. Правила, задающие атрибут class, – следующие по конкретности. Правила, задающие только имена тегов, – наименее конкретные, а правила, задающие несколько имен вложенных тегов, более конкретны, чем правила, задающие только одно имя тега.

16.1.4. Версии CSS

CSS – это довольно старый стандарт. В декабре 1996 года был принят стандарт CSS1 и определены атрибуты для задания цвета, шрифта, полей, рамок и других базовых стилей. Такие старые броузеры, как Netscape 4 и Internet Explorer 4, поддерживают CSS1, по крайней мере, частично. Вторая редакция стандарта, CSS2, была принята в мае 1998 года; она определяет более развитые характеристики, наиболее важная из которых – абсолютное позиционирование элементов. К моменту написания этих строк характеристики, предусматриваемые стандартом CSS2, поддерживались практически всеми современными броузерами. Процесс стандартизации ключевых характеристик позиционирования начинался задолго до появления стандарта CSS2 как часть отдельного проекта CSS-Positioning (CSS-P), поэтому DHTML-характеристики доступны во всех современных броузерах. (Эти важные характеристики позиционирования обсуждаются в этой же главе далее.)

Работа над CSS продолжается и в настоящее время. К моменту написания этих строк спецификация CSS2.1 была практически завершена. В ней уточнены положения спецификации CSS2, исправлены ошибки, из нее убраны стили, не нашедшие воплощения в броузерах. Третья версия CSS разделена на специализированные модули, стандартизируемые по отдельности. Стандартизация некоторых CSS3-модулей уже близка к завершению, и в части броузеров уже делается попытка реализовать отдельные функциональные возможности спецификации CSS3, такие как стиль opacity. Спецификации и рабочие проекты CSS можно найти по адресу *http://www.w3.org/Style/CSS/*.

16.1.5. Пример CSS-таблицы

Пример 16.1 представляет собой HTML-файл, определяющий и использующий таблицу стилей. Этот пример иллюстрирует описанные ранее правила стилей,

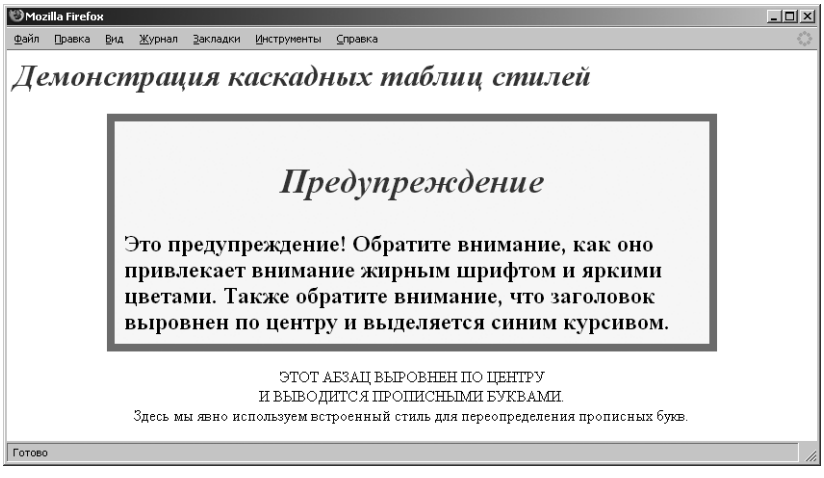

***Рис. 16.1.** Веб-страница, оформленная с помощью CSS*

базирующиеся на имени тега, классе и идентификаторе, а также содержит встроенный стиль, определяемый атрибутом `style`. На рис. 16.1 показано, как этот код визуализируется в броузере. Помните, что данный пример приведен здесь только для знакомства с синтаксисом и возможностями CSS. Полное описание CSS-стилей выходит за рамки темы этой книги.

Пример 16.1. Определение и использование каскадных таблиц стилей

```
<head>
<style type="text/css">
/* Указывает, что заголовки отображаются курсивом синего цвета. */
h1, h2 { color: blue; font-style: italic }

/*
 * Любой элемент с атрибутом class="WARNING" отображается крупными жирными
 * символами, имеет большие поля и желтый фон с жирной красной рамкой.
 */
.WARNING {
          font-weight: bold;
          font-size: 150%;
          margin: 0 1in 0 1in; /* сверху справа, снизу слева */
          background-color: yellow;
          border: solid red 8px;
          padding: 10px; /* 10 пикселов со всех 4 сторон */
}

/*
 * Текст заголовков h1 и h2 внутри элементов с атрибутом class="WARNING"
 * должен быть выровнен по центру, в дополнение к выделению синим курсивом.
 */
.WARNING h1, .WARNING h2 { text-align: center }

/* Отдельный элемент с атрибутом id="P23" отображается прописными буквами по центру.
 */
```

```
#P23 {
      text-align: center;
      text-transform: uppercase;
}
</style>
</head>
<body>
<h1>Демонстрация использования каскадных таблиц стилей</h1>
<div class="WARNING">
<h2>Предупреждение</h2>
Это предупреждение!
Обратите внимание, как оно привлекает внимание жирным шрифтом и яркими
цветами. Также обратите внимание, что заголовок выровнен по центру
и выделяется синим курсивом.
</div>

<p id="P23">
Этот абзац выровнен по центру<br>
и выводится прописными буквами.<br>
<span style="text-transform: none">
Здесь мы явно используем встроенный стиль для переопределения прописных букв.
</span>
</p>
</body>
```

16.2. CSS для DHTML

Для разработчиков DHTML-содержимого в CSS важнее всего то, что таблицы стилей посредством атрибутов обычных CSS-стилей позволяют задавать режим видимости, размер и точную позицию отдельных элементов документа. Другие CSS-стили дают возможность определять порядок наложения слоев, степень прозрачности, вырезанные области, поля, отступы, рамки и цвета. Занимаясь DHTML-программированием, важно понимать, как работают эти атрибуты стилей. В табл. 16.2 они просто перечислены, а в последующих разделах описаны более подробно.

Таблица 16.2. Атрибуты позиционирования и видимости в CSS

Атрибут(ы)	Описание
`position`	Тип позиционирования, применяемый к элементу
`top`, `left`	Позиция верхнего и левого краев элемента
`bottom`, `right`	Позиция нижнего и правого краев элемента
`width`, `height`	Размер элемента
`z-index`	«Порядок в стеке» элемента относительно любых перекрывающих его элементов (третье измерение в положении элемента)
`display`	Режим отображения элемента
`visibility`	Режим видимости элемента
`clip`	«Область отсечения» элемента (отображаются только те части документа, которые находятся внутри этой области)

Атрибут(ы)	Описание
overflow	Определяет, что следует делать, если размер элемента больше, чем предоставленное ему место
margin, border, padding	Границы и рамки элемента
background	Цвет фона или фоновый рисунок для элемента
opacity	Степень непрозрачности (или прозрачности) элемента. Этот атрибут относится к стандарту CSS3 и поддерживается не всеми броузерами. Работающая альтернатива имеется для IE

16.2.1. Ключ к DHTML: абсолютное позиционирование

CSS-атрибут position задает тип позиционирования, применяемый к элементу. У этого атрибута четыре возможных значения:

static

Это значение, применяемое по умолчанию. Оно указывает, что элемент позиционируется статически в соответствии с нормальным порядком вывода содержимого документа (для большинства западных языков – слева направо и сверху вниз). Статически позиционированные элементы не являются DHTML-элементами и не могут позиционироваться с помощью атрибутов top, left и других. Для позиционирования элемента документа с использованием технологии DHTML сначала нужно установить его атрибут position равным одному из трех других значений.

absolute

Это значение позволяет задать абсолютную позицию элемента относительно содержащего его элемента. Такие элементы позиционируются независимо от всех остальных элементов и не являются частью потока статически позиционированных элементов. Абсолютно позиционированный элемент позиционируется либо относительно тела документа, либо, если он вложен в другой абсолютно позиционированный элемент, относительно этого элемента. Это наиболее распространенный в DHTML тип позиционирования. В IE 4 абсолютное позиционирование поддерживается лишь для некоторых элементов. Чтобы организовать абсолютное позиционирование в устаревших броузерах, требуется обернуть абсолютно позиционируемые элементы в теги или .

fixed

Это значение позволяет зафиксировать положение элемента относительно окна броузера. Элементы с фиксированным позиционированием не прокручиваются с остальной частью документа и поэтому могут служить для имитации фреймов. Как и абсолютно позиционированные, фиксировано позиционированные элементы не зависят от всех остальных элементов и не являются частью потока вывода документа. Фиксированное позиционирование поддерживается большинством современных броузеров, исключая IE 6.

relative

Если атрибут position установлен в значение relative, элемент располагается в соответствии с нормальным потоком вывода, а затем его положение смеща-

ется относительно его обычного положения в потоке. Пространство, выделенное для элемента в нормальном потоке вывода документа, остается выделенным для него, и элементы, расположенные со всех сторон от него, не смыкаются для заполнения этого пространства и не выталкиваются с новой позиции элемента.

Установив для атрибута `position` элемента значение, отличное от `static`, можно задать положение элемента с помощью произвольной комбинации атрибутов `left`, `top`, `right` и `bottom`. Наиболее распространенный прием позиционирования – это указание атрибутов `left` и `top`, задающих расстояние от левого края элемента-контейнера (обычно самого документа) до левого края позиционируемого элемента и расстояние от верхнего края контейнера до верхнего края элемента. Так, чтобы поместить элемент на расстоянии 100 пикселов от левого края и 100 пикселов от верхнего края документа, можно задать CSS-стили в атрибуте `style` следующим образом:

```
<div style="position: absolute; left: 100px; top: 100px;">
```

Элемент-контейнер, относительно которого позиционируется динамический элемент, не обязательно совпадает с элементом-контейнером, содержащим динамический элемент в исходном тексте документа. Динамические элементы не являются частью обычного потока вывода элементов, поэтому их положение не задается относительно статических элементов-контейнеров, внутри которых они определены. Большинство динамических элементов позиционируются относительно самого документа (тега `<body>`). Исключение составляют динамические элементы, определенные внутри других динамических элементов. В этом случае вложенный динамический элемент позиционируется относительно ближайшего динамического предка. Если предполагается позиционировать элемент относительно контейнера, который является частью потока вывода документа, следует установить правило `position:relative` для элемента-контейнера, а в качестве значений атрибутов `top` и `left` указать `0px`. В этом случае контейнер будет позиционироваться динамически и останется при этом на обычном месте в потоке вывода документа. Любые абсолютно позиционируемые вложенные элементы позиционируются относительно элемента-контейнера.

Чаще всего задается положение верхнего левого угла элемента с помощью атрибутов `left` и `top`, но с помощью атрибутов `right` и `bottom` можно задать положение нижнего и правого краев элемента относительно нижнего и правого краев элемента-контейнера. Например, при помощи следующих стилей можно указать, чтобы правый нижний угол элемента находился в правом нижнем углу документа (предполагая, что он не вложен в другой динамический элемент):

```
position: absolute; right: 0px; bottom: 0px;
```

Для того чтобы верхний край элемента располагался в 10 пикселах от верхнего края окна, а правый – в 10 пикселах от правого края окна, можно использовать такие стили:

```
position: absolute; right: 10px; top: 10px;
```

Помимо позиции элементов CSS позволяет указывать их размер. Чаще всего это делается путем задания значений атрибутов стиля `width` и `height`. Например, следующий HTML-код создает абсолютно позиционированный элемент без содер-

жимого. Значения атрибутов `width`, `height` и `background-color` указаны так, чтобы он отображался в виде маленького синего квадрата:

```
<div style="position: absolute; top: 10px; left: 10px;
            width: 10px; height: 10px; background-color: blue">
</div>
```

Другой способ определения ширины элемента состоит в одновременном задании атрибутов `left` и `right`. Аналогично можно задать высоту элемента, одновременно указав оба атрибута, `top` и `bottom`. Однако если задать значения для `left`, `right` и `width`, то атрибут `width` переопределяет атрибут `right`, а если ограничивается высота элемента, то атрибут `height` имеет приоритет перед `bottom`.

Имейте в виду, что задавать размер каждого динамического элемента не обязательно. Некоторые элементы, такие как изображения, имеют собственный размер. Кроме того, для динамических элементов, включающих текст или другое потоковое содержимое, часто достаточно указать желаемую ширину элемента и разрешить автоматическое определение высоты в зависимости от размещения содержимого элемента.

В предыдущих примерах значения атрибутов позиционирования и размера задавались с суффиксом «px», означающим «pixels» (пикселы). Стандарт CSS допускает указание размерности в некоторых других единицах, в том числе в дюймах («in»), сантиметрах («cm»), пунктах («pt») и единицах измерения высоты строки текущего шрифта («em»). Пикселы – это наиболее часто используемые в DHTML-программировании единицы измерения. Обратите внимание, что стандарт CSS требует указания единиц измерения. Некоторые броузеры могут предполагать пикселы, если единица измерения не указана, но на это не следует особенно полагаться.

CSS позволяет задать положение и размер элемента в процентах от размера элемента-контейнера или в абсолютных единицах с помощью описанных ранее единиц измерения. Следующий HTML-код создает пустой элемент с черной рамкой, имеющий ширину и высоту в половину элемента-контейнера (или окна броузера) и расположенный в этом элементе по центру:

```
<div style="position: absolute; left: 25%; top: 25%; width: 50%; height: 50%;
            border: 2px solid black">
</div>
```

16.2.2. Пример позиционирования средствами CSS: текст с тенью

В спецификации CSS2 включен атрибут `text-shadow`, позволяющий добиться эффекта отбрасывания тени текстовыми элементами. Данный атрибут реализован только в броузере Safari, остальные производители основных броузеров отказались от его поддержки. По этой причине он был удален из CSS2.1, но в CSS3 вновь рассматривается возможность его включения. Однако добиться эффекта тени можно и без атрибута `text-shadow`. Для этого достаточно использовать CSS-средства позиционирования и продублировать текст: первый раз для вывода собственно текста, второй (может быть третий и более раз) – для воспроизведения тени (или теней). Следующий пример воспроизводит эффект отбрасывания тени (рис. 16.2):

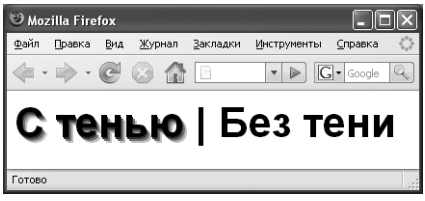

Рис. 16.2. Эффект отбрасывания тени, полученный с применением CSS-средств позиционирования

```
<div style="font: bold 32pt sans-serif;"> <!-- тени лучше выглядят с крупным шрифтом -->
<!-- Текст с тенью должен иметь относительное позиционирование, чтобы -->
<!-- можно было обеспечить смещение тени относительно нормального -->
<!-- положения текста в потоке вывода -->
<span style="position:relative;">
<!-- Далее определяются три тени различных цветов с использованием -->
<!-- абсолютного позиционирования для смещения их на разные расстояния -->
<!-- относительно обычного текста -->
<span style="position:absolute; top:5px; left:5px; color: #ccc">С тенью</span>
<span style="position:absolute; top:3px; left:3px; color: #888">С тенью</span>
<span style="position:absolute; top:1px; left:1px; color: #444">С тенью</span>
<!-- Далее следует собственно текст, который отбрасывает тень. Здесь -->
<!-- также имеет место относительное позиционирование, чтобы текст -->
<!-- выводился поверх своих теней -->
<span style="position:relative">С тенью</span>
</span>
| Без тени<!-- Для сравнения - этот текст не отбрасывает тень -->
</div>
```

Добавление эффекта тени вручную может оказаться делом достаточно сложным, к тому же это противоречит принципу отделения содержимого от представления. Решить проблему можно с помощью ненавязчивого JavaScript-кода. В примере 16.2 представлен JavaScript-модуль *Shadows.js*. В нем определяется функция `Shadows.addAll()`, которая просматривает документ (или часть документа) в поисках тегов с атрибутом `shadow`. Для всех найденных тегов анализируется значение атрибута `shadow` и с помощью DOM API к тексту, содержащемуся в тегах, добавляются тени. В качестве примера с помощью этого модуля можно попробовать воссоздать эффект, представленный на рис. 16.2:

```
<head><script src="Shadows.js"></script></head> <!-- подключить модуль -->
<body onload="Shadows.addAll( );"> <!-- добавить тени после загрузки -->
<div style="font: bold 32pt sans-serif;"> <!-- использовать крупный шрифт -->
<!-- Здесь находится атрибут shadow -->
<span shadow='5px 5px #ccc 3px 3px #888 1px 1px #444'>С тенью</span> | Без тени
</div>
</body>
```

Далее приводятся исходные тексты модуля *Shadows.js*. Примечательно, что основу данного сценария составляет DOM-код, что является обычным делом при использовании CSS. Однако есть одно исключение – в данном сценарии CSS-стили непосредственно не создаются, здесь просто устанавливаются CSS-атрибуты

в создаваемых элементах документа. Далее в этой главе мы более подробно поговорим о приемах создания CSS-стилей.

Пример 16.2. Создание эффекта тени с помощью ненавязчивого JavaScript-кода

```
/**
 * Shadows.js: создание эффекта тени в текстовых элементах средствами CSS.
 *
 * Этот модуль определяет единственный глобальный объект с именем Shadows.
 * Свойствами этого объекта являются две вспомогательные функции.
 *
 * Shadows.add(element, shadows):
 *   Добавляет заданные тени к заданному элементу. Первый аргумент -
 *   это элемент документа или идентификатор элемента. Данный элемент должен
 *   иметь единственный дочерний текстовый элемент. Эффект тени
 *   будет воспроизводиться в этом дочернем элементе.
 *   Порядок определения теней в аргументе shadows описывается далее.
 *
 * Shadows.addAll(root, tagname):
 *   Отыскивает все элементы-потомки заданного корневого элемента с указанным
 *   именем тега. Если в одном из найденных элементов обнаруживается атрибут
 *   shadow, вызывается функция Shadows.add(), которой передается элемент
 *   и значение атрибута shadow. Если имя тега не задано, выполняется проверка
 *   всех элементов. Если корневой элемент не задан, поиск ведется по всему
 *   документу. Данная функция вызывается единожды после загрузки документа.
 *
 * Порядок определения теней
 *
 * Тени задаются строкой в формате [x y color]+. Таким образом, одна или
 * более групп определяют смещение по оси x, смещение по оси y и цвет.
 * Каждое из этих значений должно соответствовать формату CSS. Если задается
 * более одной тени, самая первая тень оказывается самой нижней и ее перекрывают
 * все последующие тени. Например: "4px 4px #ccc 2px 2px #aaa"
 */
var Shadows = {};

// Добавить тени к единственному указанному элементу
Shadows.add = function(element, shadows) {
    if (typeof element == "string")
        element = document.getElementById(element);

    // Разбить строку по пробелам, предварительно отбросив начальные
    // и конечные пробелы
    shadows = shadows.replace(/^\s+/, "").replace(/\s+$/, "");
    var args = shadows.split(/\s+/);

    // Найти текстовый узел, в котором будет реализован эффект тени.
    // Этот модуль должен быть расширен, если необходимо добиться эффекта
    // во всех дочерних элементах. Однако с целью упрощения примера
    // мы решили учитывать только один дочерний элемент.
    var textnode = element.firstChild;

    // Придать контейнерному элементу режим относительного позиционирования,
    // чтобы можно было вывести тени относительно его.
    // Порядок работы со свойствами стилей рассмотрен в этой главе далее.
    element.style.position = "relative";
```

```
    // Создать тени
    var numshadows = args.length/3;       // количество теней?
    for(var i = 0; i < numshadows; i++) { // цикл по каждой
        var shadowX = args[i*3];          // смещение по оси X
        var shadowY = args[i*3 + 1];      // смещение по оси Y
        var shadowColor = args[i*3 + 2];  // и цвет

        // Создать новый элемент <span> для размещения тени
        var shadow = document.createElement("span");

        // Использовать атрибут style для указания смещения и цвета
        shadow.setAttribute("style", "position:absolute; " +
                            "left:" + shadowX + "; " +
                            "top:" + shadowY + "; " +
                            "color:" + shadowColor + ";");

        // Добавить копию текстового узла с тенью в элемент span
        shadow.appendChild(textnode.cloneNode(false));

        // Затем добавить элемент span в контейнер
        element.appendChild(shadow);
    }

    // Теперь нужно поместить текст поверх тени. Сначала создается <span>
    var text = document.createElement("span");
    text.setAttribute("style", "position: relative"); // Позиционирование
    text.appendChild(textnode); // Переместить оригинальный текстовый узел
    element.appendChild(text);  // и добавить элемент span в контейнер
};

// Просматривает дерево документа, начиная от заданного корневого элемента,
// в поисках элементов с заданным именем тега. Если в найденном элементе
// установлен атрибут shadow, он передается методу Shadows.add() для создания
// эффекта тени. Если аргумент root опущен, используется объект document.
// Если имя тега опущено, поиск ведется во всех тегах.
Shadows.addAll = function(root, tagname) {
    if (!root) root = document; // Если корневой элемент не задан,
                                // произвести поиск по всему документу
    if (!tagname) tagname = '*'; // Любой тег, если имя тега не задано

    var elements = root.getElementsByTagName(tagname); // Искать все теги
    for(var i = 0; i < elements.length; i++) {         // Для каждого тега
        var shadow = elements[i].getAttribute("shadow"); // Если тень есть,
        if (shadow) Shadows.add(elements[i], shadow);    // создать тень
    }
};
```

16.2.3. Определение положения и размеров элемента

Теперь, когда известно, как с помощью CSS задавать положение и размеры HTML-элементов, возникает естественный вопрос: как выяснить положение и размеры элемента? Например, может появиться необходимость позиционировать средствами CSS всплывающее «DHTML-окно» по центру некоторого HTML-элемента, а для этого необходимо знать его положение и размеры.

В современных броузерах координаты X и Y элемента можно определить с помощью свойств `offsetLeft` **и** `offsetTop`**. Аналогичным образом ширину и высоту эле-**

мента можно определить с помощью свойств offsetWidth и offsetHeight. Эти свойства доступны только для чтения и возвращают числовые значения в пикселах (а не CSS-строку с суффиксом «px»). Они соответствуют CSS-атрибутам left, top, width и height, но не являются частью стандарта CSS. Впрочем, они не являются частью ни одного из стандартов: впервые они появились в Microsoft IE 4 и затем были реализованы остальными производителями броузеров.

К сожалению, нередко свойств offsetLeft и offsetTop бывает недостаточно. Эти свойства определяют координаты X и Y элемента относительно некоторого другого элемента, определяемого с помощью свойства offsetParent. Для позиционируемых элементов свойство offsetParent обычно ссылается на тег <body> или <html> (для них свойство offsetParent имеет значение null) или позиционируемый предок позиционируемого элемента. Для непозиционируемых элементов в разных броузерах свойство offsetParent может принимать разные значения. Например, в IE строки таблицы позиционируются относительно вмещающей таблицы. Таким образом, переносимый способ определения положения элемента заключается в том, чтобы обойти в цикле все ссылки offsetParent и сложить вместе все смещения по каждой из координат. Вот пример программного кода, который может использоваться для этих целей:

```
// Возвращает координату элемента e по оси X.
function getX(e) {
    var x = 0;              // Начальное значение 0
    while(e) {              // Начинать с элемента e
        x += e.offsetLeft;  // Добавить смещение
        e = e.offsetParent; // И перейти по ссылке offsetParent
    }
    return x;               // Вернуть сумму всех смещений
}
```

Функция getY() может быть реализована простой заменой свойства offsetLeft свойством offsetTop.

Примечательно, что в предыдущем примере такие функции, как getX(), возвращают координаты относительно начала документа. Они соответствуют CSS-координатам и на них не влияет положение полос прокрутки броузера. В главе 17 вы узнаете, что координаты, соответствующие событиям мыши, являются оконными, а для перехода к координатам документа к ним необходимо добавить позиции полос прокрутки.

Продемонстрированный здесь метод getX() имеет один недостаток. Далее вы увидите, что с помощью CSS-атрибута overflow внутри документа можно создавать прокручиваемые области. Когда элемент располагается внутри такой прокручиваемой области, значение смещения элемента не учитывает положение полос прокрутки области. Если в веб-странице используются такие прокручиваемые области, это может потребовать более сложного способа вычисления координат, например:

```
function getY(element) {
    var y = 0;
    for(var e = element; e; e = e.offsetParent) // Цикл по offsetParent
        y += e.offsetTop;                       // Сложить значения offsetTop

    // Теперь обойти все родительские элементы, отыскать среди них элементы,
```

```
        // где установлено свойство scrollTop, и вычесть эти значения из суммы
        // смещений. Однако цикл должен быть прерван по достижении элемента
        // document.body, в противном случае будет принята во внимание величина
        // прокрутки самого документа и в результате получены оконные координаты.
        for(e = element.parentNode; e && e != document.body; e = e.parentNode)
            if (e.scrollTop) y -= e.scrollTop; // вычесть величину прокрутки

        // Данная координата Y учитывает величину прокрутки внутренних областей документа.
        return y;
    }
```

16.2.4. Третье измерение: атрибут z-index

Мы видели, что с помощью атрибутов `left`, `top`, `right` и `bottom` можно задавать координаты X и Y элементов внутри двухмерной плоскости элемента контейнера. Атрибут `z-index` определяет что-то вроде третьего измерения – он позволяет задать порядок наложения элементов, указывая, какой из перекрывающихся элементов расположится поверх других. Атрибут `z-index` представляет собой целое число. По умолчанию его значение равно нулю, но можно задавать положительные и отрицательные значения. Когда два или более элементов перекрываются, они прорисовываются в порядке от наименьшего к наибольшему значению `z-index`, т. е. элемент с наибольшим значением `z-index` перекрывает все остальные. Если перекрывающиеся элементы имеют одинаковое значение `z-index`, они прорисовываются в том порядке, в котором присутствуют в документе, поэтому наверху оказывается последний из перекрывающихся элементов.

Обратите внимание: порядок наложения определяется значением `z-index` только для смежных элементов (т. е. для дочерних элементов одного контейнера). Если перекрываются два несмежных элемента, то на основе индивидуальных значений атрибутов `z-index` нельзя указать, какой из них находится сверху. Вместо этого надо задать атрибут `z-index` для двух смежных контейнеров двух перекрывающихся элементов.

Непозиционируемые элементы (т. е. элементы с используемым по умолчанию режимом позиционирования `position:static`) всегда размещаются способом, не допускающим перекрытий, поэтому к ним атрибут `z-index` не применяется. Тем не менее для них значение `z-index` по умолчанию равно нулю, т. е. позиционируемые элементы с положительным значением `z-index` перекрывают обычный поток вывода документа, а позиционируемые элементы с отрицательным значением `z-index` оказываются перекрытыми обычным потоком вывода документа.

И наконец, следует отметить, что некоторые броузеры не учитывают атрибут `z-index`, когда он применяется к тегам `<iframe>`, в результате встраиваемые фреймы располагаются поверх других элементов, независимо от указанного порядка наложения. Такие же неприятности могут быть и с другими «оконными» элементами, например с меню `<select>`. Старые броузеры могут отображать все элементы управления форм поверх абсолютно позиционируемых элементов независимо от значений `z-index`.

16.2.5. Отображение и видимость элемента

Для управления видимостью элемента документа служат два CSS-атрибута: `visibility` и `display`. Атрибут `visibility` прост: если его значение равно `hidden`, то

элемент не отображается, если visible, — отображается. Атрибут display более универсален и служит для задания варианта отображения элемента, определяя, блочный это элемент, встраиваемый, списочный или какой-нибудь другой. Если же атрибут display имеет значение none, то элемент вообще не отображается и даже не размещается.

Различие между атрибутами стиля visibility и display имеет отношение к их воздействию на элементы, не позиционируемые динамически. Для элемента, расположенного в нормальном потоке вывода документа (с атрибутом position, равным static или relative), установка атрибута visibility в значение hidden делает элемент невидимым, но резервирует для него место в документе. Такой элемент может повторно скрываться и отображаться без изменения компоновки документа. Однако если атрибут display элемента установлен в значение none, то место в документе для него не выделяется; элементы с обеих сторон от него смыкаются, как будто его вообще не существует. (По отношению к абсолютно или фиксированно позиционируемым элементам атрибуты visibility и display имеют одинаковое действие, т. к. эти элементы в любом случае никогда не являются частью общей компоновки документа.) Обычно атрибут visibility задается при работе с динамически позиционируемыми элементами, а атрибут display бывает полезен в разворачивающихся и сворачивающихся структурированных списках.

Обратите внимание: нет особого смысла использовать атрибуты visibility и display, чтобы сделать элемент невидимым, если вы не собираетесь динамически устанавливать их в JavaScript-коде, чтобы в какой-то момент сделать его снова видимым! Как это делается, я расскажу далее в этой главе.

16.2.6. Блочная модель и детали позиционирования в CSS

CSS-стили позволяют задавать поля, рамки и отступы для любого элемента, и это несколько усложняет позиционирование элементов средствами CSS, потому что для позиционирования требуется точно знать, как вычислять значения атрибутов width, height, top и left при наличии рамок и отступов. *Блочная модель* (*box model*) в CSS предлагает для этого точную спецификацию (рис. 16.3). Она подробно описывается далее.

Начнем наше обсуждение со стилей border, margin и padding. Рамка элемента – это прямоугольник, обрисованный вокруг (полностью или частично) этого элемента. CSS-атрибуты позволяют задать стиль, цвет и толщину рамки:

```
border: solid black 1px; /* рамка рисуется черной сплошной линей,
                         /* толщиной 1 пиксел */
border: 3px dotted red;  /* рамка рисуется красной пунктирной линией */
                         /* толщиной 3 пиксела */
```

Существует возможность определять толщину, стиль и цвет рамки с помощью отдельных CSS-атрибутов, а также отдельно для каждой из сторон рамки элемента. Например, чтобы нарисовать линию под элементом, достаточно просто установить значение атрибута border-bottom. Можно даже определить толщину, стиль или цвет только одной стороны элемента. Это иллюстрирует рис. 16.3, на котором показаны такие атрибуты рамки, как border-top-width и border-left-width.

Атрибуты margin и padding задают ширину пустого пространства вокруг элемента. Отличие (очень важное) между этими атрибутами заключается в том, что ат-

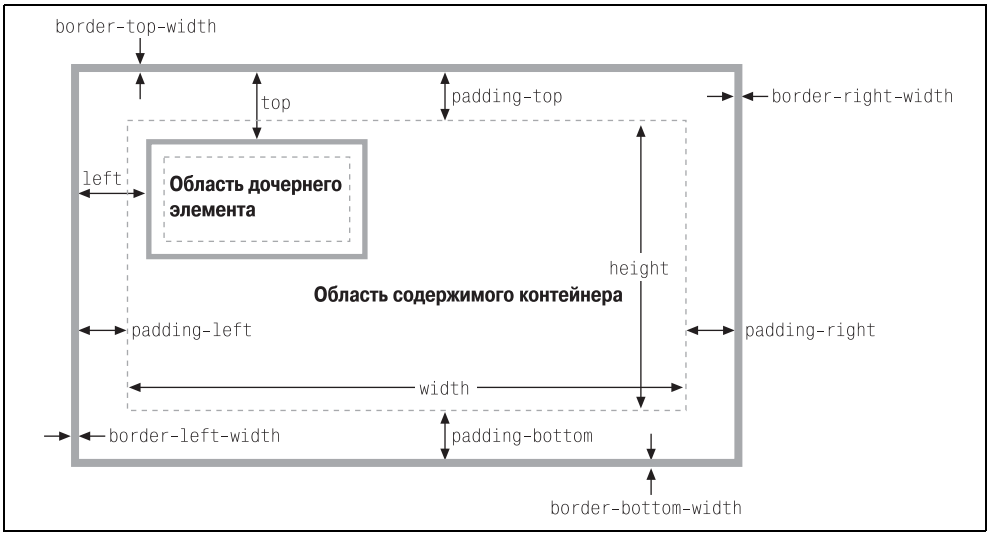

***Рис. 16.3.** Блочная модель в CSS (рамки, отступы и атрибуты позиционирования)*

рибут `margin` задает ширину пустого пространства снаружи рамки, между рамками соседних элементов, а `padding` – внутри рамки, между рамкой и содержимым элемента. Атрибут `margin` предназначен для создания визуального пространства между элементами (возможно окруженными рамками) в нормальном потоке вывода документа. Атрибут `padding` предназначен для визуального отделения содержимого элемента от его рамки. Если элемент не имеет рамки, устанавливать атрибут `padding` обычно не требуется. Если элемент позиционируется динамически, значит, он не является частью нормального потока вывода документа и потому не имеет смысла задавать значение атрибута `margin`. (Именно по этой причине на рис. 16.3 атрибуты `margin` не показаны.)

Поля и отступы элемента задаются с помощью атрибутов `margin` и `padding` соответственно:

```
margin: 5px; padding: 5px;
```

Помимо этого можно определить поля и отступы отдельно для каждой из сторон элемента:

```
margin-left: 25px;
padding-bottom: 5px;
```

С помощью атрибутов `margin` и `padding` можно также задать значения полей и отступов для всех четырех сторон элемента. В этом случае значение атрибута сначала задается для верхней стороны и далее по часовой стрелке, т. е. порядок таков: верх, правая сторона, низ, левая сторона. Например, следующий фрагмент демонстрирует два эквивалентных способа установки различных значений отдельно для каждой из сторон элемента:

```
padding: 1px 2px 3px 4px;
/* Предыдущая строка эквивалентна четырем следующим. */
padding-top: 1px;
padding-right: 2px;
```

```
padding-bottom: 3px;
padding-left: 4px;
```

Порядок работы с атрибутом `margin` ничем не отличается.

Теперь, получив представление о полях, рамках и отступах, можно переходить к детальному изучению атрибутов позиционирования в CSS. Во-первых, атрибуты `width` и `height` определяют лишь размеры содержимого элемента – они не учитывают дополнительное пространство, необходимое для размещения полей, рамок и отступов. Чтобы определить полную ширину элемента на экране вместе с рамками, необходимо сложить отступы с левой и правой сторон, толщину рамок с левой и правой сторон, а затем добавить к полученному значению ширину элемента. Аналогично, чтобы получить полную высоту, необходимо сложить отступы сверху и снизу, толщину рамок сверху и снизу, а затем добавить высоту элемента.

Поскольку атрибуты `width` и `height` определяют ширину и высоту только области содержимого элемента, может появиться мысль, что атрибуты `left` и `top` (а так же `right` и `bottom`) должны измеряться относительно области содержимого объемлющего элемента. Однако это не так. Стандарт CSS утверждает, что эти значения измеряются относительно внешнего края отступов объемлющего элемента (т. е. относительно внутреннего края рамки объемлющего элемента).

Все это иллюстрирует рис. 16.3, но чтобы не оставлять сомнений, рассмотрим несложный пример. Предположим, что у вас есть динамически позиционируемый элемент, вокруг содержимого которого имеются отступы размером 10 пикселов, а вокруг них – рамка толщиной 5 пикселов. Теперь предположим, что вы динамически позиционируете дочерний элемент внутри этого контейнера. Если установить атрибут `left` дочернего элемента равным `0px`, обнаружится, что левый край дочернего элемента будет находиться непосредственно у внутреннего края рамки контейнера. При этом значении атрибута дочерний элемент перекрывает отступы контейнера, хотя предполагается, что они остаются пустыми (т. к. для этого и предназначены отступы). Чтобы поместить дочерний элемент в левый верхний угол области содержимого контейнера, необходимо установить атрибуты `left` и `top` равными `10px`.

16.2.6.1. Особенности Internet Explorer

Теперь вы знаете, что `width` и `height` задают размер области содержимого элемента, а атрибуты `left`, `top`, `right` и `bottom` измеряются относительно отступов элемента-контейнера, а значит, вам пора узнать еще одну деталь: Internet Explorer версий от 4 до 5.5 для Windows (но не IE 5 для Mac) реализует атрибуты `width` и `height` некорректно и включает в их значения рамку и отступы (но не поля). Так, если установить ширину элемента равной 100 пикселов и поместить слева и справа отступы шириной 10 пикселов и рамку толщиной 5 пикселов, то ширина области содержимого элемента в этих версиях Internet Explorer будет равна лишь 70 пикселов.

В IE 6 CSS-атрибуты положения и размера работают корректно, когда броузер находится в стандартном режиме, и некорректно (но совместимо с предыдущими версиями), когда броузер находится в режиме совместимости. Стандартный режим (и соответственно корректная реализация блочной модели CSS) включается при наличии тега `<!DOCTYPE>` в начале документа. Этот тег объявляет, что до-

кумент соответствует стандарту HTML 4.0 (или более поздней версии) или некоторой версии стандартов XHTML. Например, любое из следующих объявлений типа HTML-документа приводит к отображению документов в IE 6 в стандартном режиме:

```
<!DOCTYPE HTML PUBLIC "//W3C//DTD HTML 4.0//EN">
<!DOCTYPE HTML PUBLIC "//W3C//DTD HTML 4.0 Strict//EN">
<!DOCTYPE HTML PUBLIC "//W3C//DTD HTML 4.0 Transitional//EN"
    "http://www.w3.org/TR/html4/loose.dtd">
```

Такое различие между стандартным режимом и режимом совместимости (иногда называемым «режимом с причудами») не уникально для Internet Explorer. Другие броузеры также реагируют на объявление `<!DOCTYPE>`, переходя в режим точного соблюдения стандартов, а при отсутствии этого объявления возвращаются к поведению по умолчанию, обеспечивающему обратную совместимость. Но как бы там ни было, только IE обладает такой вопиющей проблемой совместимости.

16.2.7. Цвет, прозрачность и полупрозрачность

Ранее при обсуждении границ мы рассматривали пример, в котором задавался цвет рамки указанием имен наиболее распространенных цветов, таких как «red» (красный) и «black» (черный). Более универсальный способ определения цветов в CSS заключается в использовании шестнадцатеричных цифр, определяющих красную, зеленую и синюю составляющие цвета. Значения каждой из составляющих могут задаваться одной или двумя цифрами. Например:

```
#000000  /* черный */
#fff     /* белый */
#f00     /* ярко-красный */
#404080  /* ненасыщенный темно-синий */
#ccc     /* светло-серый */
```

Помимо возможности задания цвета рамки с помощью такой нотации, существует также возможность задания цвета текста с помощью CSS-атрибута `color`. Кроме того, для любого элемента можно определить цвет фона с помощью атрибута `background-color`. Таблицы CSS-стилей позволяют точно указать позицию, размеры, цвета фона и рамки элемента, что обеспечивает элементарные графические средства рисования прямоугольников и (если до предела уменьшить высоту или ширину) горизонтальных или вертикальных линий. К этой теме мы еще вернемся в главе 22, когда будут обсуждаться возможности рисования столбиковых диаграмм с использованием модели DOM API и позиционирования средствами CSS.

В дополнение к атрибуту `background-color` в качестве фоновой картинки элемента можно использовать графические изображения. Атрибут `background-image` определяет фоновое изображение, а атрибуты `background-attachment`, `background-position` и `background-repeat` уточняют некоторые параметры рисования изображения. Сокращенный вариант – атрибут `background`, позволяющий указывать все эти атрибуты вместе. Атрибуты фонового рисунка могут применяться для создания довольно интересных визуальных эффектов, но это уже выходит за рамки темы данной книги.

Очень важно понять, что если цвет фона или фоновый рисунок элемента не задан, то фон элемента обычно бывает прозрачным. Например, если поверх некоторого

текста в обычном потоке вывода документа расположить элемент с абсолютным позиционированием, то по умолчанию текст будет виден через элемент . Если же элемент содержит собственный текст, символы окажутся наложенными друг на друга, образуя трудную для чтения мешанину. Однако не все элементы по умолчанию прозрачны. Например, у элементов форм нет прозрачного фона, а такие элементы, как <button>, имеют цвет фона по умолчанию. Переопределить значение цвета по умолчанию можно с помощью атрибута background-color, можно также явно сделать цвет фона прозрачным, если в этом появится необходимость.

Прозрачность, о которой мы до сих пор говорили, может быть либо полной, либо нулевой: элемент имеет либо прозрачный, либо непрозрачный фон. Однако существует возможность получить полупрозрачный элемент (для содержимого как заднего, так и переднего плана); пример полупрозрачного элемента приведен на рис. 16.4. Делается это с помощью атрибута opacity стандарта CSS3. Значением этого атрибута является число в диапазоне от 0 до 1, где 1 означает 100-процентную непрозрачность (значение по умолчанию), а 0 – 100-процентную прозрачность. Атрибут opacity поддерживается броузером Firefox. Ранние версии Mozilla поддерживали экспериментальный вариант этого атрибута с именем moz-opacity. В IE аналогичная функциональность реализуется с помощью специфичного атрибута filter. Чтобы сделать элемент непрозрачным на 75%, можно воспользоваться следующими CSS-стилями:

```
opacity: .75;              /* стандартный стиль прозрачности в CSS3 */
-moz-opacity: .75;         /* прозрачность в ранних версиях Mozilla */
filter: alpha(opacity=75); /* прозрачность в IE; обратите внимание */
                           /* на отсутствие десятичной точки */
```

16.2.8. Частичная видимость: атрибуты overflow и clip

Атрибут visibility позволяет полностью скрыть элемент документа. С помощью атрибутов overflow и clip можно отобразить только часть элемента. Атрибут overflow определяет, что происходит, когда содержимое документа превышает размер, указанный для элемента (например, в атрибутах стиля width и height). Далее перечислены допустимые значения этого атрибута и указано их назначение:

visible

Содержимое может выходить за пределы и по необходимости прорисовываться вне прямоугольника элемента. Это значение по умолчанию.

hidden

Содержимое, вышедшее за пределы элемента, обрезается и скрывается, так что никакая часть содержимого никогда не прорисовывается вне области, определяемой атрибутами размера и позиционирования.

scroll

Область элемента имеет постоянные горизонтальную и вертикальную полосы прокрутки. Если содержимое превышает размеры области, полосы прокрутки позволяют увидеть остальное содержимое. Это значение учитывается, только когда документ отображается на экране компьютера; когда документ выводится, например, на бумагу, полосы прокрутки, очевидно, не имеют смысла.

`auto`

Полосы прокрутки отображаются не постоянно, а только когда содержимое превышает размер элемента.

В то время как свойство `overflow` определяет, что должно происходить, если содержимое элемента превысит область элемента, то с помощью свойства `clip` можно точно указать, какая часть элемента должна отображаться независимо от того, выходит ли содержимое за пределы элемента. Этот атрибут особенно полезен для создания DHTML-эффектов, когда элемент открывается или проявляется постепенно.

Значение свойства `clip` задает область отсечения элемента. В CSS2 области отсечения прямоугольные, но синтаксис атрибута `clip` обеспечивает возможность в следующих версиях стандарта задавать области отсечения, отличные от прямоугольных. Синтаксис атрибута `clip`:

```
rect(top right bottom left)
```

Значения *top*, *right*, *bottom* и *left* задают границы прямоугольника отсечения относительно левого верхнего угла области элемента. Чтобы, например, вывести только часть элемента в области 100Ч100 пикселов, можно задать для этого элемента следующий атрибут `style`:

```
style="clip: rect(0px 100px 100px 0px);"
```

Обратите внимание, что четыре значения в скобках представляют собой значения длины и должны включать спецификацию единиц измерения, например `px` для пикселов. Проценты здесь не допускаются. Значения могут быть отрицательными – это будет означать, что область отсечения выходит за переделы области, определенной для элемента. Для любого из четырех значений ключевое слово `auto` указывает, что этот край области отсечения совпадает с соответствующим краем самого элемента. Например, можно вывести только левые 100 пикселов элемента с помощью следующего атрибута `style`:

```
style="clip: rect(auto 100px auto auto);"
```

Обратите внимание, что между значениями нет запятых, и края области отсечения задаются по часовой стрелке, начиная с верхнего края.

16.2.9. Пример: перекрытие полупрозрачных окон

Данный раздел завершается примером, который демонстрирует порядок работы с большинством обсуждавшихся CSS-атрибутов. В примере 16.3 CSS-стили используются для создания визуального эффекта наложения полупрозрачного окна на другое окно, обладающее полосой прокрутки. Результат приводится на рис. 16.4. Пример не содержит JavaScript-код и в нем нет никаких обработчиков событий, поэтому возможность взаимодействия с окнами отсутствует (иначе как через полосу прокрутки), но это очень интересная демонстрация эффектов, которые можно получить средствами CSS.

Пример 16.3. Отображение окон с использованием CSS-стилей

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN">
<head>
<style type="text/css">
```

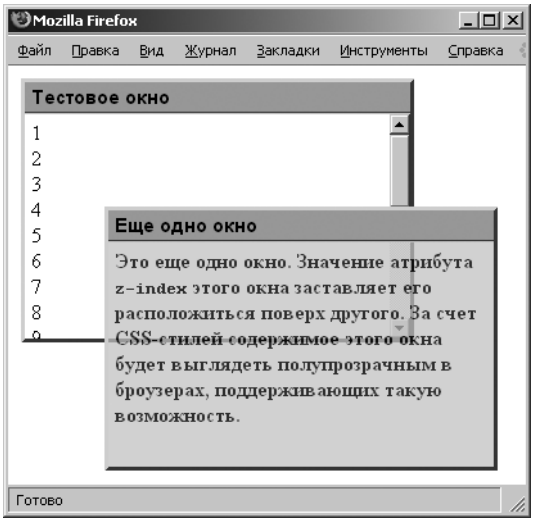

***Рис. 16.4.** Окна, созданные с помощью CSS*

```
/**
 * Эта таблица CSS-стилей определяет три правила стилей, которые служат в теле документа
 * для создания визуального эффекта "окна". В правилах использованы атрибуты
 * позиционирования для установки общего размера окна и расположения его компонентов.
 * Изменение размеров окна требует аккуратного изменения атрибутов позиционирования
 * во всех трех правилах.
 **/
div.window {                    /* Определяет размер и границу окна */
    position: absolute;         /* Положение задается в другом месте */
    width: 300px; height: 200px; /* Размер окна без учета границ */
    border: 3px outset gray;    /* Обратите внимание на 3D-эффект рамки */
}

div.titlebar {                /* Задает положение, размер и стиль заголовка */
    position: absolute;         /* Это - позиционируемый элемент */
    top: 0px; height: 18px;     /* Высота заголовка 18px + отступ и рамка */
    width: 290px;               /* 290 + 5px отступы слева и справа = 300 */
    background-color: #aaa;     /* Цвет заголовка */
    border-bottom: groove gray 2px; /* Заголовок имеет границу только снизу */
    padding: 3px 5px 2px 5px;   /* Значения по часовой стрелке:
                                /* сверху, справа, снизу, слева */
    font: bold 11pt sans-serif; /* Шрифт заголовка */
}

div.content { /* Задает размер, положение и прокрутку содержимого окна */
    position: absolute;       /* Это - позиционируемый элемент */
    top: 25px;                /* 18px заголовок+2px рамка+3px+2px отступ */
    height: 165px;            /* 200px всего - 25px заголовок - 10px отступ */
    width: 290px;             /* 300px ширина - 10px отступ */
    padding: 5px;             /* Отступы со всех четырех сторон */
    overflow: auto;           /* Разрешить появление полос прокрутки */
                              /* в случае необходимости */
    background-color: #ffffff;/* По умолчанию фон белый */
```

```
}
div.translucent {             /* Этот класс делает окно частично прозрачным */
    opacity: .75;             /* Стандартный стиль прозрачности */
    -moz-opacity: .75;        /* Прозрачность для ранних версий Mozilla */
    filter: alpha(opacity=75);/* Прозрачность для IE */
}
</style>
</head>

<body>
<!-- Порядок определения окна: элемент div "окна" с заголовком и элемент div  -->
<!-- с содержимым, вложенный между ними. Обратите внимание, как задается   -->
<!-- позиционирование с помощью атрибута style, дополняющего стили из таблицы стилей -->
<div class="window" style="left: 10px; top: 10px; z-index: 10;">
<div class="titlebar">Тестовое окно</div>
<div class="content">
1<br>2<br>3<br>4<br>5<br>6<br>7<br>8<br>9<br>0<br><!-- Множество строк для -->
1<br>2<br>3<br>4<br>5<br>6<br>7<br>8<br>9<br>0<br><!-- демонстрации прокрутки -->
</div>
</div>

<!-- Это еще одно окно с другими позицией, цветом и шрифтом -->
<div class="window" style="left: 75px; top: 110px; z-index: 20;">
<div class="titlebar">Еще одно окно</div>
<div class="content translucent"
style="background-color:#d0d0d0; font-weight:bold;">
Это еще одно окно. Значение атрибута <tt>z-index</tt> этого окна заставляет его
расположиться поверх другого. За счет CSS-стилей содержимое этого окна
будет выглядеть полупрозрачным в броузерах, поддерживающих такую возможность.
</div>
</div>
</body>
```

Основной недостаток этого примера в том, что таблица стилей задает фиксированный размер всех окон. Так как заголовок и содержимое окна должны точно позиционироваться внутри окна, изменение размера окна требует изменения значений различных атрибутов позиционирования во всех трех правилах, определенных в таблице стилей. Это трудно сделать в статическом HTML-документе, но это будет не так трудно, если мы сможем установить все необходимые атрибуты при помощи сценария. Эта возможность рассматривается в следующем разделе.

16.3. Использование стилей в сценариях

Главное в DHTML – возможность динамически изменять атрибуты стиля, применяемые к отдельным элементам документа, при помощи JavaScript. Стандарт DOM Level 2 определяет прикладной интерфейс (API), позволяющий довольно легко это делать. В главе 15 рассматривалось применение модели DOM API для получения ссылок на элементы документа либо по имени тега или идентификатору, либо рекурсивно с обходом всего документа. Получив ссылку на элемент, стилями которого вы хотите манипулировать, вы устанавливаете свойство `style` элемента, чтобы получить объект `CSS2Properties` для данного элемента документа. Этот JavaScript-объект имеет свойства, соответствующие всем атрибутам

стилей CSS1 и CSS2. Установка этих свойств имеет тот же эффект, что и установка соответствующих стилей в атрибуте `style` данного элемента. Чтение этих свойств возвращает значение CSS-атрибута, которое, возможно, было установлено в атрибуте `style` элемента. Описание объекта `CSS2Properties` вы найдете в четвертой части книги.

Важно понимать, что полученный вами объект `CSS2Properties` со свойством `style` элемента определяет только встроенные стили элемента. Невозможно использовать свойства объекта `CSS2Properties` для получения информации о примененных к элементу стилях из таблицы стилей. Устанавливая свойства для этого объекта, вы задаете встроенные стили, переопределяющие стили из таблицы стилей.

Рассмотрим следующий сценарий. Он находит все элементы `<img>` в документе и выполняет их перебор в цикле для поиска тех объектов, которые оказываются (судя по их размеру) рекламными баннерами. Найдя баннер, сценарий при помощи свойства `style.visibility` устанавливает значение CSS-атрибута `visibility` равным `hidden`, что делает баннер невидимым:

```
var imgs = document.getElementsByTagName("img"); // Находим все изображения
for(var i = 0; i < imgs.length; i++) {           // Цикл по ним
    var img=imgs[i];
    if (img.width == 468 && img.height == 60)    // Если это баннер 468x60...
        img.style.visibility = "hidden";         // прячем его!
}
```

Этот простой сценарий можно трансформировать в букмарклет, преобразовав его в URL-адрес со спецификатором `javascript:` и добавив в закладки броузера (см. раздел 13.4.1).

16.3.1. Соглашения об именах: CSS-атрибуты в JavaScript

В именах многих атрибутов CSS-стилей, таких как `font-family`, содержатся дефисы. В JavaScript дефис интерпретируется как знак минус, поэтому нельзя написать, например, такое выражение:

```
element.style.font-family = "sans-serif";
```

Таким образом, имена свойств объекта `CSS2Properties` слегка отличаются от имен реальных CSS-атрибутов. Если имя CSS-атрибута содержит дефисы, имя свойства объекта `CSS2Properties` образуется путем удаления дефисов и перевода в верхний регистр буквы, непосредственно следующей за каждым из них. Другими словами, атрибут `border-left-width` доступен через свойство `borderLeftWidth`, а к атрибуту `font-family` можно обратиться следующим образом:

```
element.style.fontFamily = "sans-serif";
```

Есть еще одно отличие между именами CSS-атрибутов и свойств объекта `CSS2Properties` в JavaScript. Слово «float» является ключевым в Java и других языках, и хотя сейчас это слово не употребляется в JavaScript, оно зарезервировано на будущее. Поэтому в объекте `CSS2Properties` не может быть свойства с именем `float`, соответствующего CSS-атрибуту `float`. Затруднение преодолевается путем добавления префикса «css» к атрибуту `float`, в результате чего образуется имя свойства `cssFloat`. Следовательно, значение атрибута `float` элемента можно установить или получить при помощи свойства `cssFloat`, объекта `CSS2Properties`.

16.3.2. Работа со свойствами стилей

При работе со свойствами стилей объекта `CSS2Properties` помните, что все значения должны быть указаны в виде строк. В таблице стилей или атрибуте `style` можно написать:

```
position: absolute; font-family: sans-serif; background-color: #ffffff;
```

Чтобы сделать то же самое для элемента `e` в JavaScript, необходимо поместить все эти значения в кавычки:

```
e.style.position = "absolute";
e.style.fontFamily = "sans-serif";
e.style.backgroundColor = "#ffffff";
```

Обратите внимание, что точки с запятыми остаются вне строк. Это обычные точки с запятой, употребляемые в синтаксисе языка JavaScript. Точки с запятой, используемые в таблицах CSS-стилей, не нужны в строковых значениях, устанавливаемых с помощью JavaScript.

Кроме того, помните, что во всех свойствах позиционирования должны быть указаны единицы измерения. Следовательно, нельзя устанавливать свойство `left` подобным образом:

```
e.style.left = 300;   // Неправильно: это число, а не строка
e.style.left = "300"; // Неправильно: отсутствуют единицы измерения
```

Единицы измерения обязательны при установке свойств стиля в JavaScript – так же, как при установке атрибутов стиля в таблицах стилей. Далее приводится правильный способ установки значения свойства `left` элемента `e` равным 300 пикселов:

```
e.style.left = "300px";
```

Чтобы установить свойство `left` равным вычисляемому значению, обязательно добавьте единицы измерения в конце вычислений:

```
e.style.left = (x0 + left_margin + left_border + left_padding) + "px";
```

Как побочный эффект добавления единиц измерения, добавление строки преобразует вычисленное значение из числа в строку.

Объект `CSS2Properties` может также использоваться для получения значений CSS-атрибутов, явно установленных в атрибуте `style` элемента, или для чтения любых встроенных значений стилей, ранее установленных JavaScript-кодом. Однако и здесь необходимо помнить, что значения, полученные из этих свойств, представляют собой строки, а не числа, поэтому следующий код (предполагающий, что для элемента `e` с помощью встраиваемых стилей установлены поля) не делает того, что от него, возможно, ожидалось:

```
var totalMarginWidth = e.style.marginLeft + e.style.marginRight;
```

А вот такой код будет работать правильно:

```
var totalMarginWidth = parseInt(e.style.marginLeft) +
                       parseInt(e.style.marginRight);
```

Это выражение просто отбрасывает спецификации единиц измерения, возвращаемые в конце обеих строк. В нем предполагается, что свойства `marginLeft` и `mar-`

ginRight заданы с одинаковыми единицами измерения. Если во встроенных стилях в качестве единиц измерения указаны исключительно пикселы, то, как правило, можно обойтись простым отбрасыванием единиц измерения, как в данном примере.

Вспомните, что некоторые CSS-атрибуты, например margin, представляют собой сокращения от других свойств, например margin-top, margin-right, margin-bottom и margin-left. Объект CSS2Properties имеет свойства, соответствующие этим сокращенным атрибутам. Так, можно установить свойство margin следующим образом:

```
e.style.margin = topMargin + "px " + rightMargin + "px " +
                 bottomMargin + "px " + leftMargin + "px";
```

Хотя возможно, кому-то будет проще установить четыре свойства полей по отдельности:

```
e.style.marginTop = topMargin + "px";
e.style.marginRight = rightMargin + "px";
e.style.marginBottom = bottomMargin + "px";
e.style.marginLeft = leftMargin + "px";
```

Можно также получить значения сокращенных свойств, но это редко имеет смысл, поскольку обычно в этом случае приходится разбивать полученное значение на отдельные компоненты. Как правило, это сделать сложно, в то время как получить свойства-компоненты по отдельности намного проще.

И наконец, позвольте мне снова подчеркнуть, что когда вы получаете объект CSS2Properties через свойство style объекта HTMLElement, свойства этого объекта представляют значения встроенных атрибутов стилей элемента. Другими словами, установка одного из этих свойств эквивалентна установке CSS-атрибута в атрибуте style элемента – она влияет только на данный элемент и имеет приоритет над конфликтующими установками стилей из других источников в CSS-каскаде. Именно такое точное управление отдельными элементами и требуется при создании DHTML-эффектов с помощью JavaScript.

Однако когда мы читаем значения этих свойств в CSS2Properties, они возвращают осмысленные значения, только если были ранее установлены нашим JavaScript-кодом или если HTML-элемент, с которым мы работаем, имеет встроенный атрибут style, установивший нужное свойство. Например, документ может включать таблицу стилей, устанавливающую левое поле для всех абзацев равным 30 пикселов, но если прочитать свойство leftMargin одного из абзацных элементов, будет получена пустая строка, если только этот абзац не имеет атрибута style, переопределяющего значение, установленное таблицей стилей.

Поэтому несмотря на то, что объект CSS2Properties может использоваться для установки стилей, переопределяющих другие стили, он не позволяет запросить CSS-каскад и определить полный набор стилей, применяемых к данному элементу. В разделе 16.4 мы кратко рассмотрим метод getComputedStyle() и его альтернативу в IE – свойство currentStyle, предоставляющие именно такую возможность.

16.3.3. Пример: всплывающие подсказки в CSS

Пример 16.4 представляет собой JavaScript-модуль, предназначенный для отображения простых всплывающих DHTML-подсказок, как показано на рис. 16.5.

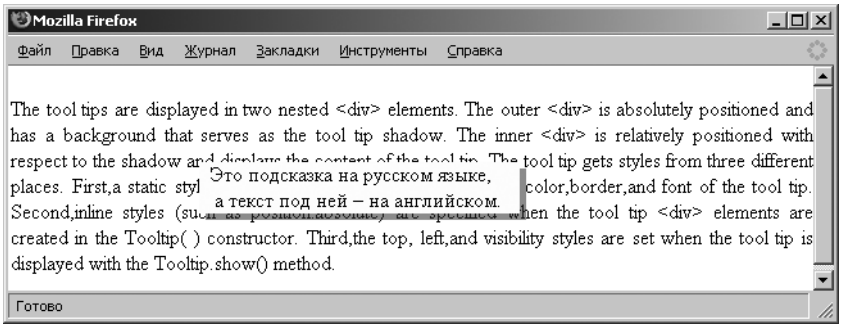

***Рис. 16.5.** Всплывающая подсказка, полученная средствами CSS*

Всплывающие подсказки выводятся в двух вложенных элементах `<div>`. Внешний элемент `<div>` позиционируется по абсолютным координатам и имеет фон, который позволяет ему выступать в роли тени. Внутренний элемент `<div>` позиционируется относительно внешнего элемента `<div>` с учетом необходимости создания эффекта тени и отображает содержимое подсказки. Стили подсказки задаются в трех разных местах. Во-первых, в статических таблицах стилей, определяющих тень, цвет фона, рамку и шрифт текста подсказки. Во-вторых, во встроенных стилях (таких как `position:absolute`), определяемых при создании элементов `<div>` в конструкторе `Tooltip()`. В-третьих, в стилях `top`, `left` и `visibility`, которые устанавливаются при выводе подсказки методом `Tooltip.show()`.

Обратите внимание: пример 16.4 – это простейшая реализация модуля, который просто отображает и скрывает подсказки. Позднее мы расширим данный пример, добавив в него механизм вывода подсказок в ответ на события от мыши (см. пример 17.3).

Пример 16.4. Реализация всплывающих подсказок средствами CSS

```
/**
 * Tooltip.js: простейшие всплывающие подсказки, отбрасывающие тень.
 *
 * Этот модуль определяет класс Tooltip. Объекты класса Tooltip создаются
 * с помощью конструктора Tooltip(). После этого подсказку можно сделать
 * видимой вызовом метода show(). Чтобы скрыть подсказку, следует
 * вызвать метод hide().
 *
 * Обратите внимание: для корректного отображения подсказок с использованием
 * этого модуля необходимо добавить соответствующие определения CSS-классов
 * Например:
 *
 * .tooltipShadow {
 *     background: url(shadow.png); /* полупрозрачная тень * /
 * }
 *
 * .tooltipContent {
 *     left: -4px; top: -4px;       /* смещение относительно тени * /
 *     background-color: #ff0;      /* желтый фон * /
 *     border: solid black 1px;     /* тонкая рамка черного цвета * /
```

```
 *     padding: 5px;               /* отступы между текстом и рамкой * /
 *     font: bold 10pt sans-serif; /* небольшой жирный шрифт * /
 * }
 *
 * В броузерах, поддерживающих возможность отображения полупрозрачных
 * изображений формата PNG, можно организовать отображение полупрозрачной
 * тени. В остальных броузерах придется использовать для тени сплошной цвет
 * или эмулировать полупрозрачность с помощью изображения формата GIF.
 */
function Tooltip( ) { // Функция-конструктор класса Tooltip
    this.tooltip = document.createElement("div"); // Создать div для тени
    this.tooltip.style.position = "absolute";     // Абсолютное позиционирование
    this.tooltip.style.visibility = "hidden";     // Изначально подсказка скрыта
    this.tooltip.className = "tooltipShadow";     // Определить его стиль

    this.content = document.createElement("div"); // Создать div с содержимым
    this.content.style.position = "relative";     // Относительное позиционирование
    this.content.className = "tooltipContent";    // Определить его стиль

    this.tooltip.appendChild(this.content);       // Добавить содержимое к тени
}

// Определить содержимое, установить позицию окна с подсказкой и отобразить ее
Tooltip.prototype.show = function(text, x, y) {
    this.content.innerHTML = text;              // Записать текст подсказки.
    this.tooltip.style.left = x + "px";         // Определить положение.
    this.tooltip.style.top = y + "px";
    this.tooltip.style.visibility = "visible"; // Сделать видимой.

    // Добавить подсказку в документ, если это еще не сделано
    if (this.tooltip.parentNode != document.body)
        document.body.appendChild(this.tooltip);
};

// Скрыть подсказку
Tooltip.prototype.hide = function() {
    this.tooltip.style.visibility = "hidden"; // Сделать невидимой.
};
```

16.3.4. DHTML–анимация

Одной из наиболее мощных DHTML-технологий, которую можно реализовать с помощью JavaScript и CSS, является анимация. В DHTML-анимации нет ничего особенного – надо лишь периодически изменять одно или несколько свойств стиля одного или нескольких элементов. Чтобы, например, передвинуть изображение влево, надо постепенно увеличивать значение свойства `style.left` этого изображения, пока последнее не займет требуемое положение. Можно также постепенно изменять свойство `style.clip` для «открытия» изображения пиксел за пикселом.

Пример 16.5 содержит простой HTML-файл, определяющий анимируемый элемент `div`, и короткий сценарий, изменяющий цвет фона элемента каждые 500 миллисекунд. Примечательно, что изменение цвета выполняется присваиванием значения свойства CSS-стиля. Анимация возникает за счет того, что изменение цвета выполняется периодически с помощью функции `setInterval()` объекта `Window`.

(Никакая DHTML-анимация не обходится без метода `setInterval()` или `setTimeout()`; возможно, перед изучением примера вы захотите почитать об этих методах объекта `Window` в четвертой части книги.) И наконец, обратите внимание на использование оператора деления по модулю (получение остатка) `%` для перебора цветов. Тот, кто забыл, как работает этот оператор, может обратиться к главе 5.

Пример 16.5. Простая анимация изменения цвета

```
<!-- Это анимируемый элемент div -->
<div id="urgent"><h1>Внимание!</h1>Веб-сервер атакован!</div>

<script>
var e = document.getElementById("urgent");        // Получаем объект Element
e.style.border = "solid black 5px";               // Рамка
e.style.padding = "50px";                         // И отступ
var colors = ["white", "yellow", "orange", "red"] // Перебираемые цвета
var nextColor = 0;                                // Текущая позиция перебора
// Вызывать следующую функцию с интервалом 500 миллисекунд
// для изменения цвета рамки.
setInterval(function() {
                e.style.borderColor=colors[nextColor++%colors.length];
            }, 500);
</script>
```

Пример 16.5 реализует очень простую анимацию. На практике анимация с использованием CSS-стилей обычно подразумевает одновременную модификацию нескольких свойств стилей (таких как `top`, `left` и `clip`). Сложную анимацию с помощью технологии, показанной в примере 16.5, создать трудно. Кроме того, чтобы не надоедать пользователю, анимация должна выполняться в течение короткого периода времени и затем останавливаться, чего нельзя сказать об анимации из примера 16.5.

В примере 16.6 показан JavaScript-файл, определяющий функцию анимации на базе CSS, намного облегчающей эту задачу даже при создании сложной анимации.

Пример 16.6. Основа для создания анимации на базе CSS

```
/**
 * AnimateCSS.js:
 * Этот файл определяет функцию с именем animateCSS(), служащую основой
 * для создания анимации на базе CSS. Аргументы функции:
 *
 *     element: Анимируемый HTML-элемент.
 *     numFrames: Общее число кадров в анимации.
 *     timePerFrame: Количество миллисекунд отображения каждого кадра.
 *     animation: Объект, определяющий анимацию; описывается далее.
 *     whendone: Необязательная функция, вызываемая по завершении анимации.
 *               Если функция указана, ей в качестве аргумента передается
 *               значение аргумента element.
 *
 * Функция animateCSS() просто определяет платформу для анимации. Выполняемую
 * анимацию определяют свойства объекта animation. Каждое свойство должно
 * иметь то же имя, что и свойство CSS-стиля. Значением каждого свойства
 * должна быть функция, возвращающая значения для этого свойства стиля.
 * Каждой функции передается номер кадра и общий промежуток времени, прошедший
```

```
 * с начала анимации, а функция может использовать это для вычисления
 * значения стиля, которое она должна вернуть для данного фрейма.
 * Например, чтобы анимировать изображение так, чтобы оно передвигалось
 * из левого верхнего угла, вы можете вызвать функцию animateCSS так:
 *
 * animateCSS(image, 25, 50, // Анимировать изображение в течение 25 кадров
 *                           // по 50 мс каждый
 *            { // Установить атрибуты top и left для каждого кадра:
 *                top: function(frame,time) { return frame*8 + "px"; },
 *                left: function(frame,time) { return frame*8 + "px"; }
 *            });
 *
 **/
function animateCSS(element, numFrames, timePerFrame, animation, whendone) {
    var frame = 0; // Текущий номер кадра
    var time = 0;  // Общее время анимации, прошедшее с начала

    // Определить вызов displayNextFrame() каждые timePerFrame миллисекунд.
    // Так будет отображаться каждый кадр анимации.
    var intervalId = setInterval(displayNextFrame, timePerFrame);

    // На этом работа animateCSS() завершается, но предыдущая строка гарантирует,
    // что следующая вложенная функция будет вызываться для каждого кадра анимации.
    function displayNextFrame() {
        if (frame >= numFrames) {    // Проверить, не закончилась ли анимация
            clearInterval(intervalId); // Если да - прекратить вызовы
            if (whendone) whendone(element); // Вызвать функцию whendone
            return;                          // И завершить работу
        }

        // Обойти в цикле все свойства, определяемые объектом анимации
        for(var cssprop in animation) {
            // Для каждого свойства вызвать его функцию анимации, передавая
            // ей номер кадра и прошедшее время. Используем возвращаемое
            // функцией значение в качестве нового значения соответствующего
            // свойства стиля для указанного элемента. Используем блок
            // try/catch, чтобы игнорировать любые исключительные ситуации,
            // возникающие из-за неверных возвращаемых значений.
            try {
                element.style[cssprop] = animation[cssprop](frame, time);
            } catch(e) {}
        }

        frame++;              // Увеличить номер кадра
        time += timePerFrame; // Увеличить прошедшее время
    }
}
```

Функции `animateCSS()`, определенной в этом примере, передается пять аргументов. Первый аргумент задает анимируемый объект `HTMLElement`. Второй и третий аргументы задают количество кадров анимации и продолжительность времени, в течение которого должен отображаться каждый кадр. Четвертый аргумент – это JavaScript-объект, описывающий выполняемую анимацию. Пятый аргумент – это необязательная функция, которая должна быть вызвана один раз по завершении анимации.

Ключевым является четвертый аргумент функции animateCSS(). Каждое свойство JavaScript-объекта должно иметь то же имя, что и свойство CSS-стиля, а значением каждого свойства должна быть функция, возвращающая допустимое значение для стиля с этим именем. При отображении нового кадра вызывается каждая из этих функций, чтобы сгенерировать новые значения для каждого атрибута стиля. Каждой функции передается номер кадра и общее время, прошедшее с начала анимации, и функция может использовать эти аргументы для вычисления нужного значения.

Код примера 16.6 достаточно прост; как мы скоро увидим, вся сложность заключена в свойствах объекта анимации, которые передаются функции animateCSS(). Функция animateCSS() определяет вложенную функцию с именем displayNextFrame() и почти ничего не делает, разве что с помощью setInterval() настраивает периодический вызов displayNextFrame().Функция displayNextFrame() выполняет цикл по свойствам объекта анимации и вызывает различные функции для вычисления новых значений свойств стиля.

Обратите внимание: поскольку функция displayNextFrame() определена внутри animateCSS(), она имеет доступ к аргументам и локальным переменным animateCSS(), несмотря даже на то, что displayNextFrame() вызывается уже после завершения работы функции animateCSS()! (Если вы не понимаете, почему, вернитесь к разделу 8.8.)

Следующий пример должен в значительной мере прояснить применение функции animateCSS(). Данный код перемещает элемент вверх по экрану, при этом постепенно открывая его путем увеличения области отсечения.

```
// Анимируем элемент с идентификатором "title", используя 40 кадров
// по 50 миллисекунд каждый
animateCSS(document.getElementById("title"), 40, 50,
           { // Так устанавливаются свойства top и clip для каждого кадра:
             top: function(f,t) { return 300f*5 + "px"; }
             clip: function(f,t) {return "rect(auto "+f*10+"px auto auto)";},
           });
```

Следующий фрагмент кода с помощью функции animateCSS() перемещает объект Button по кругу. Необязательный пятый аргумент передается функции для изменения текста кнопки на «Готово», когда анимация завершается. Функции, указанной пятым аргументом, в качестве аргумента передается анимируемый элемент:

```
// Перемещаем кнопку по кругу, а затем изменяем выводимый на ней текст
animateCSS(document.forms[0].elements[0], 40, 50, // Кнопка, 40 кадров, 50 мс
           { // Эта тригонометрия определяет круг радиусом 100 и с центром
             // в точке (200,200):
               left: function(f,t){ return 200 + 100*Math.cos(f/8) + "px"},
               top: function(f,t){ return 200 + 100*Math.sin(f/8) + "px"}
           },
function(button) { button.value = "Готово"; });
```

В JavaScript-библиотеке Scriptaculos представляет собой достаточно сложную платформу для создания анимации с множеством предопределенных интересных анимационных эффектов. Чтобы узнать об этой библиотеке, посетите сайт с остроумным названием *http://script.aculo.us/*.

16.4. Вычисляемые стили

Свойство `style` HTML-элемента соответствует HTML-атрибуту `style`, а объект `CSS2Properties`, который является значением свойства `style`, включает в себя информацию только о встроенных стилях для данного элемента. В нем отсутствуют сведения о стилях, находящихся в других местах в каскаде.

Иногда требуется знать точный набор стилей, применяемых к элементу, независимо от того, где в каскаде стилей они определены. То, что вы хотите получить, называется вычисляемым стилем элемента. К сожалению, название *вычисляемый стиль* (*computed style*) довольно расплывчато – оно означает вычисления, выполняемые до того, как элемент будет отображен веб-броузером: проверяются правила всех таблиц стилей и выясняется, какие из них должны быть применены к элементу, после чего стили этих правил объединяются со всеми встроенными стилями элемента. Затем полученная совокупная информация о стиле используется для корректного вывода элемента в окне броузера.

В соответствии со стандартом W3C API для определения вычисляемых стилей элементов должен использоваться метод `getComputedStyle()` объекта `Window`. Первым аргументом методу передается элемент, стиль которого требуется получить. Вторым аргументом передается любой псевдоэлемент, такой как «:before» или «:after», стиль которого желательно получить. Скорее всего, вас не будут интересовать псевдоэлементы, но дело в том, что в реализации этого метода в Mozilla и Firefox второй аргумент является обязательным и не может быть опущен. В результате вам часто будут встречаться вызовы `getComputedStyle()` со значением `null` во втором аргументе.

Метод `getComputedStyle()` возвращает объект `CSS2Properties`, который представляет все стили, применяемые к заданному элементу или псевдоэлементу. В отличие от объекта `CSS2Properties`, который хранит информацию о встроенном стиле, объект, возвращаемый `getComputedStyle()`, доступен только для чтения.

Броузер IE не поддерживает метод `getComputedStyle()`, но предоставляет простую альтернативу: каждый HTML-элемент имеет свойство `currentStyle`, в котором хранится вычисляемый стиль. Единственный недостаток прикладного интерфейса (API) броузера IE заключается в том, что он не предусматривает возможности получения стилей псевдоэлементов.

Далее приводится программный код, который не зависит от платформ и определяет шрифт, используемый при отображении элемента:

```
var p = document.getElementsByTagName("p")[0]; // Получить первый абзац
var typeface = "";                             // Требуется получить шрифт
if (p.currentStyle)                            // Сначала попробовать IE API
    typeface = p.currentStyle.fontFamily;
else if (window.getComputedStyle)              // Иначе - W3C API
    typeface = window.getComputedStyle(p, null).fontFamily;
```

Вычисляемые стили капризны, и при попытке получить их значение требуемая информация возвращается не всегда. Рассмотрим только что продемонстрированный пример. Для повышения степени переносимости между платформами CSS-атрибут `font-family` принимает список желательных семейств шрифтов, разделенных запятыми. Когда запрашивается значение атрибута `fontFamily` вычис-

ленного стиля, вы просто получаете значение стиля font-family, который применяется к элементу. В результате можно получить список, такой как «arial, helvetica, sans-serif», который ничего не говорит о фактически использованном шрифте. Точно так же, если элемент не позиционируется в абсолютных координатах, попытки получить его положение и размеры с помощью свойств top и left вычисленного стиля дают значение "auto". Это вполне нормальное для CSS значение, но, скорее всего, совсем не то, что вы пытались узнать.

16.5. CSS-классы

Альтернативой использованию отдельных CSS-стилей через свойство style является применение значения атрибута class через свойство className любого HTML-элемента. Динамическое изменение класса элемента может приводить к существенным изменениям стилей, применяемых к этому элементу, при этом, конечно же, предполагается, что используемый класс соответствующим образом определен в таблице стилей. Данный прием реализован в примере 18.3, выполняющем проверку правильности заполнения формы. JavaScript-код в этом примере устанавливает свойство className элементов формы в значение «valid» (верно) или «invalid» (неверно) в зависимости от того, правильная ли информация была введена пользователем. В состав примера 18.2 включена простая таблица стилей, которая определяет классы "valid" и "invalid" так, чтобы с их помощью можно было изменять цвет фона элементов ввода в форме.

Главное, что необходимо помнить об HTML-атрибуте class и соответствующем ему свойстве className, – они могут содержать более одного класса. Вообще, при работе со свойством className не принято просто устанавливать или читать его значение, как если бы это свойство содержало единственное имя класса (хотя, с целью упрощения, именно так и делается в главе 18). Вместо этого необходимо пользоваться функцией, позволяющей проверить принадлежность элемента классу, а также функциями добавления классов в свойство className элемента и их удаления из свойства className. В примере 16.7 приводятся определения таких функций. Программный код прост, но в его основе лежат регулярные выражения.

Пример 16.7. Вспомогательные функции для работы со свойством className

```
/**
 * CSSClass.js: функции для работы с CSS-классами HTML-элемента.
 *
 * Данный модуль определяет единственный глобальный символ с именем CSSClass.
 * Этот объект содержит вспомогательные функции для работы с атрибутом class
 * (свойством className) HTML-элементов. Все функции принимают два аргумента:
 * элемент e, принадлежность которого к CSS-классу требуется проверить
 * или изменить, и собственно CSS-класс c, принадлежность к которому проверяется,
 * или который добавляется или удаляется. Если элемент e - строка,
 * он воспринимается как значение атрибута id и передается методу
 * document.getElementById().
 */
var CSSClass = {}; // Создать объект пространства имен

// Возвращает true, если элемент e является членом класса, в противном случае - false.
CSSClass.is = function(e, c) {
    if (typeof e == "string") e = document.getElementById(e); // id элемента
```

```
    // Прежде чем выполнять поиск с помощью регулярного выражения,
    // оптимизировать для пары наиболее распространенных случаев.
    var classes = e.className;
    if (!classes) return false;    // Не является членом ни одного класса
    if (classes == c) return true; // Член этого класса

    // Иначе использовать регулярное выражение для поиска слова с
    // \b в регулярных выражениях означает совпадение с границей слова.
    return e.className.search("\\b" + c + "\\b") != -1;
};

// Добавляет класс с в свойство className элемента e,
// если имя класса уже не было записано ранее.
CSSClass.add = function(e, c) {
    if (typeof e == "string") e = document.getElementById(e); // id элемента
    if (CSSClass.is(e, c)) return; // Если уже член класса - ничего не делать
    if (e.className) c = " " + c;  // Добавить разделяющий пробел, если необходимо
    e.className += c;              // Добавить новый класс в конец
};

// Удалить все вхождения (если таковые имеются) класса с из свойства
// className элемента e
CSSClass.remove = function(e, c) {
    if (typeof e == "string") e = document.getElementById(e); // id элемента

    // Отыскать в className все вхождения с и заменить их "".
    // \s* соответствует любому числу пробельных символов.
    // "g" заставляет регулярное выражения искать все вхождения
    e.className = e.className.replace(new RegExp("\\b"+ c+"\\b\\s*", "g"), "");
};
```

16.6. Таблицы стилей

В предыдущих разделах описывались два приема работы с CSS: изменение встроенных стилей элементов и изменение класса элемента. Однако существует еще возможность изменения самих таблиц стилей, что демонстрируется в следующих подразделах.

16.6.1. Включение и выключение таблиц стилей

Простейший прием работы с таблицами стилей является к тому же самым переносимым и устойчивым. Стандарт HTML DOM Level 2 определяет свойство `disabled` для элементов `<link>` и `<style>`. HTML-теги не имеют соответствующего атрибута, но это свойство доступно JavaScript-сценариям. Как следует из его имени, если свойство `disabled` принимает значение `true`, таблица стилей, связанная с данным элементом `<link>` или `<style>`, будет запрещена, и в результате она игнорируется броузером.

Это наглядно демонстрирует пример 16.8. Он представляет HTML-страницу, которая включает четыре таблицы стилей. На странице выводятся четыре флажка, дающие пользователю возможность разрешать или запрещать применение каждой из четырех таблиц стилей.

Пример 16.8. Включение и выключение таблиц стилей

```
<head>
<!-- Здесь с помощью <link> и <style> определены четыре таблицы стилей. -->
<!-- Две подключаемые внешние таблицы стилей являются альтернативными -->
<!-- и потому по умолчанию отключены. -->
<!-- Все таблицы имеют атрибут id, что позволяет обращаться к ним по имени. -->
<link rel="stylesheet" type="text/css" href="ss0.css" id="ss0">
<link rel="alternate stylesheet" type="text/css" href="ss1.css"
      id="ss1" title="Крупный шрифт">
<link rel="alternate stylesheet" type="text/css" href="ss2.css"
      id="ss2" title="Высокий контраст">
<style id="ss3" title="Sans Serif">
body { font-family: sans-serif; }
</style>

<script>
// Эта функция включает или выключает таблицу стилей с заданным атрибутом id.
// Она работает с элементами <link> и <style>.
function enableSS(sheetid, enabled) {
    document.getElementById(sheetid).disabled = !enabled;
}
</script>
</head>
<body>

<!-- Это простая HTML-форма, которая позволяет включать и выключать таблицы стилей. -->
<!-- Здесь жестко определены имена таблиц в документе, но можно -->
<!-- определять их динамически на основе заголовков. -->
<form>
<input type="checkbox"
       onclick="enableSS('ss0', this.checked)" checked>Основная
<br><input type="checkbox"
           onclick="enableSS('ss1', this.checked)">Крупный шрифт
<br><input type="checkbox"
           onclick="enableSS('ss2', this.checked)">Высокий контраст
<br><input type="checkbox"
           onclick="enableSS('ss3', this.checked)" checked>Sans Serif
</form>
</body>
```

16.6.2. Объекты и правила таблиц стилей

В дополнение к возможности включения и отключения тегов <link> и <style>, которые *ссылаются* на таблицы стилей, модель DOM Level 2 определяет API для получения, обхода и манипулирования самими таблицами стилей. К моменту написания этих строк значительная часть стандарта прикладного интерфейса (API) для обхода таблиц стилей поддерживалась только одним броузером – Firefox. В IE 5 используется иной прикладной интерфейс, а остальные броузеры имеют ограниченную (или вообще не имеют) поддержку средств непосредственного манипулирования таблицами стилей.

Как правило, непосредственное манипулирование таблицами стилей редко бывает полезным. Вместо того чтобы добавлять новые правила в таблицы стилей,

обычно лучше оставить их статичными и работать со свойством `className` элемента. В то же время, если необходимо предоставить пользователю возможность полного управления таблицами стилей веб-страницы, может потребоваться организовать динамическое манипулирование таблицами (возможно с сохранением пользовательских предпочтений в виде cookie-файлов). Если будет принято решение о реализации прямого манипулирования таблицами стилей, то в этом вам поможет программный код, который приводится в данном разделе. Этот код работает в броузерах Firefox и IE, но возможно не будет работать в других броузерах.

Таблицы стилей, применяемые к документу, хранятся в массиве `styleSheets[]` в объекте документа. Если в документе определена единственная таблица стилей, к ней можно обратиться так:

```
var ss = document.styleSheets[0]
```

Элементами этого массива являются объекты `CSSStyleSheet`. Обратите внимание: эти объекты – *не то же самое*, что теги `<link>` или `<style>`, которые ссылаются на таблицы стилей или содержат эти таблицы. Объект `CSSStyleSheet` имеет свойство-массив `cssRules[]`, где хранятся правила стилей:

```
var firstRule = document.styleSheets[0].cssRules[0];
```

Броузер IE не поддерживает свойство `cssRules`, но имеет эквивалентное ему свойство `rules`.

Элементами массивов `cssRules[]` и `rules[]` являются объекты `CSSRule`. В соответствии со стандартами W3C объект `CSSRule` может представлять CSS-правила любого типа, включая *@-правила*, такие как директивы `@import` и `@page`. Однако в IE объект `CSSRule` может представлять только фактические правила таблицы стилей.

Объект `CSSRule` имеет два свойства, которые могут использоваться переносимым способом. (В W3C DOM правила, не относящиеся к правилам стилей, не имеют этих свойств и потому, возможно, вам потребуется пропускать их при обходе таблицы стилей.) Свойство `selectorText` – это CSS-селектор для данного правила, а свойство `style` – это ссылка на объект `CSS2Properties`, который описывает стили, связанные с этим селектором. Ранее уже говорилось, что `CSS2Properties` – это интерфейс доступа к встроенным стилям HTML-элементов через свойство `style`. Объект `CSS2Properties` может применяться для чтения существующих или записи новых значений стилей в правилах. Нередко при обходе таблицы стилей интерес представляет сам текст правила, а не разобранное его представление. В этом случае можно использовать свойство `cssText` объекта `CSS2Properties`, в котором содержатся правила в текстовом представлении.

Следующий фрагмент реализует обход правил таблицы стилей в цикле и наглядно демонстрирует, что с ними можно делать:

```
// Получить ссылку на первую таблицу стилей документа
var ss = document.styleSheets[0];

// Получить массив правил, используя W3C или IE API
var rules = ss.cssRules?ss.cssRules:ss.rules;

// Обойти правила в цикле
for(var i = 0; i < rules.length; i++) {
    var rule = rules[i];
```

```
    // Пропустить @import и другие правила, не являющиеся определениями стилей
    if (!rule.selectorText) continue;

    // Это текстовое представление правила
    var ruleText = rule.selectorText + " { " + rule.style.cssText + " }";

    // Если правило определяет ширину поля, предположить,
    // что в качестве единиц измерения используются пикселы, и удвоить их
    var margin = parseInt(rule.style.margin);
    if (margin) rule.style.margin = (margin*2) + "px";
}
```

В дополнение к возможности получения и изменения существующих правил таблиц стилей существует возможность добавлять правила в таблицу стилей и удалять их из таблицы. W3C-интерфейс `CSSStyleSheet` определяет методы `insertRule()` и `deleteRule()`, позволяющие добавлять и удалять правила:

```
document.styleSheets[0].insertRule("H1 { text-weight: bold; }", 0);
```

Броузер IE не поддерживает методы `insertRule()` и `deleteRule()`, но определяет практически эквивалентные им функции `addRule()` и `removeRule()`. Единственное существенное отличие (помимо имен функций) состоит в том, что `addRule()` ожидает получить тексты селектора и стиля в виде двух отдельных аргументов. В примере 16.9 приводится определение вспомогательного класса `Stylesheet`, в котором демонстрируется использование прикладных интерфейсов (API) W3C и IE для добавления и удаления правил.

Пример 16.9. Вспомогательные методы для работы с таблицами стилей

```
/**
 * Stylesheet.js: вспомогательные методы для работы с таблицами CSS-стилей.
 *
 * Этот модуль объявляет класс Stylesheet, который представляет собой просто
 * обертку для массива document.styleSheets[]. Он определяет удобные
 * межплатформенные методы чтения и изменения таблиц стилей.
 **/

// Создает новый объект Stylesheet, который служит оберткой
// для заданного объекта CSSStylesheet.
// Если аргумент ss - это число, найти таблицу стилей в массиве styleSheet[].
function Stylesheet(ss) {
    if (typeof ss == "number") ss = document.styleSheets[ss];
    this.ss = ss;
}

// Возвращает массив правил для заданной таблицы стилей.
Stylesheet.prototype.getRules = function() {
    // Если определено W3C-свойство, использовать его,
    // в противном случае использовать IE-свойство
    return this.ss.cssRules?this.ss.cssRules:this.ss.rules;
}

// Возвращает правило из таблицы стилей. Если s - это число, возвращается
// правило с этим индексом. Иначе предположить, что s - это селектор,
// тогда следует отыскать правило, соответствующее этому селектору.
Stylesheet.prototype.getRule = function(s) {
```

```
    var rules = this.getRules();
    if (!rules) return null;
    if (typeof s == "number") return rules[s];

    // Предположить, что s - это селектор
    // Обойти правила в обратном порядке, чтобы в случае нескольких
    // правил с одинаковым селектором мы получили правило с наивысшим приоритетом.
    s = s.toLowerCase();
    for(var i = rules.length-1; i >= 0; i--) {
        if (rules[i].selectorText.toLowerCase() == s) return rules[i];
    }
    return null;
};

// Возвращает объект CSS2Properties для заданного правила.
// Правило может задаваться номером или селектором.
Stylesheet.prototype.getStyles = function(s) {
    var rule = this.getRule(s);
    if (rule && rule.style) return rule.style;
    else return null;
};

// Возвращает текст стиля для заданного правила.
Stylesheet.prototype.getStyleText = function(s) {
    var rule = this.getRule(s);
    if (rule && rule.style && rule.style.cssText) return rule.style.cssText;
    else return "";
};

// Вставляет правило в таблицу стилей.
// Правило состоит из заданного селектора и строк стилей.
// Вставляется под индексом n. Если аргумент n опущен, правило
// добавляется в конец таблицы.
Stylesheet.prototype.insertRule = function(selector, styles, n) {
    if (n == undefined) {
        var rules = this.getRules();
        n = rules.length;
    }
    if (this.ss.insertRule)   // Сначала попробовать использовать W3C API
        this.ss.insertRule(selector + "{" + styles + "}", n);
    else if (this.ss.addRule) // Иначе использовать IE API
        this.ss.addRule(selector, styles, n);
};

// Удаляет правило из таблицы стилей.
// Если s - это число, удаляет правило с этим номером.
// Если s - это строка, удаляет правило с этим селектором.
// Если аргумент s не задан, удаляет последнее правило в таблице стилей.
Stylesheet.prototype.deleteRule = function(s) {
    // Если значение s не определено, превратить его в индекс
    // последнего правила
    if (s == undefined) {
        var rules = this.getRules( );
        s = rules.length-1;
    }
```

```
    // Если s - не число, отыскать соответствующее правило
    // и получить его индекс.
    if (typeof s != "number") {
        s = s.toLowerCase( ); // Преобразовать в нижний регистр
        var rules = this.getRules();
        for(var i = rules.length-1; i >= 0; i--) {
            if (rules[i].selectorText.toLowerCase() == s) {
                s = i; // Запомнить индекс удаляемого правила
                break; // и прекратить дальнейший поиск
            }
        }

        // Если правило не было найдено, просто ничего не делать.
        if (i == -1) return;
    }

    // В этой точке s содержит число.
    // Сначала попробовать использовать W3C API, а затем - IE API
    if (this.ss.deleteRule) this.ss.deleteRule(s);
    else if (this.ss.removeRule) this.ss.removeRule(s);
};
```

17

События и обработка событий

Как мы видели в главе 13, интерактивные JavaScript-программы основаны на модели программирования, управляемого событиями. При таком стиле программирования веб-броузер генерирует событие, когда с документом или некоторым его элементом что-то происходит. Например, веб-броузер генерирует событие, когда завершает загрузку документа, когда пользователь наводит указатель мыши на гиперссылку или щелкает на кнопке в форме. Если JavaScript-приложение интересует определенный тип события для определенного элемента документа, оно может зарегистрировать *обработчик события* (*event handler*) – JavaScript-функцию или фрагмент JavaScript-кода для этого типа события в интересующем вас элементе. Потом, когда возникает это событие, броузер вызовет код обработчика. Все приложения с графическим интерфейсом пользователя разработаны подобным образом: они ждут, пока пользователь что-нибудь сделает (т. е. ждут, когда произойдут события), и затем реагируют на его действия.

В качестве отступления следует отметить, что таймеры и обработчики ошибок (описание тех и других вы найдете в главе 14) связаны с программированием, управляемым событиями. Работа таймеров и обработчиков ошибок, как и обработчиков событий, описанных в этой главе, основана на регистрации функции в броузере и последующем вызове броузером этой функции, когда происходит соответствующее событие. Однако в этом случае событием является истечение указанного промежутка времени или возникновение ошибки при выполнении JavaScript-кода. Хотя таймеры и обработчики ошибок в данной главе не обсуждаются, их можно рассматривать как средства, относящиеся к обработке событий, и я рекомендую вам заново прочитать разделы 14.1 и 14.7, переосмыслив их в контексте данной главы.

Обработчики событий активно применяются в нетривиальных JavaScript-программах. Некоторые примеры JavaScript-кода с простыми обработчиками событий мы уже видели. Данная глава заполняет подробностями все пробелы в теме событий и их обработки. К сожалению, эти подробности сложнее, чем должны

были быть, поскольку дальнейшее обсуждение основано на четырех различных и несовместимых моделях обработки событий.[1] Вот эти модели.

Исходная модель обработки событий

Эта простая модель используется (хотя и без подробного документирования) в данной книге. В ограниченном объеме она была кодифицирована стандартом HTML 4 и неформально рассматривается как часть прикладного интерфейса (API) DOM Level 0. Несмотря на ограниченность ее возможностей, она реализована всеми веб-броузерами, поддерживающими JavaScript, и потому переносима.

Стандартная модель обработки событий

Это мощная и богатая возможностями модель была стандартизована в DOM уровня 2. Поддерживается всеми современными броузерами, исключая Internet Explorer.

Модель обработки событий Internet Explorer

Эта модель реализована в IE 4 и расширена в IE 5. Она обладает некоторыми, но не всеми возможностями стандартной модели обработки событий. Хотя корпорация Microsoft участвовала в создании модели обработки событий DOM Level 2 и располагала достаточным временем для реализации стандартной модели обработки событий в IE 5.5 и IE 6, разработчики этих броузеров продолжают придерживаться своей фирменной модели обработки событий.[2] Это значит, что JavaScript-программисты должны писать для броузеров IE специальный программный код, если хотят получить доступ к развитым механизмам обработки событий.

В этой главе описываются все модели обработки событий. Описание трех моделей сопровождается тремя разделами, включающими расширенные примеры обработки событий мыши, клавиатуры и события `onload`. Заканчивается глава кратким обсуждением темы генерации и отправки искусственных событий.

17.1. Базовая обработка событий

В рассмотренных ранее примерах обработчики событий записывались в виде строк JavaScript-кода, выступающих в качестве значений определенных HTML-атрибутов, таких как `onclick`. Это основа исходной модели обработки событий, но есть некоторые дополнительные нюансы, требующие понимания и рассмотренные в следующих разделах.

17.1.1. События и типы событий

Различные типы происшествий генерируют различные типы событий. Наводя мышь на гиперссылку и щелкая кнопкой мыши, пользователь вызывает собы-

---

[1] Броузер Netscape 4 также имел собственную, отличную от других и несовместимую модель обработки событий. Этот броузер в основном уже вышел из употребления, по'тому его модель обработки событий в этой книге не рассматривается.

[2] Хотя к моменту написания этих строк броузер IE 7 уже находился в стадии разработки, у автора нет никакой информации о том, будет ли он поддерживать стандартную модель обработки событий.

тия разных типов. Даже одно и то же происшествие может возбуждать различные типы событий в зависимости от контекста, например, когда пользователь щелкает на кнопке Submit, возникает событие, отличное от события, возникающего при щелчке на кнопке Reset в форме.

В исходной модели обработки событий событие – это внутренняя абстракция для веб-броузера, и JavaScript-код не может непосредственно манипулировать событием. Говоря о типе события в исходной модели обработки событий, мы на самом деле имеем в виду имя обработчика, вызываемого в ответ на событие. В этой модели код обработки событий задается с помощью атрибутов HTML-элементов (и соответствующих свойств связанных с ними JavaScript-объектов). Следовательно, если приложению требуется знать, что пользователь навел мышь на определенную гиперссылку, то используется атрибут `onmouseover` тега `<a>`, определяющего эту гиперссылку. А если приложению требуется знать, что пользователь щелкнул на кнопке Submit, используется атрибут `onclick` тега `<input>`, определяющего кнопку, или атрибут `onsubmit` элемента `<form>`, содержащего эту кнопку.

Имеется довольно много различных атрибутов обработчиков событий, которые можно использовать в исходной модели обработки событий. Они перечислены в табл. 17.1, где также указано, когда вызываются эти обработчики событий и какие HTML-элементы поддерживают атрибуты обработчиков.

В процессе развития клиентского JavaScript-программирования развивалась и поддерживаемая им модель обработки событий. В каждую новую версию броузера добавлялись новые атрибуты обработчиков событий. И наконец, спецификация HTML 4 закрепила стандартный набор атрибутов обработчиков событий для HTML-тегов. В третьем столбце табл. 17.1 указано, какие HTML-элементы поддерживают каждый из атрибутов обработчиков событий. Для событий мыши в третьей колонке указывается, что атрибут обработчика события поддерживает большинство элементов. HTML-элементы, которые не поддерживают данный тип событий, обычно размещаются в разделе `<head>` документа или не имеют графического представления. К элементам, не поддерживающим практически универсальные атрибуты обработчиков событий мыши, относятся `<applet>`, `<bdo>`, `<br>`, `<font>`, `<frame>`, `<frameset>`, `<head>`, `<html>`, `<iframe>`, `<isindex>`, `<meta>` и `<style>`.

Таблица 17.1. Обработчики событий и поддерживающие их HTML-элементы

Обработчик	Условия вызова	Поддержка
`onabort`	Прерывание загрузки изображения	`<img>`
`onblur`	Элемент теряет фокус ввода	`<button>`, `<input>`, `<label>`, `<select>`, `<textarea>`, `<body>`
`onchange`	Элемент `<select>` или другой элемент потерял фокус и его значение с момента получения фокуса изменилось	`<input>`, `<select>`, `<textarea>`
`onclick`	Была нажата и отпущена кнопка мыши; следует за событием `mouseup`. Возвращает `false` для отмены действия по умолчанию (т. е. перехода по ссылке, очистки формы, передачи данных)	Большинство элементов
`ondblclick`	Двойной щелчок	Большинство элементов

Таблица 17.1 (продолжение)

Обработчик	Условия вызова	Поддержка
onerror	Ошибка при загрузке изображения	<img>
onfocus	Элемент получил фокус ввода	<button>, <input>, <label>, <select>, <textarea>, <body>
onkeydown	Клавиша нажата. Для отмены возвращает false	Элементы формы и <body>
onkeypress	Клавиша нажата и отпущена. Для отмены возвращает false	Элементы формы и <body>
onkeyup	Клавиша отпущена	Элементы формы и <body>
onload	Загрузка документа завершена	<body>, <frameset>, <img>
onmousedown	Нажата кнопка мыши	Большинство элементов
onmousemove	Перемещение указателя мыши	Большинство элементов
onmouseout	Указатель мыши выходит за границы элемента	Большинство элементов
onmouseover	Указатель мыши находится на элементе	Большинство элементов
onmouseup	Отпущена кнопка мыши	Большинство элементов
onreset	Запрос на очистку полей формы. Для предотвращения очистки возвращает false	<form>
onresize	Изменение размеров окна	<body>, <frameset>
onselect	Выбор текста	<input>, <textarea>
onsubmit	Запрос на передачу данных формы. Чтобы предотвратить передачу, возвращает false	<form>
onunload	Документ или набор фреймов выгружен	<body>, <frameset>

17.1.1.1. Аппаратно-зависимые и аппаратно-независимые события

При внимательном изучении табл. 17.1 можно заметить, что все события делятся на две большие категории. Первая категория – это *события ввода* (*raw events*, или *input events*). Эти события генерируются, когда пользователь перемещает мышь, щелкает на кнопке мыши или нажимает клавишу. Эти низкоуровневые события просто описывают действия пользователя и не имеют другого смысла. Вторая категория событий – это *семантические события* (*semantic events*). Это высокоуровневые события, они имеют более сложный смысл и обычно происходят только в определенном контексте: когда броузер завершает загрузку документа или, например, когда должна выполниться передача данных формы. Семантическое событие часто происходит как побочный эффект низкоуровневого события. Например, когда пользователь щелкает на кнопке Submit, вызываются три обработчика событий ввода: onmousedown, onmouseup и onclick. И в результате щелчка на кнопке мыши HTML-форма, содержащая кнопку Submit, генерирует семантическое событие onsubmit.

Другое существенное отличие делит события на аппаратно-зависимые, связанные с мышью или клавиатурой, и аппаратно-независимые события, которые могут возбуждаться несколькими способами. Это различие особенно важно в плане доступности (см. раздел 13.7), поскольку одни пользователи в состоянии задействовать мышь, но не могут работать с клавиатурой, другие, наоборот, могут применять клавиатуру и не могут мышь. Семантические события, такие как `onsubmit` и `onchange`, практически всегда являются аппаратно-независимыми: все современные броузеры позволяют выполнять переход между полями HTML-форм как с помощью мыши, так и с помощью клавиатуры. События, которые имеют в своих названиях слово «key» или «mouse», совершенно очевидно являются аппаратно-зависимыми. Если вы собираетесь использовать эти события, возможно, следует реализовать обработчики для парных событий, чтобы обеспечить механизм обработки событий как мыши, так и клавиатуры. Примечательно, что событие `onclick` можно рассматривать как аппаратно-независимое. Оно не зависит от мыши, потому что активизация с помощью клавиатуры элементов формы и гиперссылок тоже приводит к возбуждению этого события.

17.1.2. Обработчики событий как атрибуты

Как мы видели в примерах из предыдущих глав, обработчики событий (в исходной модели обработки событий) задаются в виде строк JavaScript-кода, присваиваемых в качестве значений HTML-атрибутам. Например, чтобы выполнить JavaScript-код при щелчке на кнопке, укажите этот код в качестве значения атрибута `onclick` тега `<input>` (или `<button>`):

```
<input type="button" value="Нажми меня" onclick="alert('спасибо');">
```

Значение атрибута обработчика события – это произвольная строка JavaScript-кода. Если обработчик состоит из нескольких JavaScript-инструкций, они *должны* отделяться друг от друга точками с запятой. Например:

```
<input type="button" value="Щелкни здесь"
       onclick="if (window.numclicks) numclicks++; else numclicks=1;
                this.value='Щелчок # ' + numclicks;">
```

Если обработчик события требует нескольких инструкций, то, как правило, проще определить его в теле функции и затем задать HTML-атрибут обработчика события для вызова этой функции. Например, проверить введенные пользователем в форму данные перед их отправкой можно при помощи атрибута `onsubmit` тега `<form>`.[1] Проверка формы обычно требует как минимум нескольких строк кода, поэтому не надо помещать весь этот код в одно длинное значение атрибута, разумнее определить функцию проверки формы и просто задать атрибут `onsubmit` для вызова этой функции. Например, если для проверки определить функцию с именем `validateForm()`, то можно вызывать ее из обработчика события следующим образом:

```
<form action="processform.cgi" onsubmit="return validateForm();">
```

[1] Подробное описание HTML-форм, включая пример проверки правильности заполнения полей формы, приводится в главе 18.

Помните, что язык HTML нечувствителен к регистру, поэтому в атрибутах обработчиков событий допускаются буквы любого регистра. Одно из распространенных соглашений состоит в употреблении символов различных регистров, при этом префикс «on» записывается в нижнем регистре: `onClick`, `onLoad`, `onMouseOut` и т. д. Однако в этой книге для совместимости с языком XHTML, чувствительным к регистру, я предпочел везде нижний регистр.

JavaScript-код в атрибуте обработчика события может содержать инструкцию `return`, а возвращаемое значение может иметь для броузера специальный смысл. Вскоре мы это обсудим. Кроме того, следует отметить, что JavaScript-код обработчика события работает в области видимости (см. главу 4), отличной от глобальной. Это также более подробно обсуждается далее в этом разделе.

17.1.3. Обработчики событий как свойства

Как уже говорилось в главе 15, каждому HTML-элементу в документе соответствует DOM-элемент в дереве документа, и свойства этого JavaScript-объекта соответствуют атрибутам HTML-элемента. Это относится также к атрибутам обработчиков событий. Поэтому если в теге `<input>` имеется атрибут `onclick`, к указанному в нем обработчику событий можно обратиться с помощью свойства `onclick` объекта элемента формы. (Язык JavaScript чувствителен к регистру, поэтому независимо от регистра символов в имени HTML-атрибута JavaScript-свойство должно быть записано целиком в нижнем регистре.)

Так как значение HTML-атрибута, определяющего обработчик событий, является строкой JavaScript-кода, можно предполагать, что соответствующее JavaScript-свойство также представляет собой строку. Но это не так: при обращении посредством JavaScript-кода свойства обработчиков событий представляют собой функции. Убедиться в этом можно с помощью простого примера:

```
<input type="button" value="Щелкни здесь"
       onclick="alert(typeof this.onclick);">
```

Если щелкнуть на кнопке, откроется диалоговое окно, содержащее слово «function», а не слово «string». (Обратите внимание: в обработчиках событий ключевое слово `this` ссылается на объект, в котором произошло событие. Позже мы обсудим ключевое слово `this`.)

Чтобы назначить обработчик события элементу документа с помощью JavaScript, установите свойство-обработчик события равным нужной функции. Рассмотрим, например, следующую HTML-форму:

```
<form name="f1">
<input name="b1" type="button" value="Нажми меня">
</form>
```

На кнопку в этой форме можно сослаться с помощью выражения `document.f1.b1`, значит, обработчик события можно установить с помощью следующей строки кода:

```
document.f1.b1.onclick=function() { alert('Спасибо!'); };
```

Кроме того, обработчик события может быть установлен так:

```
function plead() { document.f1.b1.value += ", пожалуйста!"; }
document.f1.b1.onmouseover = plead;
```

Обратите особое внимание на последнюю строку: здесь после имени функции нет скобок. Чтобы определить обработчик события, мы присваиваем свойству-обработчику события саму функцию, а не результат ее вызова. На этом часто «спотыкаются» начинающие JavaScript-программисты.

В представлении обработчиков событий в виде JavaScript-свойств есть два преимущества. Во-первых, это сокращает степень смешения HTML- и JavaScript-кода, стимулируя модульность и позволяя получить более ясный и легко сопровождаемый код. Во-вторых, благодаря этому функции-обработчики событий являются динамическими. В отличие от HTML-атрибутов, которые представляют собой статичную часть документа и могут устанавливаться только при его создании, JavaScript-свойства могут изменяться в любое время. В сложных интерактивных программах иногда полезно динамически изменять обработчики событий, зарегистрированные для HTML-элементов.

Один небольшой недостаток определения обработчиков событий в JavaScript состоит в том, что это отделяет обработчик от элемента, которому он принадлежит. Если пользователь начнет взаимодействовать с элементом документа до его полной загрузки (и до исполнения всех его сценариев), обработчики событий для элемента могут оказаться неопределенными.

В примере 17.1 демонстрируется, как назначить функцию обработчиком события для нескольких элементов документа. Данный пример представляет собой простую функцию, определяющую обработчик события `onclick` для каждой ссылки в документе. Обработчик события запрашивает подтверждение пользователя, перед тем как разрешить переход по ссылке, на которой пользователь только что щелкнул. Если пользователь не дал подтверждения, функция-обработчик возвращает `false`, что не дает броузеру перейти по ссылке. Возвращаемые обработчиками событий значения обсуждаются в ближайших разделах.

Пример 17.1. Одна функция, много обработчиков событий

```
// Эта функция подходит для работы в качестве обработчика события
// onclick элементов <a> и <area>. Она использует ключевое слово this
// для обращения к элементу документа и может возвращать false
// для отмены перехода по ссылке.
function confirmLink() {
    return confirm("Вы действительно хотите посетить " + this.href + "?");
}

// Эта функция выполняет цикл по всем гиперссылкам в документе и назначает
// каждой из них функцию confirmLink в качестве обработчика события.
// Не вызывайте ее до того, как документ будет проанализирован
// и все ссылки определены. Лучше всего вызвать ее из обработчика
// события onload тега <body>.
function confirmAllLinks() {
    for(var i = 0; i < document.links.length; i++) {
        document.links[i].onclick = confirmLink;
    }
}
```

17.1.3.1. Явный вызов обработчиков событий

Значения свойств-обработчиков событий представляют собой функции, следовательно, их можно непосредственно вызывать при помощи JavaScript-кода. Например, пусть для определения функции проверки формы мы задали атрибут `onsubmit` тега `<form>` и хотим проверить форму в какой-то момент до попытки передачи ее пользователем. Тогда мы можем обратиться к свойству `onsubmit` объекта `Form` для вызова функции-обработчика события. Код может выглядеть следующим образом:

```
document.myform.onsubmit();
```

Однако обратите внимание, что вызов обработчика события не является способом имитации действий, происходящих при реальном возникновении этого события. Если, например, мы вызовем метод `onclick` объекта `Link`, это не заставит броузер перейти по ссылке и загрузить новый документ. Мы лишь выполним ту функцию, которую определили в качестве значения этого свойства. (Чтобы заставить броузер загрузить новый документ, необходимо установить свойство `location` объекта `Window`, как это было продемонстрировано в главе 14.) То же самое справедливо и для метода `onsubmit` объекта `Form`, и для метода `onclick` объекта `Submit`: вызов метода запускает функцию-обработчик события, но не приводит к передаче данных формы. (Чтобы на самом деле передать данные формы, следует вызвать метод `submit()` объекта `Form`.)

Одна из причин, по которой может потребоваться явный вызов функции-обработчика события, – это желание дополнить с помощью JavaScript-кода обработчик события, который (возможно) уже определен HTML-кодом. Предположим, вы хотите предпринять специальные действия, когда пользователь щелкает на кнопке, но не хотите нарушать работу какого-либо из обработчиков события `onclick`, которые могут быть определены в самом HTML-документе. (Это один из недостатков кода в примере 17.1 – добавляя обработчик к каждой гиперссылке, вы переопределяете все обработчики события `onclick`, уже определенные для этих гиперссылок.) Этот результат достигается с помощью следующего кода:

```
var b = document.myform.mybutton; // Это интересующая нас кнопка
var oldHandler = b.onclick;       // Сохраняем HTML-обработчик события
function newHandler() { /* Здесь расположен мой код обработки события */ }
    // Теперь назначаем новый обработчик события, вызывающий как новый,
    // так и старый обработчики.
b.onclick = function() { oldHandler(); newHandler(); }
```

17.1.4. Значения, возвращаемые обработчиками событий

Во многих случаях обработчик события (заданный либо HTML-атрибутом, либо JavaScript-свойством) использует возвращаемое значение для указания дальнейшего поведения. Например, если с помощью обработчика события `onsubmit` объекта `Form` выполняется проверка формы и выясняется, что пользователь заполнил не все поля, можно вернуть из обработчика значение `false`, чтобы предотвратить фактическую передачу данных формы. Гарантировать, что форма с пустым текстовым полем передана не будет, можно следующим образом:

```
<form action="search.cgi"
      onsubmit="if (this.elements[0].value.length == 0) return false;">
<input type="text">
</form>
```

Как правило, если в ответ на событие броузер выполняет некоторое действие, предлагаемое по умолчанию, можно вернуть false, чтобы предотвратить выполнение этого действия броузером. Вернуть false для отмены действия, предлагаемого по умолчанию, можно также из обработчиков событий onclick, onkeydown, onkeypress, onmousedown, onmouseup и onreset. Второй столбец табл. 17.1 содержит информацию о том, что происходит при возврате из обработчиков событий значения false.

Из правила о возвращении значения false для отмены действия есть одно исключение: когда пользователь наводит мышь на гиперссылку, по умолчанию броузер отображает ее URL-адрес в строке состояния. Чтобы этого не случилось, необходимо вернуть true из обработчика события onmouseover. Например, следующий фрагмент выводит сообщение, не являющееся URL-адресом:

```
<a href="help.htm"
   onmouseover="window.status='Помогите!!'; return true;">Help</a>
```

Никакой особой причины для этого исключения нет – просто так сложилось исторически. Однако как отмечалось в главе 14, большинство современных броузеров рассматривают возможность скрытия адреса назначения как нарушение принципов безопасности и запрещают ее. Таким образом, правило «вернуть false, чтобы отменить» ныне выглядит достаточно спорным.

Обратите внимание: от обработчиков событий никогда не требуется обязательно явно возвращать значение. Если значение не возвратить, то выполняется действие, предлагаемое по умолчанию.

17.1.5. Обработчики событий и ключевое слово this

Если обработчик события определен с помощью HTML-атрибута или JavaScript-свойства, ваши действия состоят в присваивании функции свойству элемента документа. Другими словами, вы определяете новый метод элемента документа. Ваш обработчик события вызывается как метод элемента, в котором произошло событие, поэтому ключевое слово this ссылается на этот целевой элемент. Это поведение полезно и не удивительно.

Однако убедитесь, что понимаете следствия этого поведения. Предположим, у вас есть объект o с методом mymethod. Обработчик события можно зарегистрировать следующим образом:

```
button.onclick= o.mymethod;
```

В результате выполнения этой инструкции свойство button.onclick будет ссылаться на ту же функцию, что и o.mymethod. Эта функция теперь является методом и для o, и для button. Вызывая этот обработчик события, броузер вызывает функцию как метод объекта button, а не объекта o. Ключевое слово this ссылается на объект Button, а не на ваш объект o. Не ошибитесь, думая, что можно обмануть броузер, вызвав обработчик события как метод какого-либо другого объекта. Для того чтобы это сделать, необходимо явное указание, например, такое:

```
button.onclick = function() { o.mymethod(); }
```

17.1.6. Область видимости обработчиков событий

Как говорилось в разделе 8.8, JavaScript-функции относятся к лексическому контексту. Это значит, что они работают в той области видимости, в которой определены, а не в той, из которой вызываются. Если обработчик события определяется путем присваивания HTML-атрибуту строки JavaScript-кода, то при этом неявно определяется JavaScript-функция. Важно понимать, что область видимости для обработчика события, определенного подобным образом, не совпадает с областью видимости других глобальных JavaScript-функций, определенных обычным образом. Это значит, что обработчики событий, определенные в виде HTML-атрибутов, выполняются в контексте, отличном от контекста других функций.[1]

Вспомните, в главе 4 мы говорили, что область видимости функции определяется цепочкой областей видимости или списком объектов, которые по очереди просматриваются при поиске определения переменной. Когда переменная `x` разыскивается или разрешается в обычной функции, интерпретатор JavaScript сначала ищет локальную переменную или аргумент, проверяя объект вызова функции на наличие свойства с таким именем. Если такое свойство не найдено, JavaScript переходит к следующему объекту в цепочке областей видимости – глобальному объекту. Интерпретатор проверяет свойства глобального объекта, чтобы выяснить, является ли переменная глобальной.

Обработчики событий, определенные как HTML-атрибуты, имеют более сложную цепочку областей видимости, чем только что описанная. Началом цепочки областей видимости является объект вызова. Здесь определены любые аргументы, переданные обработчику события (далее в этой главе мы увидим, что в некоторых развитых моделях обработки событий обработчикам передается аргумент), а также любые локальные переменные, определенные в теле обработчика. Следующим объектом в цепочке областей видимости является, однако, не глобальный объект, а объект, который вызвал обработчик события. Предположим, что объект `Button` в HTML-форме определяется с помощью тега `<input>`, а затем назначением атрибута `onclick` определяется обработчик события. Если в коде обработчика есть переменная с именем `form`, то она разрешается в свойство `form` объекта `Button`. Это может быть удобным при создании обработчиков событий в виде HTML-атрибутов.

```
<form>
<!-- В обработчиках событий ключевое слово "this" ссылается -->
<!-- на элемент-источник события -->
<!-- Благодаря этому можно получить ссылку на соседний элемент формы -->
<!-- следующим образом -->
<input id="b1" type="button" value="Button 1"
       onclick="alert(this.form.b2.value);">
<!-- Элемент-источник также находится в цепочке областей видимости, -->
<!-- поэтому можно опустить ключевое слово "this" -->
<input id="b2" type="button" value="Button 2"
       onclick="alert(form.b1.value);">
<!-- И элемент <form> находится в цепочке областей видимости, -->
<!-- поэтому можно опустить идентификатор "form". -->
```

[1] Это важно понимать, однако хотя приведенное ниже обсуждение интересно, оно довольно сложное. При первом прочтении этой главы можно его пропустить и вернуться к нему позже.

```
<input id="b3" type="button" value="Button 3"
       onclick="alert(b4.value);">
<!-- Объект Document находится в цепочке областей видимоси, поэтому можно -->
<!-- вызывать его методы без добавления идентификатора "document". -->
<!-- Хотя такой стиль нельзя считать правильным. -->
<input id="b4" type="button" value="Button 4"
       onclick="alert(getElementById('b3').value);">
</form>
```

Как видно из этого простого примера, цепочка областей видимости обработчика события не заканчивается на объекте, определяющем обработчик события: она продолжается вверх по иерархии и включает в себя как минимум элемент `<form>`, содержащий кнопку, и объект `Document`, который содержит форму.[1] Последним объектом в цепочке области видимости является объект `Window`, как и всегда в клиентском JavaScript-коде.

Другой способ представить себе расширенную цепочку областей видимости обработчиков событий заключается в том, чтобы проанализировать порядок трансляции JavaScript-кода, находящегося в атрибуте HTML-обработчика события, в JavaScript-функцию. Рассмотрим следующие строки из предыдущего примера:

```
<input id="b3" type="button" value="Button 3"
       onclick="alert(b4.value);">
```

Эквивалентный программный код на языке JavaScript мог бы выглядеть следующим образом:

```
var b3 = document.getElementById('b3'); // Отыскивает интересующую кнопку
b3.onclick = function() {
    with (document) {
        with(this.form) {
            with(this) {
                alert(b4.value);
            }
        }
    }
}
```

Повторяющиеся инструкции `with` создают расширенную цепочку областей видимости. Если вы забыли о назначении этой не часто используемой инструкции, прочитайте раздел 6.18.

Наличие целевого объекта в цепочке областей видимости может быть полезным. В то же время наличие расширенной цепочки областей видимости, включающей другие элементы документа, может оказаться досадным неудобством. Заметьте, например, что и объект `Window`, и объект `Document` определяют методы с именем `open()`. Если идентификатор `open` применяется без уточнения, то почти всегда имеет место обращение к методу `window.open()`. Однако в обработчике события, определенном как HTML-атрибут, объект `Document` расположен в цепочке областей видимости раньше объекта `Window`, и отдельный идентификатор `open` будет ссылаться на метод `document.open()`. Аналогично посмотрим, что произойдет, ес-

[1] Точный состав цепочки областей видимости никогда не был стандартизован и может зависеть от реализации.

ли добавить свойство с именем window объекту Form (или определить поле ввода со свойством name="window"). В этом случае, если определить внутри формы обработчик события, в котором есть выражение window.open(), идентификатор window разрешится как свойство объекта Form, а не как глобальный объект Window, и у обработчиков событий внутри формы не будет простого способа обращения к глобальному объекту Window или вызова метода window.open()!

Мораль в том, что при определении обработчиков событий как HTML-атрибутов необходима аккуратность. Подобным образом лучше создавать только очень простые обработчики. В идеале они должны просто вызывать глобальную функцию, определенную в другом месте, и, возможно, возвращать ее результат:

```
<script>function validateForm() { /* Код проверки формы */ }</script>
<form onsubmit="return validateForm();">...</form>
```

Даже используя такую необычную цепочку областей видимости, простой обработчик событий, подобный приведенному, будет функционировать, однако сокращая программный код, вы минимизируете вероятность того, что длинная цепочка областей видимости нарушит правильность функционирования обработчика. Несмотря на это следует помнить, что при исполнении функций используется область видимости, в которой они определены, а не область видимости, из которой они вызываются. Поэтому, хотя метод validateForm() вызывается из области видимости, которая отличается от обычной, он будет выполняться в собственной глобальной области видимости.

Кроме того, поскольку нет стандарта на точный состав цепочки областей видимости для обработчика события, лучше всего предполагать, что она содержит только целевой элемент и глобальный объект Window. Например, нужно ссылаться на целевой элемент посредством this, а если целевым является элемент <input>, ссылаться на содержащий его объект Form с помощью form, вместо this.form. Но не полагайтесь на наличие в цепочке областей видимости объектов Form или Document. Например, не используйте идентификатор action вместо form.action или getElementById() вместо document.getElementById().

Не забывайте, что все это обсуждение области видимости обработчика события применимо только к обработчикам событий, определенным как HTML-атрибуты. Если обработчик события задается путем присваивания функции соответствующему свойству-обработчику события, то никакой специальной цепочки областей видимости не возникает, и функция выполняется в той области видимости, в которой она определена. Ею почти всегда является глобальная область видимости, если только это не вложенная функция, тогда цепочка областей видимости снова становится интересной!

17.2. Развитые средства обработки событий в модели DOM Level 2

Технологии обработки событий, рассмотренные в этой главе, являются частью модели DOM Level 0 – стандартного прикладного интерфейса (API), поддерживаемого любым броузером, который поддерживает JavaScript. Стандарт DOM Level 2 определяет развитый прикладной интерфейс обработки событий, значительно отличающийся (и значительно более мощный) от API Level 0. Стандарт

Level 2 не присоединяет существующий прикладной интерфейс к стандарту DOM, но и не отменяет API Level 0. Базовые задачи обработки событий по-прежнему можно решать средствами простого прикладного интерфейса.

Модель событий DOM Level 2 поддерживается всеми современными броузерами, за исключением Internet Explorer.

17.2.1. Распространение событий

В модели обработки событий DOM Level 0 броузер передает события тем элементам документа, в которых они происходят. Если объект имеет соответствующий обработчик события, этот обработчик запускается. И больше ничего не происходит. В DOM Level 2 ситуация сложнее. В развитой модели обработки событий, когда событие происходит в элементе документа (известном как *целевой* узел события), вызывается обработчик (или обработчики) событий целевого узла, но, кроме того, одну или две возможности обработать данное событие получает каждый из элементов-предков этого элемента. Распространение события осуществляется в три этапа. Сначала на этапе *перехвата* события распространяются от объекта `Document` вниз по дереву документа к целевому узлу. Если у какого-либо из предков целевого элемента (но не у него самого) есть специально зарегистрированный перехватывающий обработчик события, на данном этапе распространения события этот обработчик запускается. (Скоро мы узнаем, как регистрируются обычные и перехватывающие обработчики событий.)

Следующий этап распространения события происходит *в самом целевом узле*: запускаются любые предусмотренные для этого обработчики событий, зарегистрированные непосредственно в целевом узле. Этот этап аналогичен обработке событий, предоставляемой моделью событий Level 0.

Третий этап распространения события – это этап *всплывания*, на котором событие распространяется, или «всплывает», обратно, вверх по иерархии документа от целевого узла к объекту `Document`. Если на этапе перехвата по дереву документа распространяются все события, то в этапе всплывания участвуют не все типы событий: например, событие `submit` не имеет смысла распространять вверх по документу за рамки элемента `<form>`, к которому оно относится. В то же время универсальные события, такие как `mousedown`, могут быть интересны любому элементу документа, поэтому они всплывают по иерархии документа, вызывая любые предусмотренные для этого обработчики событий во всех предках целевого узла. Как правило, события ввода всплывают, а высокоуровневые семантические события – нет. (Полный список всплывающих и невсплывающих событий приведен в табл. 17.3 далее в этой главе.)

Любой обработчик может остановить дальнейшее распространение события, вызвав метод `stopPropagation()` объекта `Event`, представляющего данное событие. Более подробную информацию об объекте `Event` и его методе `stopPropagation()` можно найти далее в этой главе.

Некоторые события приводят к выполнению веб-броузером связанных с ними действий, предлагаемых по умолчанию. Например, когда в теге `<a>` происходит событие `click`, действием броузера, предлагаемым по умолчанию, является переход по гиперссылке. Такие действия по умолчанию выполняются только после завершения всех трех фаз распространения события, и любой обработчик, вы-

званный во время распространения события, имеет возможность отменить действие по умолчанию, вызвав метод `preventDefault()` объекта `Event`.

Подобная схема распространения событий может показаться усложненной, но она обеспечивает важный механизм централизации кода обработки событий. Стандарт DOM Level 1 предоставляет доступ ко всем элементам документа и допускает возникновение событий (таких как события `mouseover`) в любом из них. Это значит, что имеется намного больше мест, где могут быть зарегистрированы обработчики событий, чем в старой модели обработки событий Level 0. Предположим, что вы хотите вызывать обработчик события при наведении указателя мыши на элемент `<p>` в вашем документе. Вместо регистрации обработчика события `onmouseover` для каждого тега `<p>` можно зарегистрировать один обработчик события в объекте `Document` и в ходе распространения этого события обработать его либо на этапе перехвата, либо на этапе всплывания.

И еще одна важная деталь, относящаяся к распространению событий. В модели Level 0 можно зарегистрировать только один обработчик для определенного типа события в определенном объекте. В то же время в модели Level 2 можно зарегистрировать произвольное количество функций-обработчиков для определенного типа события в определенном объекте. Это также относится к предкам целевого узла события, чья функция или функции обработки вызываются в ходе перехвата и всплывания события в документе.

17.2.2. Регистрация обработчиков событий

В API Level 0 можно зарегистрировать обработчик события, установив значение атрибута в HTML-коде или значение свойства объекта в JavaScript-коде. В модели событий Level 2 обработчик события регистрируется для определенного элемента вызовом метода `addEventListener()` этого объекта. (Хотя стандарт DOM определяет для этого прикладного интерфейса (API) термин *слушатель* (*listener*), для единообразия мы по-прежнему будем оперировать термином *обработчик* (*handler*).) Этот метод принимает три аргумента. Первый – имя типа события, для которого регистрируется обработчик. Тип события должен быть строкой, содержащей имя HTML-атрибута обработчика в нижнем регистре без начальных букв «on». Другими словами, если в модели Level 0 используется HTML-атрибут `onmousedown` или свойство `onmousedown`, то в модели событий Level 2 необходимо использовать строку `"mousedown"`.

Второй аргумент `addEventListener()` представляет собой функцию-обработчик (или слушатель), которая должна вызываться при возникновении событий указанного типа. Когда вызывается ваша функция, ей в качестве единственного аргумента передается объект `Event`. Этот объект содержит информацию о событии (например, какая кнопка мыши была нажата) и определяет методы, такие как `stopPropagation()`. Позднее мы подробнее рассмотрим интерфейс `Event` и его подынтерфейсы.

Последний аргумент метода `addEventListener()` – логическое значение. Если оно равно `true`, указанный обработчик события перехватывает события в ходе их распространения на этапе перехвата. Если аргумент равен `false`, значит, это нормальный обработчик события, который вызывается, когда событие происходит непосредственно в данном элементе или в потомке элемента, а затем всплывает обратно к данному элементу.

Например, вот как при помощи функции `addEventListener()` можно зарегистрировать обработчик события `submit` элемента `<form>`:

```
document.myform.addEventListener("submit",
                                 function(e) { return validate(e.target); }
                                 false);
```

Можно перехватить все события `mousedown`, происходящие внутри элемента `<div>` с определенным именем, вызвав функцию `addEventListener()` следующим образом:

```
var mydiv = document.getElementById("mydiv");
mydiv.addEventListener("mousedown", handleMouseDown, true);
```

В этих примерах предполагается, что функции `validate()` и `handleMouseDown()` определены в каком-либо другом месте вашего JavaScript-кода.

Обработчики событий, зарегистрированные с помощью функции `addEventListener()`, выполняются в той области видимости, в которой они определены. Они не вызываются с расширенной цепочкой областей видимости, используемой для обработчиков событий, определенных в виде HTML-атрибутов, как это было описано в разделе 17.1.6.

Обработчики событий в модели Level 2 регистрируются вызовом метода, а не установкой атрибута или свойства, поэтому можно зарегистрировать несколько обработчиков для данного типа события в данном объекте. Если вызвать функцию `addEventListener()` несколько раз для регистрации нескольких функций-обработчиков одного типа события в одном объекте, то при возникновении события этого типа в данном объекте (или при всплывании к данному объекту, или при перехвате этого события данным объектом) будут вызваны все зарегистрированные вами функции. Однако здесь важно понимать, что стандарт DOM не гарантирует порядок вызова функций-обработчиков данного объекта, поэтому не следует рассчитывать, что они будут вызваны в том порядке, в котором вы их зарегистрировали. Заметьте также, что если для одного элемента несколько раз зарегистрировать одну и ту же функцию-обработчик, то все регистрации, кроме первой, игнорируются.

Зачем может потребоваться регистрировать более одной функции-обработчика одного события в одном объекте? Это может быть очень полезно для структуризации вашего программного кода. Предположим, что вы пишете универсальный модуль JavaScript-кода, использующий событие `mouseover` в изображениях для смены изображений. Теперь предположим, что у вас есть еще один модуль, с помощью которого вы собираетесь задействовать то же событие `mouseover` для вывода дополнительной информации об изображении во всплывающей DHTML-подсказке. В случае применения API Level 0 придется объединить два модуля в один, чтобы они могли совместно использовать одно свойство `onmouseover` объекта `Image`. В то же время в API Level 2 каждый модуль может зарегистрировать нужный ему обработчик события, не зная о другом модуле и не вмешиваясь в его работу.

Пару с методом `addEventListener()` образует метод `removeEventListener()`, требующий тех же трех аргументов, но не добавляющий, а удаляющий функцию обработки события из объекта. Часто бывает полезно временно зарегистрировать обработчик события, а потом удалить его. Например, при наступлении события `mousedown` можно зарегистрировать временные перехватывающие обработчики

для событий mousemove и mouseup, чтобы видеть, перемещает ли пользователь указатель мыши. Затем, когда будет получено событие mouseup, можно отменить регистрацию этих обработчиков. При этом код удаления обработчика события может выглядеть следующим образом:

```
document.removeEventListener("mousemove", handleMouseMove, true);
document.removeEventListener("mouseup", handleMouseUp, true);
```

Методы addEventListener() и removeEventListener() определены интерфейсом EventTarget. В веб-броузерах, поддерживающих модуль Events модели DOM Level 2, этот интерфейс, реализованный для узлов Element и Document, предоставляет указанные методы регистрации.[1] В IV части книги они описываются в тех же разделах, в которых описываются узлы Document и Element, но там отсутствует описание интерфейса EventTarget.

17.2.3. Метод addEventListener() и ключевое слово this

В исходной модели событий Level 0, когда функция регистрируется как обработчик события для элемента документа, она становится методом этого элемента (как это обсуждалось ранее в разделе 17.1.5). Когда вызывается обработчик события, он вызывается как метод элемента, и внутри функции ключевое слово this ссылается на элемент, в котором произошло событие.

Стандарт DOM Level 2 написан без учета языковых особенностей и указывает, что обработчики событий – это скорее объекты, а не простые функции. В то же время привязка к JavaScript стандарта DOM делает обработчиками событий JavaScript-функции, а не JavaScript-объекты. К сожалению, привязка ничего не говорит о том, как вызывать функции-обработчики и какое значение должно принимать ключевое слово this.

Несмотря на недостатки стандартизации, все известные реализации вызывают обработчики, зарегистрированные с помощью функции getEventListener(), как если бы они были методами целевого элемента. Таким образом, когда вызывается обработчик события, ключевое слово this ссылается на объект, в котором зарегистрирован обработчик. Если вы предпочитаете не полагаться на это не вполне определенное поведение, можете воспользоваться свойством currentTarget объекта Event, который передается функции-обработчику. Далее при обсуждении объекта Event вы узнаете, что свойство currentTarget ссылается на объект, в котором был зарегистрирован обработчик события.

17.2.4. Регистрация объектов в качестве обработчиков событий

Для регистрации функций-обработчиков событий служит метод addEventListener(). В объектно-ориентированном программировании можно определять об-

[1] Если быть более точным, стандарт DOM утверждает, что все узлы в документе (включая, например, узлы Text) реализуют интерфейс EventTarget. Однако на практике веб-броузеры поддерживают возможность регистрации обработчиков событий только для узлов Element и Document, а также для объекта Window, несмотря даже на то, что он не относится к стандарту DOM.

работчики событий как методы специального объекта и затем вызывать их в этом качестве. Для Java-программистов стандарт DOM допускает именно это: в нем указано, что обработчики событий – это объекты, реализующие интерфейс EventListener и метод с именем handleEvent(). При регистрации обработчика события в Java методу addEventListener() передается объект, а не функция. Для простоты привязка DOM API к JavaScript не требует реализации интерфейса EventListener и позволяет вместо этого передавать методу addEventListener() прямые ссылки на функции.

Если в объектно-ориентированной JavaScript-программе в качестве обработчиков событий выступают объекты, то для их регистрации можно использовать следующую функцию:

```
function registerObjectEventHandler(element, eventtype, listener, captures) {
    element.addEventListener(eventtype,
                             function(event) { listener.handleEvent(event); }
                             captures);
}
```

Любой объект может быть зарегистрирован как слушатель событий, если в нем определен метод handleEvent(). Этот метод вызывается как метод объекта-слушателя, и ключевое слово this ссылается на этот объект, а не на элемент документа, сгенерировавший событие.

Хотя это не является частью спецификации DOM, броузер Firefox (и другие, построенные на базе Mozilla) допускает вместо ссылки на функцию передачу объектов-слушателей событий, определяющих метод handleEvent(), непосредственно в метод addEventListener(). Для этих броузеров специальная функция регистрации, подобная той, которую мы определяли ранее, не нужна.

17.2.5. Модули и типы событий

Как уже говорилось, стандарт DOM Level 2 имеет модульную структуру, поэтому реализация может поддерживать одни его части и не поддерживать другие. Events – один из таких модулей. Проверить, поддерживает ли броузер этот модуль, можно следующим образом:

```
document.implementation.hasFeature("Events", "2.0")
```

Однако модуль Events содержит только API для базовой инфраструктуры обработки событий. Поддержка определенных типов событий делегируется субмодулям. Каждый субмодуль предоставляет поддержку определенной категории связанных типов событий и определяет тип Event, передаваемый обработчикам событий для каждого из этих типов. Например, субмодуль с именем MouseEvents предоставляет поддержку событий mousedown, mouseup, click и событий родственных типов. Он также определяет интерфейс MouseEvent. Объект, реализующий этот интерфейс, передается функции-обработчику любого типа события, поддерживаемого данным модулем.

В табл. 17.2 перечислены все модули событий, определяемые ими интерфейсы и типы событий, которые они поддерживают. Обратите внимание: DOM Level 2 не стандартизует ни один из типов событий клавиатуры, поэтому в данном списке нет модуля событий клавиатуры. Тем не менее современные броузеры под-

держивают события клавиатуры, о чем подробнее рассказывается в этой главе далее. В табл. 17.2 и оставшейся части этой книги отсутствует описание модуля MutationEvents. Событие Mutation возбуждается при изменении структуры документа. Оно может использоваться приложениями, такими как HTML-редакторы, но обычно не реализуется броузерами и практически игнорируется веб-программистами.

Таблица 17.2. Модули, интерфейсы и типы событий

Имя модуля	Интерфейс Event	Типы событий
HTMLEvents	Event	abort, blur, change, error, focus, load, reset, resize, scroll, select, submit, unload
MouseEvents	MouseEvent	click, mousedown, mousemove, mouseout, mouseover, mouseup
UIEvents	UIEvent	DOMActivate, DOMFocusIn, DOMFocusOut
MutationEvents	MutationEvent	DOMAttrModified, DOMCharacterDataModified, DOMNodeInserted, DOMNodeInsertedIntoDocument, DOMNodeRemoved, DOMNodeRemovedFromDocument, DOMSubtreeModified

Как видно из таблицы, модули HTMLEvents и MouseEvents определяют типы событий, схожие с модулем событий уровня 0. Модуль UIEvents определяет типы событий, напоминающие события focus, blur и click, поддерживаемые элементами HTML-форм, но обобщенные таким образом, чтобы генерироваться любым элементом документа, который может получать фокус или активизироваться как-то иначе.

Как уже говорилось, когда происходит событие, его обработчику передается объект, реализующий интерфейс Event, связанный с данным типом события. Свойства этого объекта предоставляют информацию о событии, которая может быть полезна обработчику. В табл. 17.3 снова перечислены стандартные события, но на этот раз организованные по типам, а не по модулям событий. Для каждого типа событий в этой таблице указан вид объекта события, передаваемого его обработчику, а также говорится, всплывает ли событие этого типа в иерархии документа в процессе распространения события (столбец «B») и есть ли для данного события действие по умолчанию, которое может быть отменено методом preventDefault() (столбец «C»). Для событий модуля HTMLEvents в последнем, пятом, столбце таблицы указано, какие HTML-элементы могут генерировать данное событие. Для всех остальных типов событий в пятом столбце указано, какие свойства объекта события содержат существенные подробности о событии (эти свойства описаны в следующем разделе). Обратите внимание: свойства, перечисленные в этом столбце, не включают свойств, определяемых базовым интерфейсом Event и содержащих осмысленные значения для всех типов событий.

Полезно сравнить табл. 17.3 с табл. 17.1, в которой перечислены обработчики событий Level 0, определяемые в HTML 4. Типы событий, поддерживаемые двумя этими моделями, в значительной степени совпадают (исключая модуль UIEvents). Стандарт DOM Level 2 добавляет поддержку типов событий abort, error, resize и scroll, которые не были стандартизованы в HTML 4, и исключает поддержку типа события dblclick, являющегося частью стандарта HTML 4. (Вместо этого,

как мы скоро увидим, свойство `detail` объекта, передаваемого обработчику события `click`, определяет количество последовательных щелчков мыши.)

Таблица 17.3. Типы событий

Тип события	Интерфейс	В	С	Поддержка/детализирующие свойства
abort	Event	Да	Нет	<img>, <object>
blur	Event	Нет	Нет	<a>, <area>, <button>, <input>, <label>, <select>, <textarea>
change	Event	Да	Нет	<input>, <select>, <textarea>
click	MouseEvent	Да	Да	screenX, screenY, clientX, clientY, altKey, ctrlKey, shiftKey, metaKey, button, detail
error	Event	Да	Нет	<body>, <frameset>, <img>, <object>
focus	Event	Нет	Нет	<a>, <area>, <button>, <input>, <label>, <select>, <textarea>
load	Event	Нет	Нет	<body>, <frameset>, <iframe>, <img>, <object>
mousedown	MouseEvent	Да	Да	screenX, screenY, clientX, clientY, altKey, ctrlKey, shiftKey, metaKey, button, detail
mousemove	MouseEvent	Да	Нет	screenX, screenY, clientX, clientY, altKey, ctrlKey, shiftKey, metaKey
mouseout	MouseEvent	Да	Да	screenX, screenY, clientX, clientY, altKey, ctrlKey, shiftKey, metaKey, relatedTarget
mouseover	MouseEvent	Да	Да	screenX, screenY, clientX, clientY, altKey, ctrlKey, shiftKey, metaKey, relatedTarget
mouseup	MouseEvent	Да	Да	screenX, screenY, clientX, clientY, altKey, ctrlKey, shiftKey, metaKey, button, detail
reset	Event	Да	Нет	<form>
resize	Event	Да	Нет	<body>, <frameset>, <iframe>
scroll	Event	Да	Нет	<body>
select	Event	Да	Нет	<input>, <textarea>
submit	Event	Да	Да	<form>
unload	Event	Нет	Нет	<body>, <frameset>
DOMActivate	UIEvent	Да	Да	detail
DOMFocusIn	UIEvent	Да	Нет	Отсутствуют
DOMFocusOut	UIEvent	Да	Нет	Отсутствуют

17.2.6. Интерфейсы и детализирующие свойства событий

Когда происходит событие, прикладной интерфейс (API) модели DOM Level 2 предоставляет дополнительную информацию (например, где и когда оно произошло) о нем в виде свойств объекта, передаваемого обработчику события. С ка-

ждым модулем событий связан интерфейс событий, в котором содержится информация, относящаяся к этому типу событий. В табл. 17.2 представлены три различных модуля событий и три различных интерфейса событий.

Эти три интерфейса фактически связаны друг с другом и образуют иерархию. Интерфейс `Event` является вершиной иерархии; все объекты событий реализуют этот базовый интерфейс. `UIEvent` — это подынтерфейс интерфейса `Event`: любой объект события, реализующий `UIEvent`, также реализует все методы и свойства `Event`. Интерфейс `MouseEvent` является подынтерфейсом `UIEvent`. Это значит, например, что объект события, переданный обработчику события `click`, реализует все методы и свойства, определенные в каждом из интерфейсов `MouseEvent`, `UIEvent` и `Event`.

В следующих разделах представлен каждый из интерфейсов событий и выделены их наиболее важные свойства и методы. Полные описания всех интерфейсов можно найти в четвертой части книги.

17.2.6.1. Интерфейс Event

Типы событий, определенные в модуле `HTMLEvents`, используют интерфейс `Event`. Все остальные типы событий используют подынтерфейсы этого интерфейса, т. е. интерфейс `Event` реализуется всеми объектами событий и предоставляет детальную информацию, применимую ко всем типам событий. Интерфейс `Event` определяет следующие свойства (обратите внимание, что эти свойства и свойства всех подынтерфейсов интерфейса `Event` доступны только для чтения):

`type`

Тип произошедшего события. Значением этого свойства является имя типа события, и это та же строка, которая была использована при регистрации обработчика события (например, `"click"` или `"mouseover"`).

`target`

Узел документа, в котором произошло событие; может не совпадать с `currentTarget`.

`currentTarget`

Узел, в котором в данный момент обрабатывается событие (т. е. узел, чей обработчик события работает в данный момент). Если событие обрабатывается на этапах перехвата или всплывания события, значение этого свойства отличается от значения свойства `target`. Как говорилось ранее, необходимо использовать это свойство вместо ключевого слова `this` в собственных функциях обработки событий.

`eventPhase`

Число, указывающее, какой этап распространения события выполняется в данный момент. Значением является одна из констант `Event.CAPTURING_PHASE`, `Event.AT_TARGET` или `Event.BUBBLING_PHASE`.

`timeStamp`

Объект `Date`, указывающий, когда произошло событие.

`bubbles`

Логическое значение, указывающее, всплывает ли это событие (и события этого типа) вверх по дереву документов.

`cancelable`

Логическое значение, указывающее, связано ли с этим событием действие по умолчанию, которое может быть отменено методом `preventDefault()`.

В дополнение к этим семи свойствам в интерфейсе `Event` определены два метода: `stopPropagation()` и `preventDefault()`. Они также реализуются всеми объектами событий. Любой обработчик события может вызвать метод `stopPropagation()` для предотвращения распространения события за пределы узла, в котором оно обрабатывается в данный момент. Любой обработчик события может вызвать метод `preventDefault()`, чтобы предотвратить выполнение броузером действия по умолчанию, связанного с событием. Вызов `preventDefault()` в API DOM Level 2 эквивалентен возвращению обработчиком значения `false` в модели событий Level 0.

17.2.6.2. Интерфейс UIEvent

`UIEvent` является подынтерфейсом интерфейса `Event`. Он определяет тип объекта события, передаваемого событиям типа `DOMFocusIn`, `DOMFocusOut` и `DOMActivate`. Эти типы событий используются достаточно редко и, что более важно, интерфейс `UIEvent` является родительским интерфейсом для `MouseEvent`. Интерфейс `UIEvent` определяет два свойства в дополнение к свойствам, определяемым интерфейсом `Event`.

`view`

Объект `Window` (в терминологии DOM – *представление*), внутри которого произошло событие.

`detail`

Число, которое может предоставить дополнительную информацию о событии. Для событий `click`, `mousedown` и `mouseup` это поле содержит количество щелчков: 1 для одинарного щелчка, 2 – для двойного и 3 – для тройного. (Обратите внимание: каждый щелчок мыши генерирует событие, но если несколько щелчков следуют достаточно быстро, на это указывает значение `detail`. То есть событию мыши со значением свойства `detail`, равным 2, всегда предшествует событие мыши со значением свойства `detail`, равным 1.) Для событий `DOMActivate` это поле равно 1 в случае нормальной активации и 2 – в случае гиперактивации, например двойного щелчка или нажатия комбинации клавиш Shift-Enter.

17.2.6.3. Интерфейс MouseEvent

Интерфейс `MouseEvent` наследует свойства и методы интерфейсов `Event` и `UIEvent`, а также определяет следующие дополнительные свойства:

`button`

Число, указывающее, какая кнопка мыши изменила свое состояние во время события `mousedown`, `mouseup` или `click`. Значение 0 обозначает левую кнопку, 1 – среднюю кнопку, а 2 – правую кнопку. Это свойство применяется, только когда кнопка изменяет состояние, и не применяется, например, для получения информации о том, удерживается ли нажатой кнопка во время события `mousemove`. Примечательно, что Netscape 6 ведет себя неправильно, используя вместо значений 0, 1 и 2 значения 1, 2 и 3. В Netscape 6.1 эта ошибка исправлена.

`altKey`, `ctrlKey`, `metaKey`, `shiftKey`

Эти четыре логических значения указывают, нажимались ли клавиши Alt, Ctrl, Meta и Shift, когда происходило событие мыши. В отличие от свойства `button`, эти свойства клавиатуры действительны для любого типа события мыши.

`clientX`, `clientY`

Эти два свойства указывают координаты X и Y указателя мыши относительно клиентской области или окна броузера. Обратите внимание, что данные координаты не учитывают прокрутку документа: если событие происходит в верхнем крае окна, значение свойства `clientY` равно 0 независимо от того, насколько далеко был прокручен документ. К сожалению, DOM Level 2 не предоставляет стандартного способа трансляции оконных координат в координаты документа. В броузерах, не относящихся к линейке Internet Explorer, можно сложить значения `window.pageXOffset` и `window.pageYOffset` (подробности см. в разделе 14.3.1).

`screenX`, `screenY`

Эти два свойства задают координаты X и Y указателя мыши относительно верхнего левого края дисплея. Эти значения полезны, если вы планируете открыть новое окно в месте возникновения события мыши или рядом с ним.

`relatedTarget`

Это свойство ссылается на узел, который связан с целевым узлом события. Для события `mouseover` это узел, который указатель мыши оставил, перейдя к целевому узлу. Для событий `mouseout` это узел, в который переместился указатель мыши, оставив целевой узел. Для других типов событий это свойство не используется.

17.2.7. Смешанная модель обработки событий

До сих пор обсуждались традиционная модель обработки событий уровня 0, а также новая модель стандарта DOM Level 2. С целью сохранения обратной совместимости броузеры, поддерживающие модель Level 2, продолжают поддерживать и модель обработки событий Level 0. Это означает, что существует возможность использовать обе модели обработки событий в пределах одного документа.

Важно понимать, что веб-броузеры, поддерживающие модель обработки событий Level 2, всегда передают объект события обработчикам событий, даже тем из них, которые зарегистрированы установкой HTML-атрибута или JavaScript-свойства с использованием модели Level 0. Когда обработчик события определяется как HTML-атрибут, он неявно преобразуется в функцию, которая принимает аргумент с именем `event`. Это означает, что такие обработчики событий могут использовать идентификатор `event` для обращения к объекту события. (Позднее вы увидите, что указание идентификатора `event` в HTML-атрибутах допускается также при использовании модели обработки событий IE.)

Стандарт DOM учитывает то обстоятельство, что модель обработки событий уровня 0 по-прежнему остается в обиходе, а потому указывает, что реализация модели Level 0 должна трактовать зарегистрированные в этой модели обработчики так, как если бы они были зарегистрированы методом `addEventListener()`. То есть, если функция `f` присваивается свойству `onclick` элемента `e` документа

(или установкой соответствующего HTML-атрибута `onclick`), это эквивалентно следующему вызову функции регистрации:

```
e.addEventListener("click", f, false);
```

Когда вызывается функция `f`, ей в виде аргумента передается объект события, несмотря даже на то, что она была зарегистрирована с использованием модели Level 0.

17.3. Модель обработки событий Internet Explorer

Модель обработки событий, поддерживаемая Internet Explorer версий 4, 5, 5.5 и 6, является переходной, находящейся посередине между исходной моделью Level 0 и стандартной моделью DOM Level 2. Модель обработки событий IE включает объект `Event`, предоставляющий информацию о произошедшем событии. Однако вместо передачи функциям обработки событий объект `Event` сделан доступным в виде свойства объекта `Window`. Модель Internet Explorer поддерживает всплывание при распространении события, но не поддерживает перехват, как модель DOM (хотя в IE 5 и более поздних версиях предусматривается специальная возможность перехвата событий мыши). В броузере IE 4 обработчики событий регистрируются так же, как в исходной модели Level 0. Однако в IE 5 и более поздних версиях с помощью специальных (и нестандартных) функций может регистрироваться несколько обработчиков.

В следующих разделах эта модель обработки событий представлена более подробно в сравнении с исходной моделью уровня 0 и стандартной моделью уровня 2. Следовательно, прежде чем читать описание модели IE, вы должны убедиться, что разобрались в этих двух моделях.

17.3.1. Объект Event в IE

Как и стандартная модель DOM Level 2, модель обработки событий IE предоставляет подробную информацию о каждом произошедшем событии в виде свойств объекта `Event`. Объекты `Event`, определенные в стандартной модели обработки событий, на самом деле разработаны на основе объекта `Event` из IE, поэтому можно заметить много общего между свойствами объекта `Event` в IE и свойствами объектов `Event`, `UIEvent` и `MouseEvent` в DOM.

Далее приводится перечень наиболее важных свойств объекта `Event` в IE:

`type`

Строка, указывающая тип произошедшего события. Значение этого свойства совпадает с именем обработчика события без начальных символов «on» (например, `"click"` или `"mouseover"`). Свойство совместимо со свойством `type` в модели обработки событий DOM.

`srcElement`

Элемент документа, в котором произошло событие. Сравнимо со свойством `target` объекта `Event` в DOM.

`button`

Целое число, обозначающее нажатую кнопку мыши. Значение 1 обозначает левую кнопку, 2 – правую кнопку, а 4 – среднюю кнопку мыши. Если нажато

несколько кнопок, эти значения складываются вместе, например значение 3 соответствует левой и правой кнопкам, нажатым вместе. Сравните это со свойством `button` объекта `MouseEvent` в DOM Level 2, но обратите внимание, что хотя имена свойств совпадают, интерпретация их значений разная.

`clientX`, `clientY`

Данные целочисленные свойства совместимы с одноименными свойствами `MouseEvent` в DOM Level 2 и указывают координаты указателя мыши в момент события относительно левого верхнего угла окна. Для документов, имеющих больший размер, чем окно, данные координаты не совпадают с позицией в документе. Для преобразования этих оконных координат в координаты документа может потребоваться прибавить к ним величину прокрутки документа. Информацию о том, как это сделать, вы найдете в разделе 14.3.1.

`offsetX`, `offsetY`

Эти целочисленные свойства указывают положение указателя мыши относительно исходного элемента. Они позволяют, например, определить, на каком пикселе объекта `Image` был выполнен щелчок. Эти свойства не имеют эквивалента в модели обработки событий DOM Level 2.

`altKey`, `ctrlKey`, `shiftKey`

Эти логические свойства указывают, были ли нажаты клавиши Alt, Ctrl и Shift при возникновении события. Эти свойства с совместимы с одноименными свойствами объекта `MouseEvent` в DOM Level 2. При этом следует отметить, что у объекта `Event` в IE нет свойства `metaKey`.

`keyCode`

Целочисленное свойство. Задает код клавиши для событий `keydown` и `keyup`, а также код Unicode-символа для события `keypress`. Коды символов преобразуются в строки методом `String.fromCharCode()`. События клавиатуры описываются более подробно в этой главе далее.

`fromElement`, `toElement`

Свойство `fromElement` указывает для события `mouseover` элемент документа, в котором находился указатель мыши. Свойство `toElement` указывает для события `mouseout` элемент документа, в который переместился указатель мыши. Свойства сравнимы со свойством `relatedTarget` объекта `MouseEvent` в DOM Level 2.

`cancelBubble`

Логическое свойство. Будучи установленным в `true`, предотвращает дальнейшее всплывание события вверх по иерархии включения элементов. Сравнимо с методом `stopPropagation()` объекта `Event` в DOM Level 2.

`returnValue`

Логическое свойство, которое может быть установлено в `false` для предотвращения выполнения пользователем действия по умолчанию, связанного с событием. Это – альтернатива традиционному приему возвращения значения `false` обработчиком события. Сравнимо с методом `preventDefault()` объекта `Event` в DOM Level 2.

Полное описание объекта `Event` в IE приведено в четвертой части книги.

17.3.2. Объект Event в IE как глобальная переменная

Несмотря на то что модель обработки событий в IE предоставляет информацию о событии в объекте Event, этот объект передается только обработчикам событий, зарегистрированным с помощью нестандартного метода attachEvent() (который будет описан позже). Остальные обработчики событий вызываются без аргументов. Однако получить доступ к объекту Event из обработчиков событий в IE можно с помощью свойства event глобального объекта Window. Это значит, что функция обработки события в IE может обращаться к объекту Event как к window.event или просто как к event. Использование глобальной переменной там, где годится аргумент функции, может показаться странным, но схема IE работает, т. к. в программной модели обработки событий неявно предполагается, что одновременно всегда обрабатывается только одно событие. Поскольку два события никогда не обрабатываются одновременно, можно спокойно сохранять информацию о текущем обрабатываемом событии в глобальной переменной.

Объект Event представляет собой глобальную переменную, и это несовместимо со стандартной моделью DOM Level 2. Обойти это препятствие можно с помощью одной строки кода. Чтобы функция обработки события работала в обеих объектных моделях, напишите ее так, чтобы она ожидала аргумент, а затем, если аргумент не передан, инициализируйте этот аргумент из глобальной переменной. Например:

```
function portableEventHandler(e) {
    if (!e) e = window.event; // Получение информации о событии в IE
    // Тело обработчика событий
}
```

Другой распространенный прием основан на использовании оператора || для возвращения первого определенного аргумента:

```
function portableEventHandler(event) {
    var e = event || window.event;
    // Тело обработчика событий
}
```

17.3.3. Регистрация обработчика события в IE

В IE 4 обработчики событий регистрируются так же, как и в исходной модели обработки событий Level 0: указанием их в виде HTML-атрибутов или присваиванием функций свойствам-обработчикам событий элементов документа.

IE 5 и более поздние версии предоставляют методы attachEvent() и detachEvent(), которые реализуют возможность регистрации более одной функции-обработчика для события заданного типа в заданном объекте. При вызове обработчику события, зарегистрированному с помощью метода attachEvent(), в качестве аргумента будет передана копия глобального объекта window.event.

```
function highlight() { /* Тело обработчика события */ }
document.getElementById("myelt").attachEvent("onmouseover", highlight);
```

Методы attachEvent() и detachEvent() работают аналогично методам addEventListener() и removeEventListener() со следующими исключениями:

- Поскольку модель обработки событий IE не поддерживает возможность перехвата событий, методы `attachEvent()` и `detachEvent()` ожидают всего два аргумента: тип события и функцию-обработчик.
- Имена обработчиков событий, передаваемых методу в IE, должны включать префикс «on». Например, методу `attachEvent()` следует передавать строку `"onclick"`, а не `"click"`, как методу `addEventListener()`.
- Функции, регистрируемые с помощью `attachEvent()`, вызываются как глобальные функции, а не как методы элемента документа, в котором произошло событие. То есть когда исполняется обработчик события, зарегистрированный с помощью `attachEvent()`, ключевое слово `this` ссылается на объект `Window`, а не на целевой элемент события.
- Метод `atachEvent()` позволяет зарегистрировать несколько раз функцию-обработчик события с тем же именем. Когда произойдет событие заданного типа, функция-обработчик вызывается столько раз, сколько она была зарегистрирована.

17.3.4. Всплывание события в IE

В модели обработки событий IE, в отличие от модели DOM Level 2, нет понятия перехвата события. Однако так же, как в модели Level 2, в модели IE события всплывают по иерархии включения. Как и в модели Level 2, всплывание события относится только к необработанным событиям, или событиям ввода (в основном к событиям мыши и клавиатуры), но не к высокоуровневым семантическим событиям. Основное различие между всплыванием события в IE и в DOM Level 2 состоит в способе прекращения всплывания. В отличие от объекта `Event` в DOM, в объекте `Event` в IE нет метода `stopPropagation()`. Чтобы предотвратить или остановить всплывание события далее вверх по иерархии включения, обработчик события в IE должен установить свойство `cancelBubble` объекта `Event` в `true`:

```
window.event.cancelBubble = true;
```

Обратите внимание: свойство `cancelBubble` применяется только к текущему событию. Когда генерируется новое событие, `window.event` присваивается новое событие `Event`, а `cancelBubble` принимает значение по умолчанию — `false`.

17.3.5. Перехват событий мыши

При реализации любого пользовательского интерфейса, поддерживающего операции перетаскивания мышью (например, раскрывающегося меню), очень важно иметь возможность перехватывать события мыши, чтобы можно было корректно обрабатывать перемещение указателя мыши независимо от того, какой объект перетаскивает пользователь. В модели обработки событий DOM это можно реализовать с помощью перехватывающих обработчиков событий. В IE 5 и более поздних версиях аналогичная операция выполняется с помощью методов `setCapture()` и `releaseCapture()`.

Методы `setCapture()` и `releaseCapture()` имеются у всех HTML-элементов. Когда вызывается метод `setCapture()` некоторого элемента, все последующие события мыши направляются этому элементу, а обработчик события сможет обработать их, прежде чем они начнут всплытие. Примечательно, что это относится только к событиям мыши `mousedown`, `mouseup`, `mousemove`, `mouseover`, `mouseout`, `click` и `dblclick`.

После вызова setCapture() события мыши распространяются по специальному маршруту, пока не будет вызван метод releaseCapture() или пока перехват событий не прервется. Перехват событий мыши может быть прерван в результате потери броузером фокуса ввода, вызова диалогового окна методом alert(), отображения системного меню и в других подобных случаях. Если происходит один из таких случаев, элемент, в контексте которого был вызван метод setCapture(), получает событие onlosecapture, извещающее о том, что он больше не будет получать перехватываемые события мыши.

В большинстве случаев метод setCapture() вызывается в ответ на событие mousedown, что гарантирует получение событий мыши тем же самым элементом. Элемент выполняет действия по перетаскиванию в ответ на событие mousemove и вызывает releaseCapture() в ответ на (перехваченное) событие mouseup.

Использование методов setCapture() и relaseCapture() иллюстрирует пример 17.4 далее в этой главе.

17.3.6. Метод attachEvent() и ключевое слово this

Как уже отмечалось ранее, обработчики событий, зарегистрированные методом attachEvent(), вызываются как глобальные функции, а не как методы элемента, в котором они зарегистрированы. Это означает, что внутри таких обработчиков событий ключевое слово this ссылается на глобальный объект Window. Само по себе это не представляет собой большую проблему, однако дело осложняется тем, что в объект события в IE не имеет эквивалентного DOM-свойства currentTarget. Свойство srcElement указывает на элемент, сгенерировавший событие, но если событие уже начало всплытие, им может быть другой элемент, отличный от элемента, обрабатывающего событие.

Если необходимо написать универсальный обработчик события (который может быть зарегистрирован в любом элементе) и при этом знать, какой элемент его зарегистрировал, нельзя регистрировать обработчик методом attachEvent(). Нужно либо регистрировать обработчик с помощью модели обработки событий Level 0, либо определить функцию-обертку, которую и зарегистрировать:

```
// Это обработчик события и элемент, который его регистрирует
function genericHandler() {
    /* Программный код, использующий ключевое слово this */
}
var element = document.getElementById("myelement");

// Обработчик может быть зарегистрирован с использованием API уровня 0
element.onmouseover = genericHandler;

// Или можно создать замыкание
element.attachEvent("onmouseover", function() {
                        // Вызов обработчика как метода элемента
                        genericHandler.call(element, event);
                    });
```

Проблема с прикладным интерфейсом (API) Level 0 заключается в том, что он не позволяет регистрировать несколько функций-обработчиков, а проблема с замыканиями связана с утечками памяти в IE. Более подробные сведения об этом приводятся в следующем разделе.

17.3.7. Обработчики событий и утечки памяти

Как уже говорилось в пункте 8.8.4.2, Internet Explorer (вплоть до версии 6) страдает утечками памяти, связанными с тем, что в качестве обработчиков событий используются вложенные функции. Рассмотрим следующий фрагмент:

```
// Добавить обработчик, проверяющий правильность заполнения формы
function addValidationHandler(form) {
    form.attachEvent("onsubmit", function( ) { return validate(); });
}
```

Когда вызывается эта функция, она добавляет обработчик события в заданный элемент формы. Обработчик события определен как вложенная функция, и хотя сама функция не ссылается ни на один из элементов формы, ссылки на них оказываются захваченными в ее области видимости, как часть замыкания. В результате элемент формы ссылается на JavaScript-объект `Function`, а объект (через цепочку области видимости) – обратно, на элемент формы. Такого рода циклические ссылки могут вызывать в IE утечки памяти.

Одно из решений этой проблемы заключается в том, чтобы старательно избегать вложенных функций при программировании для IE. Другое решение – тщательно удалить все обработчики событий в ответ на событие `onunload()`. В примере, который приводится в следующем разделе, используется второй вариант.

17.3.8. Пример: модель обработки событий, совместимая с IE

В этом разделе основное внимание уделяется ряду несовместимостей между моделями обработки событий в IE и в DOM Level 2. В примере 17.2 представлен модуль, который преодолевает многие из этих несовместимостей. В модуле определены две функции, `Handler.add()` и `Handler.remove()`, которые добавляют и удаляют обработчики событий для заданного элемента. На платформах, поддерживающих метод `addEventListener()`, эти функции являются тривиальными обертками вокруг стандартных методов. Однако в IE 5 и более поздних версиях эти методы определены так, что преодолевают следующие несовместимости:

- Обработчики событий вызываются как методы зарегистрировавшего их элемента.
- Обработчикам событий для расширения возможностей передается смоделированный объект события, соответствующий стандарту DOM.
- Повторные попытки регистрации обработчиков событий игнорируются.
- С целью предотвращения утечек памяти в IE при выгрузке документа отменяется регистрация всех обработчиков событий.

Чтобы обработчики событий вызывались с корректным значением ключевого слова `this`, а также для того, чтобы передавать им смоделированный объект события, в примере 17.2 функции-обработчики должны быть обернуты в другие функции, корректно вызывающие обработчиков. Самая интересная часть в этом примере – это программный код, который отображает функцию-обработчик, передаваемую методу `Handler.add()`, на функцию-обертку, фактически регистрируемую методом `attachEvent()`. Такого рода отображение должно выполняться

так, чтобы метод `Handler.remove()` **мог удалить правильную функцию-обертку при удалении обработчиков в ходе выгрузки документа.**

Пример 17.2. Код обеспечения совместимости с моделью события IE

```
/*
 * Handler.js -- Переносимые функции регистрации и отмены регистрации
 *
 * Этот модуль определяет функции регистрации и отмены регистрации
 * обработчиков событий Handler.add() и Handler.remove(). Обе функции
 * принимают три аргумента:
 *
 *   element: DOM-элемент, документ или окно, куда добавляется
 *     или откуда удаляется обработчик события.
 *
 *   eventType: строка, определяющая тип события, для обработки которого
 *     вызывается обработчик. Используются имена типов в соответствии
 *     со стандартом DOM, в которых отсутствует префикс "on".
 *     Примеры: "click", "load", "mouseover".
 *
 *   handler: Функция, которая вызывается при возникновении указанного
 *     события в заданном элементе. Данная функция вызывается как метод
 *     элемента, в котором она зарегистрирована, и ключевое слово "this"
 *     будет ссылаться на этот элемент. Функции-обработчику в качестве
 *     аргумента передается объект события. Данный объект будет либо
 *     стандартным объектом Event, либо смоделированным объектом.
 *     В случае передачи смоделированного объекта он будет обладать
 *     следующими свойствами, совместимыми со стандартом DOM: type, target,
 *     currentTarget, relatedTarget, eventPhase, clientX, clientY, screenX,
 *     screenY, altKey, ctrlKey, shiftKey, charCode, stopPropagation()
 *     и preventDefault()
 *
 * Функции Handler.add() и Handler.remove() не имеют возвращаемых значений.
 *
 * Handler.add() игнорирует повторные попытки зарегистрировать один и тот
 * же обработчик события для одного и того же типа события и элемента.
 * Handler.remove() не выполняет никаких действий, если вызывается
 * для удаления незарегистрированного обработчика.
 *
 * Примечания к реализации:
 *
 * В броузерах, поддерживающих стандартные функции регистрации
 * addEventListener() и removeEventListener(), Handler.add()
 * и Handler.remove() просто вызывают эти функции, передавая значение false
 * в третьем аргументе (это означает, что обработчики событий никогда
 * не будут зарегистрированы как перехватывающие обработчики событий).
 *
 * В версиях Internet Explorer, поддерживающих attachEvent(), функции
 * Handler.add() и Handler.remove() используют методы attachEvent()
 * и detachEvent(). Для вызова функций-обработчиков с корректным значением
 * ключевого слова this используются замыкания.
 * Поскольку в Internet Explorer замыкания могут приводить к утечкам памяти,
 * Handler.add() автоматически регистрирует обработчик события onunload,
 * в котором отменяется регистрация всех обработчиков при выгрузке страницы.
```

```
 * Для хранения информации о зарегистрированных обработчиках функция
 * Handler.add() создает в объекте Window свойство с именем _allHandlers,
 * а во всех элементах, для которых регистрируются обработчики, создается
 * свойство с именем _handlers.
 */
var Handler = {};

// В DOM-совместимых броузерах наши функции являются тривиальными
// обертками вокруг addEventListener() и removeEventListener().
if (document.addEventListener) {
    Handler.add = function(element, eventType, handler) {
        element.addEventListener(eventType, handler, false);
    };

    Handler.remove = function(element, eventType, handler) {
        element.removeEventListener(eventType, handler, false);
    };
}

// В IE 5 и более поздних версиях используются attachEvent() и detachEvent()
// с применением некоторых приемов, делающих их совместимыми
// с addEventListener и removeEventListener.
else if (document.attachEvent) {
    Handler.add = function(element, eventType, handler) {
        // Не допускать повторную регистрацию обработчика
        // _find() - частная вспомогательная функция определена далее.
        if (Handler._find(element, eventType, handler) != -1) return;

        // Эта вложенная функция определяется для того, чтобы иметь
        // возможность вызвать функцию как метод элемента.
        // Эта же функция регистрируется вместо фактического обработчика событий.
        var wrappedHandler = function(e) {
            if (!e) e = window.event;

            // Создать искусственный объект события, отчасти
            // совместимый со стандартом DOM.
            var event = {
                _event: e,    // Если потребуется настоящий объект события IE
                type: e.type, // Тип события
                target: e.srcElement,   // Где возникло событие
                currentTarget: element, // Где обрабатывается
                relatedTarget: e.fromElement?e.fromElement:e.toElement,
                eventPhase: (e.srcElement==element)?2:3,

                // Координаты указателя мыши
                clientX: e.clientX, clientY: e.clientY,
                screenX: e.screenX, screenY: e.screenY,

                // Состояние клавиш
                altKey: e.altKey, ctrlKey: e.ctrlKey,
                shiftKey: e.shiftKey, charCode: e.keyCode,

                // Функции управления событиями
                stopPropagation: function(){this._event.cancelBubble = true;},
                preventDefault: function() {this._event.returnValue = false;}
            }
```

```
    // Вызвать функцию-обработчик как метод элемента, передать
    // искусственный объект события как единственный аргумент.
    // Если функция Function.call() определена - использовать ее,
    // иначе применить маленький трюк
    if (Function.prototype.call)
        handler.call(element, event);
    else {
        // Если функция Function.call отсутствует,
        // симулировать ее вызов.
        element._currentHandler = handler;
        element._currentHandler(event);
        element._currentHandler = null;
    }
};

// Зарегистрировать вложенную функцию как обработчик события.
element.attachEvent("on" + eventType, wrappedHandler);

// Теперь необходимо сохранить информацию о пользовательской
// функции-обработчике и вложенной функции, которая вызывает этот
// обработчик. Делается это для того, чтобы можно было
// отменить регистрацию обработчика методом remove()
// или автоматически при выгрузке страницы.

// Сохранить всю информацию об обработчике в объекте.
var h = {
    element: element,
    eventType: eventType,
    handler: handler,
    wrappedHandler: wrappedHandler
};

// Определить документ, частью которого является обработчик.
// Если элемент не имеет свойства "document" - это не окно
// и не элемент документа, следовательно, это должен быть
// сам объект document.
var d = element.document || element;

// Теперь получить ссылку на объект window, связанный с этим документом.
var w = d.parentWindow;

// Необходимо связать этот обработчик с окном,
// чтобы можно было удалить его при выгрузке окна.
var id = Handler._uid(); // Сгенерировать уникальное имя свойства
if (!w._allHandlers) w._allHandlers = {}; // Создать объект, если необходимо
w._allHandlers[id] = h; // Сохранить обработчик в этом объекте

// И связать идентификатор информации об обработчике с этим элементом.
if (!element._handlers) element._handlers = [];
element._handlers.push(id);

// Если связанный с окном обработчик события onunload
// еще не зарегистрирован, зарегистрировать его.
if (!w._onunloadHandlerRegistered) {
    w._onunloadHandlerRegistered = true;
    w.attachEvent("onunload", Handler._removeAllHandlers);
}
```

```
};

Handler.remove = function(element, eventType, handler) {
    // Отыскать заданный обработчик в массиве element._handlers[].
    var i = Handler._find(element, eventType, handler);
    if (i == -1) return; // Если нет зарегистрированных обработчиков,
                         // ничего не делать

    // Получить ссылку на окно для данного элемента.
    var d = element.document || element;
    var w = d.parentWindow;

    // Отыскать уникальный идентификатор обработчика.
    var handlerId = element._handlers[i];
    // И использовать его для поиска информации об обработчике.
    var h = w._allHandlers[handlerId];
    // Используя эту информацию, отключить обработчик от элемента.
    element.detachEvent("on" + eventType, h.wrappedHandler);
    // Удалить один элемент из массива element._handlers.
    element._handlers.splice(i, 1);
    // И удалить информацию об обработчике из объекта _allHandlers.
    delete w._allHandlers[handlerId];
};

// Вспомогательная функция поиска обработчика в массиве element._handlers
// Возвращает индекс в массиве или -1, если требуемый обработчик не найден
Handler._find = function(element, eventType, handler) {
    var handlers = element._handlers;
    if (!handlers) return -1; // Если нет зарегистрированных обработчиков,
                              // ничего не делать
        // Получить ссылку на окно для данного элемента
    var d = element.document || element;
    var w = d.parentWindow;

    // Обойти в цикле обработчики, связанные с этим элементом, отыскать
    // обработчик с требуемыми типом и функцией. Обход идет в обратном порядке,
    // потому что отмена регистрации обработчиков, скорее всего,
    // будет выполняться в порядке, обратном их регистрации.
    for(var i = handlers.length-1; i >= 0; i--) {
        var handlerId = handlers[i]; // Получить идентификатор обработчика
        var h = w._allHandlers[handlerId]; // Получить информацию
       // Если тип события и функция совпадают, значит, требуемый обработчик найден.
        if (h.eventType == eventType && h.handler == handler)
            return i;
    }
    return -1; // Совпадений не найдено
};

Handler._removeAllHandlers = function() {
    // Данная функция регистрируется как обработчик события onunload
    // с помощью attachEvent. Это означает, что ключевое слово this
    // ссылается на объект window, в котором возникло это событие.
    var w = this;

    // Обойти все зарегистрированные обработчики
    for(id in w._allHandlers) {
```

```
            // Получить информацию об обработчике по идентификатору
            var h = w._allHandlers[id];
            // Использовать ее для отключения обработчика
            h.element.detachEvent("on" + h.eventType, h.wrappedHandler);
            // Удалить информацию из объекта window
            delete w._allHandlers[id];
        }
    }

    // Частная вспомогательная функция для генерации уникальных
    // идентификаторов обработчиков
    Handler._counter = 0;
    Handler._uid = function() { return "h" + Handler._counter++; };
}
```

17.4. События мыши

Теперь, после знакомства с тремя моделями обработки событий, можно перейти к рассмотрению практических примеров программного кода, выполняющего обработку событий. В этом разделе подробно обсуждаются события мыши.

17.4.1. Преобразование координат указателя мыши

Когда возникает событие мыши, свойства `clientX` и `clientY` объекта события хранят координаты указателя мыши. Эти координаты являются координатами в окне, т. е. измеряются относительно верхнего левого угла клиентской области окна броузера и не учитывают прокрутку документа. Зачастую возникает необходимость преобразовать эти значения в координаты документа, например, чтобы отобразить всплывающую подсказку рядом с указателем мыши, а для определения координат всплывающего окна необходимо иметь координаты указателя в документе. Пример 17.3 продолжает пример 16.4. В примере 16.4 просто выводилось окно с всплывающей подсказкой в заданных координатах документа. Данный пример расширяет возможности прошлого примера за счет добавления метода `Tooltip.schedule()`, который отображает всплывающую подсказку с учетом координат, полученных от объекта события мыши. Поскольку событие мыши поставляет оконные координаты, метод `schedule()` преобразует их в координаты документа с помощью методов модуля `Geometry`, приводившегося в примере 14.2.

Пример 17.3. Позиционирование всплывающих подсказок по событиям мыши

```
// Следующие значения используются методом schedule(), определенным далее.
// Они используются как константы, но доступны для записи, поэтому вы можете
// переопределить эти значения, предлагаемые по умолчанию.
Tooltip.X_OFFSET = 25; // пикселов вправо от указателя мыши
Tooltip.Y_OFFSET = 15; // пикселов вниз от указателя мыши
Tooltip.DELAY = 500;   // миллисекунд после события mouseover

/**
 * Данный метод планирует появление всплывающей подсказки над указанным
 * элементом через Tooltip.DELAY миллисекунд от момента события.
 * Аргумент "e" должен быть объектом события mouseover. Данный метод извлекает
 * координаты мыши из объекта события, преобразует их из оконных координат
 * в координаты документа и добавляет вышеуказанные смещения.
```

```
 * Определяет текст подсказки, обращаясь к атрибуту "tooltip" заданного
 * элемента. Данный метод автоматически регистрирует обработчик события
 * onmouseout и отменяет его регистрацию. Этот обработчик выполняет скрытие
 * подсказки или отменяет ее запланированное появление.
 */
Tooltip.prototype.schedule = function(target, e) {
    // Получить текст для отображения. Если текст отсутствует - ничего не делать.
    var text = target.getAttribute("tooltip");
    if (!text) return;

    // Объект события хранит оконные координаты указателя мыши.
    // Поэтому они преобразуются в координаты документа с помощью модуля Geometry.
    var x = e.clientX + Geometry.getHorizontalScroll();
    var y = e.clientY + Geometry.getVerticalScroll();

    // Добавить смещения, чтобы подсказка появилась правее и ниже указателя мыши.
    x += Tooltip.X_OFFSET;
    y += Tooltip.Y_OFFSET;

    // Запланировать появление подсказки.
    var self = this; // Это необходимо для вложенных функций
    var timer = window.setTimeout(function() { self.show(text, x, y); },
                                  Tooltip.DELAY);

    // Зарегистрировать обработчик onmouseout, чтобы скрыть подсказку
    // или отменить появление запланированной подсказки.
    if (target.addEventListener) target.addEventListener("mouseout", mouseout,
                                                         false);
    else if (target.attachEvent) target.attachEvent("onmouseout", mouseout);
    else target.onmouseout = mouseout;

    // Реализация слушателя события приводится далее
    function mouseout() {
        self.hide(); // Скрыть подсказку, если она уже на экране,
        window.clearTimeout(timer); // отменить все запланированные подсказки
        // и удалить себя, т.к. обработчик запускается единожды
        if (target.removeEventListener)
            target.removeEventListener("mouseout", mouseout, false);
        else if (target.detachEvent)
            target.detachEvent("onmouseout",mouseout);
        else target.onmouseout = null;
    }
}

// Определить единственный глобальный объект Tooltip для общего пользования
Tooltip.tooltip = new Tooltip();

/*
 * Следующая статическая версия метода schedule() использует
 * глобальный объект tooltip
 * Используется метод следующим образом:
 *
 * <a href="www.davidflanagan.com" tooltip="good Java/JavaScript blog"
 * onmouseover="Tooltip.schedule(this, event)">David Flanagan's blog</a>
 */
Tooltip.schedule = function(target, e) {Tooltip.tooltip.schedule(target, e); }
```

17.4.2. Пример: перетаскивание элементов документа

Итак, мы обсудили вопросы распространения событий, регистрации обработчиков и различных интерфейсов объектов событий для моделей DOM Level 2 и IE, и можем, наконец, на практическом примере показать, как это все работает. В примере 19.4 представлена JavaScript-функция `drag()`, которая, будучи вызвана из обработчика события `mousedown`, позволяет пользователю перетащить элемент документа. Функция `drag()` может работать как в модели DOM, так и в модели IE.

Функция `drag()` принимает два аргумента. Первый – перетаскиваемый элемент. Это может быть элемент, в котором произошло событие `mousedown`, или содержащий его элемент (например, можно разрешить пользователю, перетаскивая заголовок окна, переместить все окно). Однако в обоих случаях он должен ссылаться на элемент документа, абсолютно позиционируемый с помощью CSS-атрибута `position`. Второй аргумент – это объект события, связанный с вызываемым событием `mousedown`.

Функция `drag()` записывает позицию, в которой произошло событие `mousedown`, и затем регистрирует обработчики для событий `mousemove` и `mouseup`, следующих за событием `mousedown`. Обработчик события `mousemove` отвечает за перемещение элемента документа, а обработчик события `mouseup` – за отмену регистрации себя и обработчика `mousemove`. Важно отметить, что обработчики `mousemove` и `mouseup` зарегистрированы как перехватывающие, т. к. пользователь может двигать мышь быстрее, чем элемент документа будет успевать за ней следовать, и некоторые из этих событий могут происходить вне исходного элемента документа. Кроме того, обратите внимание, что функции `moveHandler()` и `upHandler()`, регистрируемые для обработки этих событий, определены как вложенные внутри функции `drag()` и поэтому могут использовать ее аргументы и локальные переменные, что значительно упрощает их реализацию.

Пример 17.4. Перетаскивание элементов документа

```
/**
 * Drag.js: перетаскивание абсолютно позиционируемых HTML-элементов.
 *
 * Данный модуль определяет единственную функцию drag(),
 * которая предназначена для вызова из обработчика события onmousedown.
 * Последующие события mousemove будут вызывать перемещение заданного элемента.
 * Событие mouseup завершит операцию перетаскивания.
 * Если элемент перемещается за пределы экрана, окно прокручиваться не будет.
 * Данная реализация работает в обеих моделях: DOM уровня 2 и IE.
 *
 * Аргументы:
 *
 *   elementToDrag: элемент, получивший событие mousedown или содержащий
 *     его контейнерный элемент. Он должен позиционироваться в абсолютных
 *     координатах. Значения его свойств style.left и style.top будут
 *     изменяться по мере перетаскивания элемента пользователем.
 *
 *   event: объект Event события mousedown.
 **/
function drag(elementToDrag, event) {
```

```
    // Координаты мыши (в оконных координатах)
    // в точке, откуда начинается перемещение элемента
    var startX = event.clientX, startY = event.clientY;

    // Начальная позиция (в координатах документа) перетаскиваемого элемента.
    // Поскольку elementToDrag позиционируется в абсолютных
    // координатах, предполагается, что его свойство offsetParent
    // ссылается на элемент body документа.
    var origX = elementToDrag.offsetLeft, origY = elementToDrag.offsetTop;

    // Несмотря на то, что координаты исчисляются в различных системах
    // координат, мы можем вычислить разницу между ними и использовать
    // ее в функции moveHandler(). Этот прием будет работать,
    // потому что при перетаскивании документ не прокручивается.
    var deltaX = startX - origX, deltaY = startY - origY;

    // Зарегистрировать обработчики событий mousemove и mouseup,
    // которые последуют вслед за событием mousedown.
    if (document.addEventListener) { // Модель событий DOM уровня 2
        // Зарегистрировать перехватывающие обработчики событий
        document.addEventListener("mousemove", moveHandler, true);
        document.addEventListener("mouseup", upHandler, true);
    }
    else if (document.attachEvent) { // Модель событий IE 5+

    // В модели обработки событий IE перехват событий производится
    // вызовом метода setCapture() элемента, выполняющего перехват.
    elementToDrag.setCapture();
    elementToDrag.attachEvent("onmousemove", moveHandler);
    elementToDrag.attachEvent("onmouseup", upHandler);

    // Интерпретировать событие потери перехвата как событие mouseup.
    elementToDrag.attachEvent("onlosecapture", upHandler);
}
else { // Модель событий IE 4
    // В IE 4 мы не можем использовать attachEvent() или setCapture(),
    // поэтому вставляем обработчики событий непосредственно в объект документа
    // и уповаем на то, что требуемые события мыши всплывут
    var oldmovehandler = document.onmousemove; // Используется в upHandler()
    var olduphandler = document.onmouseup;
    document.onmousemove = moveHandler;
    document.onmouseup = upHandler;
}

    // Событие обработано, необходимо прервать его дальнейшее распространение.
    if (event.stopPropagation) event.stopPropagation( ); // DOM уровня 2
    else event.cancelBubble = true; // IE

    // Теперь необходимо предотвратить выполнение действия по умолчанию.
    if (event.preventDefault) event.preventDefault( ); // DOM уровня 2
    else event.returnValue = false; // IE

/**
 * Следующий обработчик перехватывает события mousemove в процессе
 * перетаскивания элемента. Он отвечает за перемещение элемента.
 **/
function moveHandler(e) {
    if (!e) e = window.event; // Модель событий IE
```

```
    // Переместить элемент в текущие координаты указателя мыши, при необходимости
    // подстроить его позицию на смещение начального щелчка.
    elementToDrag.style.left = (e.clientX - deltaX) + "px";
    elementToDrag.style.top = (e.clientY - deltaY) + "px";

    // И прервать дальнейшее распространение события.
    if (e.stopPropagation) e.stopPropagation( ); // DOM уровня 2
    else e.cancelBubble = true; // IE
}

/**
 * Этот обработчик перехватывает заключительное событие mouseup,
 * которое возникает в конце операции перетаскивания.
 **/
function upHandler(e) {
    if (!e) e = window.event; // Модель событий IE

        // Отменить регистрацию перехватывающих обработчиков событий.
        if (document.removeEventListener) { // Модель событий DOM
            document.removeEventListener("mouseup", upHandler, true);
            document.removeEventListener("mousemove", moveHandler, true);
        }
        else if (document.detachEvent) { // Модель событий IE 5+
            elementToDrag.detachEvent("onlosecapture", upHandler);
            elementToDrag.detachEvent("onmouseup", upHandler);
            elementToDrag.detachEvent("onmousemove", moveHandler);
            elementToDrag.releaseCapture();
        }
        else { // Модель событий IE 4
            // Восстановить первоначальные обработчики, если они были
            document.onmouseup = olduphandler;
            document.onmousemove = oldmovehandler;
        }

        // И прервать дальнейшее распространение события.
        if (e.stopPropagation) e.stopPropagation( ); // DOM уровня 2
        else e.cancelBubble = true; // IE
    }
}
```

Следующий фрагмент демонстрирует, как можно использовать функцию `drag()` **в HTML-документе (это упрощенная версия примера 16.3, куда была добавлена возможность перетаскивания элементов).**

```
<script src="Drag.js"></script> <!-- Подключить сценарий Drag.js -->
<!-- Определить перетаскиваемый элемент -->
<div style="position:absolute; left:100px; top:100px; width:250px;
background-color: white; border: solid black;">
<!-- Добавить "рукоятку", за которую перетаскивается этот элемент. -->
<!-- Обратите внимание на атрибут onmousedown. -->
<div style="background-color: gray; border-bottom: dotted black;
            padding: 3px; font-family: sans-serif; font-weight: bold;"
     onmousedown="drag(this.parentNode, event);">
Перетащи меня <!-- Содержимое "заголовка" -->
</div>
<!-- Содержимое перетаскиваемого элемента -->
```

```
<p>Это тест. Тестирование, тестирование и еще раз тестирование.
<p>Это тест.<p>Тест.
</div>
```

Ключевым здесь является атрибут `onmousedown` во вложенном элементе `<div>`. Несмотря на то что функция `drag()` использует модели обработки событий DOM и IE, регистрация ее для удобства производится с применением модели Level 0.

Вот другой простой пример использования функции `drag()`. В нем определяется изображение, которое можно перетащить, если при этом будет удерживаться клавиша Shift:

```
<script src="Drag.js"></script>
<img src="draggable.gif" width="20" height="20"
    style="position:absolute; left:0px; top:0px;"
    onmousedown="if (event.shiftKey) drag(this, event);">
```

17.5. События клавиатуры

Как вы уже знаете, события и обработка событий – это те области, которые являются источником множества несовместимостей между броузерами. Так вот, наибольшее число несовместимостей дают события клавиатуры: они не стандартизованы в модуле `Events` модели DOM Level 2, поэтому броузеры линеек IE и Mozilla трактуют их по-разному. К сожалению, этот факт лишь отражает состояние дел в сфере обработки ввода с клавиатуры. Прикладные программные интерфейсы операционных и оконных систем, на основе которых построены броузеры, обычно отличаются сложностью. Обработка ввода текстовой информации является непростой задачей, которая осложняется наличием различных раскладок клавиатуры, вплоть до необходимости обработки ввода на идеографических языках.

Несмотря на существующие сложности, по меньшей мере для Firefox и IE можно создавать сценарии, выполняющие обработку событий клавиатуры. В этом разделе демонстрируется несколько простых сценариев, затем вашему вниманию будет представлен универсальный класс `Keymap`, который отображает события клавиатуры на JavaScript-функции, предназначенные для их обработки.

17.5.1. Типы событий клавиатуры

Существует три типа событий клавиатуры: `keydown`, `keypress` и `keyup`, которые соответствуют обработчикам событий `onkeydown`, `onkeypress` и `onkeyup`. Как правило, одно нажатие клавиши генерирует три события, когда клавиша отпускается: `keydown`, `keypress` и `keyup`. Если клавиша удерживается в нажатом состоянии и при этом включен режим автоповтора, между событиями `keydown` и `keyup` может произойти несколько событий `keypress`, но такое поведение зависит от настройки системных параметров и параметров броузера, поэтому полагаться на него нельзя.

Из трех клавиатурных событий событие `keypress` наиболее дружественное по отношению к пользователю: объект события, ассоциированный с ним, содержит код фактического символа нажатой клавиши. События `keydown` и `keyup` являются низкоуровневыми, объекты этих событий содержат так называемый «виртуальный код клавиши», который соответствует аппаратному коду, генерируемому клавиатурой. Для алфавитно-цифровых символов из набора ASCII эти виртуаль-

ные коды совпадают с ASCII-кодами, но они обработаны лишь частично. Если нажать и удерживать клавишу Shift и при этом нажать клавишу 2, событие keydown сообщит, что была нажата комбинация клавиш Shift-2. Событие keypress выполнит полную интерпретацию и сообщит, что нажатая комбинация клавиш соответствует печатному символу @. (В разных раскладках клавиатуры могут быть получены разные результаты.)

Функциональные клавиши, которые не соответствуют печатным символам, такие как Backspace, Enter, Escape, клавиши со стрелками, Page Up, Page Down и клавиши от F1 до F12, генерируют события keydown и keyup. В некоторых броузерах они также генерируют событие keypress. Однако в IE событие keypress генерируется только тогда, когда результатом нажатия является ASCII-код, т. е. печатный или управляющий символ. Функциональные клавиши, не соответствующие ни одному из печатных символов, имеют виртуальные коды, которые доступны через объект события keydown. Например, клавиша «стрелка влево» генерирует код 37 (по крайней мере, в стандартной североамериканской раскладке клавиатуры).

Таким образом, как правило, событие keydown наилучшим образом подходит для обработки нажатий функциональных клавиш, а событие keypress – для обработки нажатий клавиш с печатными символами.

17.5.2. Информация о событиях клавиатуры

Объекты событий, которые передаются обработчикам keydown, keypress и keyup, относятся к одному и тому же типу, однако интерпретация свойств этих объектов должна производиться в зависимости от типа события. Реализация объектов событий зависит от типа броузера и потому имеет разные свойства в Firefox и IE.

Если в момент нажатия клавиши была нажата и удерживалась клавиша Alt, Ctrl или Shift, этот факт отмечается в свойствах altKey, ctrlKey и shiftKey объекта события. Фактически эти свойства являются переносимыми: они доступны как в Firefox, так и в IE и для всех типов событий клавиатуры. (Единственное исключение – комбинации с клавишей Alt в IE рассматриваются как непечатаемые, поэтому они не генерируют событие keypress.)

Однако операция получения кода клавиши или символа из события клавиатуры менее переносима. В Firefox для этих целей определены два свойства. Свойство keyCode содержит низкоуровневый виртуальный код клавиши и передается с событием keydown. Свойство charCode содержит печатный символ, сгенерированный в результате нажатия клавиши, и передается с событием keypress. В Firefox функциональные клавиши генерируют событие keypress – в этом случае свойство charCode содержит ноль, а свойство keyCode содержит виртуальный код клавиши.

В IE существует единственное свойство keyCode, содержимое которого зависит от типа события. Для событий keydown свойство keyCode содержит виртуальный код клавиши, для события keypress – код символа.

Коды символов могут быть преобразованы в соответствующие им символы с помощью статической функции String.fromCharCode(). Для корректной обработки кодов клавиш достаточно просто знать, какие клавиши какие коды генерируют. В пример 17.6, который приводится в конце раздела, включена карта кодов функциональных клавиш (по крайней мере, в стандартной североамериканской раскладке клавиатуры).

17.5.3. Фильтрация ввода с клавиатуры

Обработчики событий клавиатуры могут применяться в элементах <input> и <textarea> для фильтрации ввода, получаемого от пользователя. Например, предположим, что вы хотите заставить пользователя вводить только прописные символы:

```
Фамилия: <input id="surname" type="text"
                onkeyup="this.value = this.value.toUpperCase();">
```

При возникновении события клавиатуры элемент <input> добавляет введенные символы в свое свойство value. К тому моменту, когда произойдет событие keyup, свойство value уже обновится, и потому его можно просто преобразовать его в верхний регистр. Этот прием работает независимо от положения курсора ввода в текстовом поле, позволяя использовать модель DOM Level 0. Вам не требуется знать, какая клавиша была нажата, а потому не нужно обращаться к объекту события. (Обратите внимание: если пользователь вставляет в поле ввода скопированный текст с помощью мыши, обработчик события onkeyup не вызывается. Чтобы обработать эту ситуацию, вам, скорее всего, потребуется зарегистрировать обработчик события onchange. Дополнительную информацию об элементах формы и их обработчиках событий вы найдете в главе 18.)

Сложнее выполнять фильтрацию событий клавиатуры с использованием обработчика onkeypres, когда необходимо ограничить возможность ввода некоторых символов, например предотвратить возможность ввода алфавитных символов в поле с числовыми данными. Пример 17.5 содержит определение ненавязчивого модуля, который позволяет выполнять фильтрацию именно такого рода. Он отыскивает теги <input type=text>, в которых имеется дополнительный (нестандартный) атрибут с именем allowed. Модуль регистрирует обработчики события keypress для всех таких текстовых полей ввода с целью ограничить возможность ввода символами, перечисленными в атрибуте allowed. В комментарии, расположенном в начале примера 17.5, приводятся некоторые фрагменты HTML-кода, в которых демонстрируется порядок использования модуля.

Пример 17.5. Ограничение возможности ввода определенными наборами символов

```
/**
 * InputFilter.js: ненавязчивая фильтрация нажатий клавиш для тегов <input>
 *
 * Данный модуль отыскивает все элементы <input type="text"> в документе,
 * которые имеют нестандартный атрибут "allowed". Регистрирует обработчик
 * события onkeypress для всех таких элементов с целью ограничить возможность
 * ввода символов только теми, которые перечислены в значении атрибута allowed.
 * Если элемент <input> имеет при этом атрибут "messageid", значение
 * этого атрибута воспринимается как id другого элемента документа.
 * Когда пользователь пытается ввести недопустимый символ, отображается
 * элемент messageid. Когда пользователь вводит допустимый символ,
 * элемент messageid скрывается. Элемент с данным идентификатором предназначен
 * для вывода пояснений, почему попытка ввода того или иного символа была отвергнута.
 * Изначально этот элемент с помощью CSS-стиля должен быть сделан невидимым.
 *
 * Далее приводятся некоторые примеры HTML-кода, использующие этот модуль.
 * Почтовый индекс:
 * <input id="zip" type="text" allowed="0123456789" messageid="zipwarn">
```

```
 * <span id="zipwarn" style="color:red;visibility:hidden">Только цифры</span>
 *
 * В броузерах, таких как IE, которые не поддерживают addEventListener(),
 * обработчик keypress регистрируется этим модулем за счет переопределения
 * возможно существующего обработчика события keypress.
 *
 * Этот модуль абсолютно ненавязчив, поскольку он не определяет никаких
 * символов в глобальном пространстве имен.
 */
(function() { // Весь модуль оформлен в виде анонимной функции
    // По окончании загрузки документа вызывается функция init()
    if (window.addEventListener) window.addEventListener("load", init, false);
    else if (window.attachEvent) window.attachEvent("onload", init);

    // Найти все теги <input>, для которых необходимо зарегистрировать
    // обработчик события
    function init() {
        var inputtags = document.getElementsByTagName("input");
        for(var i = 0 ; i < inputtags.length; i++) { // Обойти все теги
            var tag = inputtags[i];
            if (tag.type != "text") continue; // Только текстовые поля
            var allowed = tag.getAttribute("allowed");
            if (!allowed) continue; // И только если есть аттрибут allowed

            // Зарегистрировать функцию-обработчик
            if (tag.addEventListener)
                tag.addEventListener("keypress", filter, false);
            else {
                // attachEvent не используется, потому что в этом случае
                // функции-обработчику передается некорректное значение
                // ключевого слова this.
                tag.onkeypress = filter;
            }
        }
    }

    // Это обработчик события keypress, который выполняет фильтрацию ввода
    function filter(event) {
        // Получить объект события и код символа переносимым способом
        var e = event || window.event; // Объект события клавиатуры
        var code = e.charCode || e.keyCode; // Какая клавиша была нажата

        // Если была нажата функциональная клавиша, не фильтровать ее
        if (e.charCode == 0) return true; // Функциональная клавиша (только Firefox)
        if (e.ctrlKey || e.altKey) return true; // Нажата Ctrl или Alt
        if (code < 32) return true; // Управляющий ASCII-символ

        // Теперь получить информацию из элемента ввода
        var allowed = this.getAttribute("allowed"); // Допустимые символы
        var messageElement = null; // Сообщение об ошибке
        var messageid = this.getAttribute("messageid"); // id элемента с сообщением,
                                                        // если есть
        if (messageid) // Если существует атрибут messageid, получить элемент
            messageElement = document.getElementById(messageid);

        // Преобразовать код символа в сам символ
```

```
            var c = String.fromCharCode(code);

            // Проверить, принадлежит ли символ к набору допустимых символов
            if (allowed.indexOf(c) != -1) {
                // Если с - допустимый символ, скрыть сообщение, если существует
                if (messageElement) messageElement.style.visibility = "hidden";
                return true; // И принять ввод символа
            }
            else {
                // Если с - недопустимый символ, отобразить сообщение
                if (messageElement) messageElement.style.visibility = "visible";
                // И отвергнуть это событие keypress
                if (e.preventDefault) e.preventDefault();
                if (e.returnValue) e.returnValue = false;
                return false;
            }
        }
    })(); // Конец определения анонимной функции и ее вызов.
```

17.5.4. Быстрые комбинации клавиш и класс Keymap

Программы с графическим интерфейсом, как правило, определяют быстрые комбинации клавиш для команд, доступных через раскрывающиеся меню, панели инструментов и тому подобное. Веб-броузеры (и HTML) в основном ориентированы на использование мыши, и по умолчанию веб-приложения быстрые комбинации клавиш не поддерживают. Тем не менее такая поддержка возможна. Если веб-приложение моделирует раскрывающиеся меню средствами DHTML, необходимо также предусмотреть поддержку комбинаций быстрых клавиш для доступа к пунктам этого меню. Пример 17.6 демонстрирует, как можно этого добиться. В нем определяется класс `Keymap`, который отображает идентификаторы комбинаций клавиш, такие как «Escape», «Delete», «Alt_Z» и «alt_ctrl_shift_F5», на JavaScript-функции, вызываемые в ответ на нажатия этих комбинаций.

Привязки клавиш передаются конструктору `Keymap()` в виде JavaScript-объекта, в котором именами свойств являются идентификаторы комбинаций клавиш, а значениями свойств – функции-обработчики. Добавление и удаление привязок выполняется с помощью методов `bind()` и `unbind()`. Установка объекта `Keymap` в HTML-элемент (чаще всего в объект `Document`) выполняется методом `install()`. В процессе установки происходит регистрация обработчиков событий `onkeydown` и `onkeypress` в данном элементе с целью перехвата нажатий как функциональных, так и алфавитно-цифровых клавиш.

Начинается пример 17.6 с обширного комментария, где модуль описывается более подробно. Особое внимание следует обратить на раздел комментария, озаглавленный как «Ограничения».

Пример 17.6. Класс Keymap для реализации быстрых комбинаций клавиш

```
/*
 * Keymap.js: привязка клавиатурных событий к функциям-обработчикам.
 *
 * Этот модуль определяет класс Keymap. Экземпляр этого класса представляет
 * собой отображение идентификаторов комбинаций клавиш (определяемых далее)
 * на функции-обработчики. Объект Keymap может устанавливаться в HTML-элемент
```

```
 * для обработки событий keydown и keypress. Когда возникает такое событие,
 * объект с помощью карты отображения комбинаций вызывает соответствующую
 * функцию-обработчик.
 *
 * При создании объекта Keymap ему передается JavaScript-объект, который
 * представляет первоначальный набор привязок. Имена свойств этого объекта
 * должны совпадать с идентификаторами комбинаций клавиш, а значениями
 * этих свойств являются функции-обработчики.
 *
 * После создания объекта Keymap добавлять новые привязки можно
 * методом bind(), который принимает идентификатор комбинации
 * и функцию-обработчик. Удалять существующие привязки можно методом
 * unbind(), которому передается идентификатор комбинации клавиш.
 *
 * Чтобы использовать объект Keymap, следует вызвать его метод install(),
 * передав ему HTML-элемент, такой как объект document. Метод install()
 * добавляет к заданному объекту обработчики событий onkeypress и onkeydown,
 * возможно, замещая установленные ранее обработчики.
 * Когда вызываются эти обработчики, они определяют идентификатор
 * комбинации клавиш из события и вызывают функцию-обработчик,
 * привязанную к этой комбинации, если таковая существует.
 * Если комбинация клавиш не связана с какой-либо функцией, вызывается
 * функция-обработчик, предлагаемая по умолчанию (см. далее), если она определена.
 * Один объект Keymap может быть установлен в несколько HTML-элементов.
 *
 * Идентификаторы комбинаций клавиш
 *
 * Идентификаторы комбинаций клавиш - это строковое представление клавиши,
 * нечувствительное к регистру символов, плюс возможная клавиша-модификатор,
 * удерживаемая к моменту нажатия основной клавиши.
 * Имя клавиши - это обычно текст, написанный на самой клавише
 * в английской раскладке. Допустимыми именами клавиш считаются:
 * "A", "7", "F2", "PageUp", "Left", "Delete", "/", "~".
 * Для клавиш, соответствующих печатным символам, именем клавиши является
 * сам символ, который генерируется при нажатии клавиши.
 * Для клавиш, соответствующих непечатным символам, именами клавиш являются
 * производные от констант KeyEvent.DOM_VK_, определяемых броузером Firefox.
 * Они получаются простым отбрасыванием из имени константы префикса "DOM_VK_"
 * и удаления всех символов подчеркивания. Например, константа
 * DOM_VK_BACK_SPACE превращается в имя BACKSPACE. Полный список имен
 * находится в объекте Keymap.keyCodeToFunctionKey того же модуля.
 *
 * Идентификаторы клавиш могут содержать префиксы клавиш-модификаторов, такие
 * как Alt_, Ctrl_ и Shift_. Имена клавиш-модификаторов нечувствительны
 * к регистру символов, но если их несколько в идентификаторе комбинации,
 * они должны следовать в алфавитном порядке.
 * Примеры некоторых идентификаторов комбинаций клавиш, включающих
 * клавиши-модификаторы: "Shift_A", "ALT_F2" и "alt_ctrl_delete".
 * Обратите внимание: идентификатор "ctrl_alt_delete" считается недопустимым,
 * потому что имена клавиш-модификаторов в нем следуют не в алфавитном порядке.
 *
 * Знаки пунктуации, получаемые с помощью клавиши Shift, обычно возвращаются в виде
 * соответствующего символа. Например, Shift-2 генерирует идентификатор "@".
 * Но если при этом удерживается клавиша Alt или Ctrl, используется
```

```
 * немодифицированный символ. Например, мы получим идентификатор
 * Ctrl_Shift_2, а не Ctrl_@.
 *
 * Функции-обработчики
 *
 * Когда функция-обработчик вызывается, ей передаются три аргумента:
 *   1) HTML-элемент, в котором возникло событие
 *   2) Идентификатор нажатой комбинации клавиш
 *   3) Объект события keydown
 *
 * Обработчик по умолчанию
 *
 * В качестве имени функции-обработчика может использоваться
 * зарезервированное слово "default". Эта функция вызывается,
 * когда отсутствует специальная привязка.
 *
 * Ограничения
 *
 * Невозможно привязать функцию-обработчик ко всем клавишам. Операционные
 * системы обычно перехватывают некоторые комбинации (например, Alt-F4).
 * Броузеры также могут перехватывать некоторые комбинации
 * (например, Ctrl-S). Этот программный код зависит от типа броузера
 * операционной системы и региональных параметров.
 * Функциональные клавиши и модифицированные функциональные клавиши
 * обрабатываются без проблем, точно так же без проблем обрабатываются
 * немодифицированные алфавитно-цифровые клавиши. Менее устойчиво
 * обслуживаются комбинации алфавитно-цифровых клавиш с клавишами Ctrl
 * и Alt и в особенности с клавишами, содержащими символы пунктуации.
 */
// Функция-конструктор
function Keymap(bindings) {
    this.map = {};  // Определить ассоциативный массив identifier->handler
    if (bindings) { // Скопировать начальные привязки в него
                    // с преобразованием в нижний регистр
        for(name in bindings) this.map[name.toLowerCase( )] = bindings[name];
    }
}

// Связывает заданный идентификатор комбинации клавиш с заданной функцией-обработчиком
Keymap.prototype.bind = function(key, func) {
    this.map[key.toLowerCase()] = func;
};

// Удаляет привязку по заданному идентификатору комбинации клавиш
Keymap.prototype.unbind = function(key) {
    delete this.map[key.toLowerCase()];
};

// Установить этот объект Keymap в заданный HTML-элемент
Keymap.prototype.install = function(element) {
    // Это функция-обработчик события
    var keymap = this;
    function handler(event) { return keymap.dispatch(event); }

    // Установить ее
```

```
    if (element.addEventListener) {
        element.addEventListener("keydown", handler, false);
        element.addEventListener("keypress", handler, false);
    }
    else if (element.attachEvent) {
        element.attachEvent("onkeydown", handler);
        element.attachEvent("onkeypress", handler);
    }
    else {
        element.onkeydown = element.onkeypress = handler;
    }
};

// Этот объект отображает значения keyCode на имена
// функциональных клавиш, которые не соответствуют печатным символам.
// IE и Firefox используют практически совместимые коды клавиш.
// Однако эти коды зависят от текущей раскладки клавиатуры
// и могут иметь разные значения.
Keymap.keyCodeToFunctionKey = {
    8:"backspace", 9:"tab", 13:"return", 19:"pause", 27:"escape", 32:"space",
    33:"pageup", 34:"pagedown", 35:"end", 36:"home", 37:"left", 38:"up",
    39:"right", 40:"down", 44:"printscreen", 45:"insert", 46:"delete",
    112:"f1", 113:"f2", 114:"f3", 115:"f4", 116:"f5", 117:"f6", 118:"f7",
    119:"f8", 120:"f9", 121:"f10", 122:"f11", 123:"f12",
    144:"numlock", 145:"scrolllock"
};

// Этот объект отображает значения кодов клавиш в событии
// keydown на имена клавиш, соответствующих печатным символам.
// Алфавитно-цифровые символы имеют свой ASCII-код, но знаки пунктуации
// не имеют. Обратите внимание: коды зависят от региональных
// параметров и в национальных раскладках могут работать некорректно.
Keymap.keyCodeToPrintableChar = {
    48:"0", 49:"1", 50:"2", 51:"3", 52:"4", 53:"5", 54:"6", 55:"7", 56:"8",
    57:"9", 59:";", 61:"=", 65:"a", 66:"b", 67:"c", 68:"d",
    69:"e", 70:"f", 71:"g", 72:"h", 73:"i", 74:"j", 75:"k", 76:"l", 77:"m",
    78:"n", 79:"o", 80:"p", 81:"q", 82:"r", 83:"s", 84:"t", 85:"u", 86:"v",
    87:"w", 88:"x", 89:"y", 90:"z", 107:"+", 109:"-", 110:".", 188:",",
    190:".", 191:"/", 192:"`", 219:"[", 220:"\\", 221:"]", 222:"\""
};

// Этот метод перенаправляет клавиатурные события в соответствии с привязками.
Keymap.prototype.dispatch = function(event) {
    var e = event || window.event; // Учесть особенности модели событий IE

    // Вначале у нас нет ни клавиш модификаторов, ни имени основной клавиши
    var modifiers = ""
    var keyname = null;

    if (e.type == "keydown") {
        var code = e.keyCode;
        // Игнорировать события keydown для клавиш Shift, Ctrl и Alt
        if (code == 16 || code == 17 || code == 18) return;

        // Получить имя клавиши из карты
        keyname = Keymap.keyCodeToFunctionKey[code];
```

```
    // Если это не функциональная клавиша, но при этом нажата клавиша
    // Ctrl или Alt, необходимо интерпретировать ее как функциональную
    if (!keyname && (e.altKey || e.ctrlKey))
        keyname = Keymap.keyCodeToPrintableChar[code];

    // Если имя этой клавиши было определено, задать ее модификаторы.
    // Иначе просто вернуть управление и игнорировать это событие keydown.
    if (keyname) {
        if (e.altKey) modifiers += "alt_";
        if (e.ctrlKey) modifiers += "ctrl_";
        if (e.shiftKey) modifiers += "shift_";
    }
    else return;
}
else if (e.type == "keypress") {
    // Если была нажата клавиша Ctrl или Alt, то мы уже обработали ее.
    if (e.altKey || e.ctrlKey) return;

    // В Firefox событие keypress возникает даже для непечатных клавиш.
    // В этом случае просто вернуть управление и сделать вид, что
    // ничего не произошло.
    if (e.charCode != undefined && e.charCode == 0) return;

    // Firefox передает печатные клавиши в e.charCode, IE - в e.charCode
    var code = e.charCode || e.keyCode;

    // Данный код - это ASCII-код, поэтому можно просто преобразовать
    // его в строку.
    keyname=String.fromCharCode(code);

    // Если имя клавиши в верхнем регистре, преобразовать его
    // в нижний регистр и добавить модификатор shift.
    // Делается это для корректной обработки режима CAPS LOCK,
    // когда прописные буквы передаются без установленного модификатора shift.
    var lowercase = keyname.toLowerCase();
    if (keyname != lowercase) {
        keyname = lowercase; // Использовать форму имени в нижнем регистре
        modifiers = "shift_";// и добавить модификатор shift.
    }
}

// Теперь известны имя клавиши и модификаторы, далее необходимо
// отыскать функцию-обработчик для данной комбинации клавиш
var func = this.map[modifiers+keyname];

// Если ничего не было найдено, использовать обработчик, предлагаемый
// по умолчанию, если он существует
if (!func) func = this.map["default"];

if (func) { // Если обработчик данной комбинации существует, вызвать его
    // Указать, в каком элементе произошло событие
    var target = e.target;              // Модель DOM
    if (!target) target = e.srcElement; // Модель IE

    // Вызвать функцию-обработчик
    func(target, modifiers+keyname, e);

    // Прервать дальнейшее распространение события и предотвратить
```

```
        // выполнение действия, предлагаемого по умолчанию.
        // Обратите внимание: preventDefault обычно не предотвращает
        // выполнение верхнеуровневых команд броузера, таких как
        // нажатие клавиши F1 для вызова справочной службы.
        if (e.stopPropagation) e.stopPropagation();  // Модель DOM
        else e.cancelBubble = true;                  // Модель IE
        if (e.preventDefault) e.preventDefault();    // DOM
        else e.returnValue = false;                  // IE
        return false;                                // Ранняя модель событий
    }
};
```

17.6. Событие onload

Программный JavaScript-код, выполняющий модификации документа, в котором он содержится, как правило, должен запускаться лишь после того, как документ будет полностью загружен (детальное обсуждение этого вопроса вы найдете в разделе 13.5.7). Когда загрузка документа завершается, броузеры генерируют событие `onload` в объекте `Window`, и это событие обычно используется для запуска программного кода, которому требуется доступ ко всему документу. Если веб-страница содержит несколько независимых модулей, которые должны запускаться в ответ на событие `onload`, вам пригодится не зависящая от платформы вспомогательная функция, подобная представленной в примере 17.7.

Пример 17.7. Переносимый способ регистрации обработчиков события onload

```
/*
 * runOnLoad.js: переносимый способ регистрации обработчиков события onload.
 *
 * Данный модуль определяет единственную функцию runOnLoad(),
 * выполняющую регистрацию переносимым способом функций-обработчиков,
 * которые могут вызываться только после полной загрузки документа,
 * когда будет доступна стуктура DOM.
 *
 * Функциям, зарегистрированным с помощью runOnLoad(), не передается ни одного
 * аргумента при вызове. Они вызываются не как методы какого-либо объекта
 * и потому в них не должно использоваться ключевое слово this.
 * Функции, зарегистрированные с помощью runOnLoad(), вызываются
 * в порядке их регистрации. При этом нет никакой возможности отменить
 * регистрацию функции после того, как она передана функции runOnLoad().
 *
 * В старых броузерах, не поддерживающих addEventListener() или attachEvent(),
 * эта функция выполняет регистрацию с использованием свойства window.onload
 * модели DOM уровня 0. Она будет работать некорректно в документах,
 * где установлен атрибут onload в тегах <body> или <frameset>.
 */
function runOnLoad(f) {
    if (runOnLoad.loaded) f();    // Если документ уже загружен, просто вызвать f().
    else runOnLoad.funcs.push(f); // Иначе сохранить для вызова позднее
}

runOnLoad.funcs = []; // Массив функций, которые должны быть вызваны
                      // после загрузки документа
```

```
runOnLoad.loaded = false; // Функции еще не запускались.

// Запускает все зарегистрированные функции в порядке их регистрации.
// Допускается вызывать runOnLoad.run() более одного раза: повторные
// вызовы игнорируются. Это позволяет вызывать runOnLoad() из функций
// инициализации для регистрации других функций.
runOnLoad.run = function() {
    if (runOnLoad.loaded) return; // Если функция уже запускалась, ничего не делать
    for(var i = 0; i < runOnLoad.funcs.length; i++) {
        try { runOnLoad.funcs[i](); }
        catch(e) { /* Исключение, возникшее в одной из функций, не должно
                      делать невозможным запуск оставшихся */ }
    }

    runOnLoad.loaded = true; // Запомнить факт запуска.
    delete runOnLoad.funcs;  // Но не запоминать сами функции.
    delete runOnLoad.run;    // И даже забыть о существовании этой функции!
};

// Зарегистрировать метод runOnLoad.run() как обработчик события onload окна
if (window.addEventListener)
    window.addEventListener("load", runOnLoad.run, false);
else if (window.attachEvent) window.attachEvent("onload", runOnLoad.run);
else window.onload = runOnLoad.run;
```

17.7. Искусственные события

Обе модели обработки событий, DOM Level 2 и IE, позволяют искусственно создавать объекты событий и отправлять их обработчикам событий, зарегистрированных в элементах документа. В сущности, этот прием используется броузерами для вызова обработчиков событий, зарегистрированных в элементах (и в случае всплывающих событий обработчиков, зарегистрированных в элементах-предках). В модели обработки событий DOM Level 0 потребность в искусственных событиях не так велика, поскольку обработчики событий доступны через различные свойства обработчика. Однако в развитых моделях обработки событий нет возможности получить список обработчиков, зарегистрированных с помощью `addEventListener()` или `attachEvent()`, поэтому обработчики могут быть вызваны только с использованием приема, демонстрируемого в этом разделе.

В модели обработки событий DOM искусственное событие создается методом `Document.createEvent()`, инициализация события происходит методом `Event.initEvent()`, `UIEvent.initUIEvent()` или `MouseEvent.initMouseEvent()`, а отправка – методом `dispatchEvent()` узла, которому это событие отправляется. В модели обработки событий IE новый объект события создается методом `Document.createEventObject` и затем отправляется методом `fireEvent()` целевого элемента. В примере 17.8 демонстрируется порядок использования этих методов. Он определяет не зависящую от платформы функцию, которая отправляет искусственные события типа `dataavailable`, а также функцию, которая регистрирует обработчики событий этого типа.

Важно понимать, что искусственные события, отправляемые методом `dispatchEvent()` или `fireEvent()`, не ставятся в очередь для последующей асинхронной обработки. Они доставляются немедленно, и их обработчики вызываются синхрон-

но еще до того, как `dispatchEvent()` и `fireEvent()` вернут управление. Это означает, что искусственные события не могут использоваться для отложенного исполнения программного кода, когда броузер выполнит обработку всех ожидающих событий. Для этого необходимо вызвать метод `setTimeout()` со значением времени задержки равным нулю.

Существует возможность синтезировать и отправлять низкоуровневые события ввода, такие как события мыши, но реакция элементов документа на эти события точно не определена. Как правило, полезнее использовать эти возможности для организации высокоуровневых семантических событий, для которых в броузерах не предусматриваются действия по умолчанию. По этой причине в примере 17.8 задействован тип событий `dataavailable`.

Пример 17.8. Отправка искусственных событий

```
/**
 * DataEvent.js: отправляет и принимает события ondataavailable.
 *
 * Этот модуль определяет две функции, DataEvent.send() и DataEvent.receive(),
 * с помощью которых выполняются отправка искусственных событий dataavailable
 * и регистрация обработчиков этих событий. Программный код написан так, чтобы
 * работать в броузере Firefox и других DOM-совместимых броузарах, а также в IE.
 *
 * Модель обработки событий DOM позволяет искусственно генерировать события
 * любого типа, но модель обработки событий IE поддерживает искусственные
 * события лишь предопределенных типов. События dataavailable относятся
 * к наиболее универсальному предопределенному типу, поддерживаемому IE,
 * именно потому они здесь используются.
 *
 * Обратите внимание: отправка события методом DataEvent.send() не означает,
 * что событие будет поставлено в очередь на обработку, как это происходит
 * с реальными событиями. Вместо этого зарегистрированные обработчики
 * вызываются немедленно.
 */
var DataEvent = {};

/**
 * Отправляет искусственное событие ondataavailable заданному элементу.
 * Объект события включает в себя свойства с именами datatype и data,
 * которым присваиваются заданные значения. Свойство datatype принимает
 * значение строки или другого элементарного типа (или null),
 * идентифицирующего тип этого сообщения, а data может принимать значение
 * любого JavaScript-типа, включая массивы и объекты.
 */
DataEvent.send = function(target, datatype, data) {
    if (typeof target == "string") target = document.getElementById(target);

    // Создать объект события. Если создать его невозможно, просто вернуть управление
    if (document.createEvent) { // Модель событий DOM
        // Создать событие с заданным именем модуля событий.
        // Для событий мыши используется "MouseEvents".
        var e = document.createEvent("Events");
        // Инициировать объект события, используя метод init заданного модуля.
        // Здесь указываются тип события, способность к всплытию
        // и признак невозможности отмены.
```

```
        // См. описание Event.initEvent, MouseEvent.initMouseEvent и UIEvent.initUIEvent
        e.initEvent("dataavailable", true, false);
    }
    else if (document.createEventObject) { // Модель событий IE
        // В модели событий IE достаточно вызвать простой метод
        var e = document.createEventObject();
    }
    else return; // В других броузерах ничего не делать

    // Здесь к объекту события добавляются нестандартные свойства.
    // Кроме того, необходимо определить значения существующих свойств.
    e.datatype = datatype;
    e.data = data;

    // Отправить событие заданному элементу.
    if (target.dispatchEvent) target.dispatchEvent(e); // DOM
    else if (target.fireEvent) target.fireEvent("ondataavailable", e); // IE
};

/**
 * Регистрирует обработчик события ondataavailable в заданном элементе.
 */
DataEvent.receive = function(target, handler) {
    if (typeof target == "string") target = document.getElementById(target);
    if (target.addEventListener)
        target.addEventListener("dataavailable", handler, false);
    else if (target.attachEvent)
        target.attachEvent("ondataavailable", handler);
};
```

18

Формы и элементы форм

Как мы видели в примерах на протяжении этой книги, работа с HTML-формами – это основная часть почти всех JavaScript-программ. В данной главе объясняются детали программирования форм в JavaScript. Предполагается, что вы уже в какой-то степени знакомы с созданием HTML-форм и с содержащимися в них элементами ввода. Если нет, обратитесь к хорошей книге по HTML.[1]

Те, кто уже знаком с программированием HTML-форм на стороне сервера, заметят, что в случае форм, в которых используется JavaScript, все делается по-другому. В серверной модели форма с содержащимися в ней данными целиком отправляется на веб-сервер. Акцент делается на обработке полного набора входных данных и динамической генерации веб-страницы в качестве отклика. В JavaScript применяется совершенно иная модель программирования. В JavaScript-программах акцент делается не на передаче и обработке данных формы, а на обработке событий. Форма и все расположенные на ней элементы ввода имеют обработчики событий, позволяющие программировать реакцию на взаимодействие пользователя с формой. Если пользователь, например, щелкает на флажке, JavaScript-программа может получить уведомление через обработчик события и отреагировать на него, изменив значение, отображаемое в каком-либо другом элементе формы.

В серверных программах HTML-форма без кнопки Submit бесполезна (возможен вариант, когда форма содержит только одно текстовое поле ввода и позволяет нажать клавишу Enter для передачи данных). В то же время в JavaScript кнопка Submit не требуется (если, конечно, JavaScript-программа не работает совместно с программой на стороне сервера). В JavaScript форма может содержать произвольное количество кнопок с обработчиками событий, выполняющими при щелчке произвольное количество действий.

---

[1] Например, Чак Муссиано и Билл Кеннеди «HTML и XHTML. Подробное руководство», 6-е издание. – Пер. с англ. – СПб.: Символ-Плюс, 2008.

Как мы видели, рассматривая примеры в этой книге, обработчики событий почти всегда представляют собой центральный элемент JavaScript-программы. А чаще других используются обработчики событий, связанные с формой и ее элементами. В этой главе вводятся объект Form и различные JavaScript-объекты, представляющие элементы формы. Завершается она примером, показывающим, как с помощью JavaScript проверить на стороне клиента введенные пользователем данные перед отправкой их программе, исполняемой на стороне веб-сервера.

18.1. Объект Form

JavaScript-объект Form представляет HTML-форму. Как уже говорилось в главе 15, формы доступны в виде элементов массива forms[], являющегося свойством объекта Document. Формы расположены в этом массиве в том же порядке, что и в документе. Следовательно, элемент document.forms[0] ссылается на первую форму документа. На последнюю форму документа можно сослаться посредством следующего выражения:

```
document.forms[document.forms.length-1]
```

Наиболее интересное свойство объекта Form — массив elements[], содержащий JavaScript-объекты различных типов, представляющие различные элементы ввода формы. Элементы этого массива также располагаются в том же порядке, в котором они расположены в документе. На третий элемент второй формы документа в текущем окне можно сослаться так:

```
document.forms[1].elements[2]
```

Остальные свойства объекта Form менее важны. Свойства action, encoding, method и target соответствуют одноименным атрибутам тега <form>. Все эти свойства и атрибуты позволяют контролировать, как данные формы отправляются на веб-сервер и где будут отображаться результаты; следовательно, они полезны только в том случае, когда форма действительно отсылается серверной программе. Полное описание этих атрибутов вы найдете в специализированных изданиях, посвященных языку HTML или разработке серверных программ. Стоит отметить, что все свойства Form — это строки, доступные и для чтения, и для записи, так что JavaScript-программа может динамически устанавливать их значения перед отправкой формы.

До появления JavaScript данные формы передавались при щелчке на специальной кнопке Submit, а для сброса значений элементов формы применялась специальная кнопка Reset. JavaScript-объект Form поддерживает два метода, submit() и reset(), служащие той же цели. Вызов метода submit() формы передает данные формы, а вызов reset() сбрасывает значения элементов формы.

В дополнение к методам submit() и reset() объект Form предоставляет обработчик события onsubmit, предназначенный для обнаружения факта отправки данных формы, и обработчик события onreset, предназначенный для обнаружения факта сброса значений полей формы. Обработчик onsubmit вызывается непосредственно перед передачей данных формы; он может отменить передачу, вернув значение false. Это дает возможность JavaScript-программе проверить введенные пользователем данные на наличие ошибок, чтобы предотвратить передачу серве-

ру неполных или неверных данных. В конце этой главы мы увидим пример такой проверки. Обратите внимание: обработчик `onsubmit` вызывается только при щелчке на кнопке Submit. Вызов метода `submit()` формы не приводит к вызову обработчика `onsubmit`.

Обработчик события `onreset` работает схожим образом. Он вызывается непосредственно перед очисткой формы и может предотвратить очистку, вернув значение `false`. Это позволяет JavaScript-программе выдать запрос на подтверждение очистки данных, что может быть полезным в случае большой или подробной формы. Это можно сделать с помощью следующего обработчика события:

```
<form...
onreset="return confirm('Действительно удалить ВСЕ данные и начать сначала?')"
>
```

Как и обработчик `onsubmit`, `onreset` вызывается только при щелчке на кнопке Reset. Вызов метода `reset()` формы не приводит к вызову `onreset`.

18.2. Определение элементов формы

Элементы HTML-формы позволяют создавать простые пользовательские интерфейсы для JavaScript-программ. На рис. 18.1 показана сложная форма, содержащая минимум по одному элементу каждого из базовых типов. На тот случай,

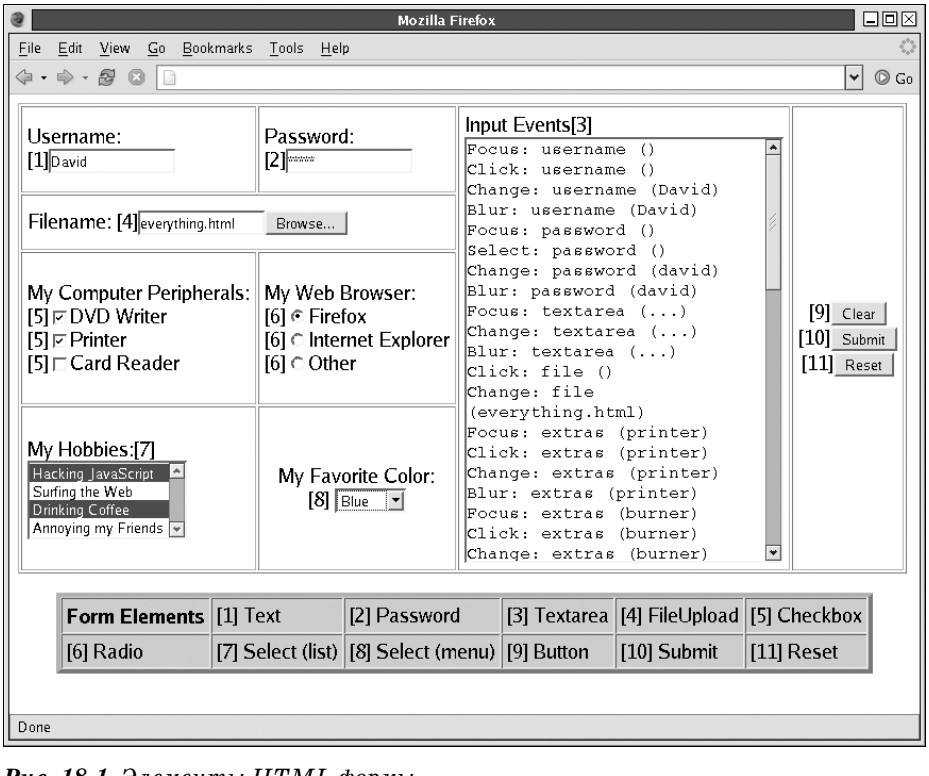

***Рис. 18.1.** Элементы HTML-формы*

если вы еще не знакомы с элементами HTML-форм, на рисунке приведены номера, идентифицирующие каждый тип элементов. Мы завершим этот раздел примером 18.1, демонстрирующим HTML- и JavaScript-код, используемый для создания показанной на рисунке формы и подключения обработчиков событий ко всем элементам формы.

В табл. 18.1 перечислены типы элементов формы, доступных HTML-дизайнерам и JavaScript-программистам. В первом столбце таблицы представлены типы элементов формы, во втором – HTML-теги, определяющие элементы данного типа, в третьем – значения свойства `type` для элементов каждого типа. Как мы видели, каждый объект `Form` содержит массив `elements[]`, в котором хранятся объекты, представляющие элементы формы. В каждом из этих элементов имеется свойство `type`, которое может применяться, чтобы отличить один тип элементов от другого. По свойству `type` неизвестного элемента формы JavaScript-код может определить тип элемента и выяснить, что можно делать с этим элементом. И наконец, четвертый столбец таблицы предоставляет наиболее важный или наиболее часто используемый обработчик события для элемента данного типа, а пятый – краткое описание каждого типа.

Обратите внимание: имена «Button» (кнопка), «Checkbox» (флажок) и прочие из первого столбца табл. 18.1 могут не соответствовать фактическим названиям клиентских JavaScript-объектов. Полное описание элементов различных типов вы найдете в части IV книги в разделах Input, Option, Select и Textarea. Кроме того, эти элементы форм подробно обсуждаются в этой главе далее.

Таблица 18.1. Элементы HTML-форм

Объект	HTML-тег	Свойство type	Событие	Описание
`Button`	`<input type="button">` или `<button type="button">`	`"button"`	`onclick`	Кнопка
`Checkbox`	`<input type="checkbox">`	`"checkbox"`	`onclick`	Флажок
`File`	`<input type="file">`	`"file"`	`onchange`	Поле для ввода имени файла, загружаемого на веб-сервер
`Hidden`	`<input type="hidden">`	`"hidden"`	Обработчиков событий нет	Данные, сохраняемые вместе с формой, но невидимые пользователю
`Option`	`<option>`	Нет	Обработчики событий подключаются к объекту `Select`, а не к отдельным объектам `Option`	Один элемент объекта `Select`
`Password`	`<input type="password">`	`"password"`	`onchange`	Поле для ввода пароля (набранные символы невидимы)

Объект	HTML-тег	Свойство type	Событие	Описание
Radio	<input type="radio">	"radio"	onclick	Переключатель – одновременно может быть установлен только один
Reset	<input type="reset"> или <button type="reset">	"reset"	onclick	Кнопка, очищающая значения формы
Select	<Select>	"select-one"	onchange	Список или выпадающее меню, в котором может быть выбран один элемент (см. также объект Option)
Select	<select multiple>	"select-multiple"	onchange	Список, в котором может быть выбрано несколько элементов (см. также объект Option).
Submit	<input type="submit"> или <button type="submit">	"submit"	onclick	Кнопка для передачи данных формы
Text	<input type="text">	"text"	onchange	Однострочное поле ввода
Textarea	<textarea>	"textarea"	onchange	Многострочное поле ввода

Теперь, получив представление о различных типах элементов формы и HTML-тегах, используемых для их создания, посмотрим на HTML-код из примера 18.1, предназначенный для создания формы, которая показана на рис. 18.1. Хотя большую часть примера занимает HTML-код, в нем также есть JavaScript-код, в котором определены обработчики событий каждого из элементов формы. Вы заметите, что обработчики событий не определяются как HTML-атрибуты. Здесь ими являются JavaScript-функции, присваиваемые свойствам объектов из массива elements[] формы. Все обработчики событий вызывают функцию report(), содержащую код, который работает с разными элементами формы. В следующем разделе этой главы описано все, что вам надо знать для понимания работы функции report().

Пример 18.1. HTML-форма, содержащая все виды элементов

```
<form name="everything">                <!-- HTML-форма "все в одном"... -->
  <table border="border" cellpadding="5"> <!-- в виде большой HTML-таблицы -->
  <tr>
    <td>Username:<br>[1]<input type="text" name="username" size="15"></td>
    <td>Password:<br>[2]<input type="password" name="password" size="15"></td>
    <td rowspan="4">Input Events[3]<br>
```

```
      <textarea name="textarea" rows="20" cols="28"></textarea></td>
    <td rowspan="4" align="center" valign="center">
     [9]<input type="button" value="Clear" name="clearbutton"><br>
     [10]<input type="submit" name="submitbutton" value="Submit"><br>
     [11]<input type="reset" name="resetbutton" value="Reset"></td></tr>
  <tr>
    <td colspan="2">
      Filename: [4]<input type="file" name="file" size="15"></td></tr>
  <tr>
    <td>My Computer Peripherals:<br>
     [5]<input type="checkbox" name="extras" value="burner">DVD Writer<br>
     [5]<input type="checkbox" name="extras" value="printer">Printer<br>
     [5]<input type="checkbox" name="extras" value="card">Card Reader</td>
    <td>My Web Browser:<br>
     [6]<input type="radio" name="browser" value="ff">Firefox<br>
     [6]<input type="radio" name="browser" value="ie">Internet Explorer<br>
     [6]<input type="radio" name="browser" value="other">Other</td></tr>
  <tr>
    <td>My Hobbies:[7]<br>
      <select multiple="multiple" name="hobbies" size="4">
        <option value="programming">Hacking JavaScript
        <option value="surfing">Surfing the Web
        <option value="caffeine">Drinking Coffee
        <option value="annoying">Annoying my Friends
      </select></td>
    <td align="center" valign="center">My Favorite Color:<br>[8]
      <select name="color">
        <option value="red">Red       <option value="green">Green
        <option value="blue">Blue     <option value="white">White
        <option value="violet">Violet <option value="peach">Peach
      </select></td></tr>
  </table>
</form>

<div align="center"> <!-- Еще одна таблица - ключ к предыдущей -->
  <table border="4" bgcolor="pink" cellspacing="1" cellpadding="4">
    <tr>
      <td align="center"><b>Form Elements</b></td>
      <td>[1] Text</td> <td>[2] Password</td> <td>[3] Textarea</td>
      <td>[4] FileU</td> <td>[5] Checkbox</td></tr>
    <tr>
      <td>[6] Radio</td> <td>[7] Select (list)</td>
      <td>[8] Select (menu)</td> <td>[9] Button</td>
      <td>[10] Submit</td> <td>[11] Reset</td></tr>
  </table>
</div>

<script>
// Эта обобщенная функция добавляет сведения о событии к тексту в большом
// многострочном поле ввода на форме. Она вызывается из различных
// обработчиков событий.
function report(element, event) {
    if ((element.type == "select-one") ||
        (element.type == "select-multiple")){
```

```
            value = " ";
            for(var i = 0; i < element.options.length; i++)
                if (element.options[i].selected)
                    value += element.options[i].value + " ";
        }
        else if (element.type == "textarea") value = "...";
        else value = element.value;
        var msg = event + ": " + element.name + " (" + value + ")\n";
        var t = element.form.textarea;
        t.value = t.value + msg;
    }

    // Эта функция добавляет к каждому элементу формы набор обработчиков событий.
    // Она не проверяет, поддерживается ли в этом элементе данный обработчик,
    // добавляются все обработчики событий. Обратите внимание, что обработчики
    // событий вызывают приведенную ранее функцию report().
    // Мы определяем обработчики событий, присваивая функции свойствам
    // JavaScript-объектов, а не строки - атрибутам HTML-элементов.

    function addhandlers(f) {
        // Цикл по всем элементам формы.
        for(var i = 0; i < f.elements.length; i++) {
            var e = f.elements[i];
            e.onclick = function() { report(this, 'Click'); }
            e.onchange = function() { report(this, 'Change'); }
            e.onfocus = function() { report(this, 'Focus'); }
            e.onblur = function() { report(this, 'Blur'); }
            e.onselect = function() { report(this, 'Select'); }
        }

        // Определить специальные обработчики событий для трех кнопок.
        f.clearbutton.onclick = function() {
            this.form.textarea.value=''; report(this,'Click');
        }
        f.submitbutton.onclick = function () {
            report(this, 'Click'); return false;
        }
        f.resetbutton.onclick = function( ) {
            this.form.reset( ); report(this, 'Click'); return false;
        }
    }

    // В заключение активизировать форму, добавив всевозможные
    // обработчики событий!
    addhandlers(document.everything);
    </script>
```

18.3. Сценарии и элементы формы

В предыдущем разделе перечислены элементы форм, предоставляемые языком HTML, и объясняется, как эти элементы встраиваются в HTML-документы. В этом разделе делается следующий шаг – вы узнаете, как можно работать с этими элементами из JavaScript-программы.

18.3.1. Именование форм и элементов форм

У каждого элемента формы есть атрибут name, который должен быть установлен в HTML-теге, если форма предназначена для отправки серверной программе. Хотя отправка данных формы, как правило, не представляет интереса для JavaScript-программ, вы скоро увидите, что есть еще одна причина указывать значение этого атрибута.

У тега <form> также есть атрибут name, который можно установить. Этот атрибут не имеет никакого отношения к отправке форм. Как уже говорилось в главе 15, он придуман для удобства JavaScript-программистов. Если в теге <form> определен атрибут name, то объект Form, создаваемый для данной формы, как обычно, сохраняется как элемент массива forms[] объекта Document, а также в собственном персональном свойстве объекта Document. Имя этого нового свойства и представляет собой значение атрибута name. В частности, в примере 18.1 мы определили форму с помощью следующего тега:

```
<form name="everything">
```

Это позволило нам ссылаться на объект Form так:

```
document.everything
```

Часто это удобнее, чем нотация массива:

```
document.forms[0]
```

Кроме того, использование имени формы делает код позиционно независимым: он останется работоспособным, даже если документ будет реорганизован так, что формы расположатся в нем в ином порядке.

У HTML-тегов <img> и <applet> есть еще атрибут name, действующий так же, как атрибут name тега <form>. Однако в формах именование идет дальше, т. к. у всех элементов формы тоже есть собственный атрибут name. Когда вы даете имя элементу формы, вы создаете новое свойство объекта Form, ссылающееся на этот элемент. Именем этого свойства становится значение атрибута. Следовательно, вы можете следующим образом сослаться на элемент с именем «zipcode», находящийся на форме с именем «address»:

```
document.address.zipcode
```

Разумно выбрав имена, можно сделать синтаксис более элегантным, чем альтернативный синтаксис, в котором индексы массивов жестко «зашиты» в исходные тексты программы (и зависят от позиции):

```
document.forms[1].elements[4]
```

Чтобы в группе переключателей мог быть установлен только один, всем входящим в группу элементам должны быть даны одинаковые имена. В примере 18.1 мы определили три переключателя, для которых значения атрибута name одинаковы – browser. Общепринятой, хотя и не обязательной, практикой также является назначение одинаковых атрибутов name группе взаимосвязанных флажков. Когда несколько элементов формы имеют одинаковые значения атрибута name, интерпретатор JavaScript просто помещает эти элементы в массив с указанным именем. Элементы массива располагаются в том порядке, в котором они присут-

ствуют в документе. Поэтому на объекты Radio (переключатель) из примера 18.1 можно ссылаться так:

```
document.everything.browser[0]
document.everything.browser[1]
document.everything.browser[2]
```

18.3.2. Свойства элементов форм

У всех (или у большинства) элементов форм есть общие свойства, перечисленные далее. Кроме того, у некоторых элементов есть специальные свойства, которые будут описаны в этой главе позже, когда мы будем рассматривать элементы форм различных типов по отдельности.

type

Доступная только для чтения строка, идентифицирующая тип элемента формы. Значения этого свойства для каждого типа элементов форм перечислены в третьем столбце табл. 18.1.

form

Ссылка на объект Form, в котором содержится этот элемент.

name

Доступная только для чтения строка, указанная в HTML-атрибуте name.

value

Доступная для чтения и записи строка, задающая «значение», содержащееся в элементе формы или представляемое им. Эта строка отсылается на веб-сервер при передаче формы и только иногда представляет интерес для JavaScript-программ. Для элементов Text (однострочное текстовое поле) и Textarea (многострочное текстовое поле) это свойство содержит введенный пользователем текст. Для элементов Button это свойство задает отображаемый на кнопке текст, который иногда требуется изменять из сценария. Свойство value для элементов Radio (переключатель) и Checkbox (флажок) не редактируется и никак не представляется пользователю. Это просто устанавливаемая HTML-атрибутом value строка, которая отсылается веб-серверу при отправке данных формы. Мы обсудим свойство value, когда позднее в этой главе будем рассматривать различные категории элементов формы.

18.3.3. Обработчики событий элементов форм

Большинство элементов форм поддерживают следующие обработчики событий:

onclick

Вызывается при щелчке левой кнопкой мыши на данном элементе. Этот обработчик особенно полезен для кнопок и сходных с ними элементов формы.

onchange

Вызывается, когда пользователь изменяет значение, представляемое элементом, например вводит текст или выбирает пункт в списке. Кнопки и сходные с ними элементы обычно не поддерживают этот обработчик события, т. к. у них нет значения, которое можно редактировать. Обратите внимание: этот

обработчик не вызывается, например, при каждом нажатии пользователем очередной клавиши в ходе заполнения поля ввода. Он вызывается, только когда пользователь изменяет значение элемента и затем перемещает фокус ввода к какому-либо другому элементу формы. То есть вызов этого обработчика события указывает на завершенное изменение.

`onfocus`

Вызывается, когда элемент формы получает фокус ввода.

`onblur`

Вызывается, когда элемент формы теряет фокус ввода.

В примере 18.1 показано, как определить обработчики событий для элементов формы. Пример разработан таким образом, что сообщает о событиях в момент их возникновения, перечисляя их в элементе `Textarea`. Это дает возможность поэкспериментировать с элементами форм и вызываемыми ими обработчиками событий.

Важно знать, что в коде обработчика события ключевое слово `this` всегда ссылается на элемент документа, вызвавший данное событие. Во всех элементах формы имеется свойство `form`, ссылающееся на форму, в которой содержится элемент, поэтому обработчики событий элемента формы всегда могут обратиться к объекту `Form` как к `this.form`. Сделав еще один шаг, мы можем сказать, что обработчик события для одной формы может ссылаться на соседний элемент формы, имеющий имя `x`, как `this.form.x`.

Обратите внимание: в этом разделе перечислены только четыре обработчика событий, представляющие особую важность для элементов форм. Помимо них элементы форм, как и (почти) все HTML-элементы, поддерживают различные обработчики (такие как `onmousedown`). Подробное обсуждение событий и обработчиков событий приведено в главе 17, а в примере 17.5 той же главы демонстрируется порядок использования обработчиков событий от клавиатуры в элементах форм.

18.3.4. Элементы Button, Submit и Reset

Элемент Button (кнопка) – один из наиболее часто используемых элементов формы, т. к. он предоставляет понятный визуальный способ вызова пользователем какого-либо запрограммированного сценарием действия. Элемент `Button` не имеет собственного поведения, предлагаемого по умолчанию, и не представляет никакой пользы без обработчика события `onclick` (или другого события). Свойство `value` элемента Button управляет текстом, появляющимся на самой кнопке. Это свойство можно установить, изменив текст (только «чистый» текст, а не HTML-текст), присутствующий на кнопке, и это часто бывает полезно.

Обратите внимание: гиперссылки предоставляют такой же обработчик события `onclick`, что и кнопки, и любой объект-кнопку можно заменить гиперссылкой, выполняющей при щелчке такое же действие. Когда вам требуется элемент, который графически выглядит как кнопка, используйте кнопку. Когда действие, вызываемое обработчиком `onclick`, можно классифицировать как «переход по ссылке», используйте ссылку.

Элементы `Submit` и `Reset` очень похожи на элемент `Button`, но имеют связанные с ними действия, предлагаемые по умолчанию (передача или очистка формы).

Так как у этих элементов имеются действия по умолчанию, они могут быть полезны даже без обработчика событий onclick. В то же время действия, выполняемые ими по умолчанию, определяют, что эти элементы полезнее для форм, отправляемых на веб-сервер, чем для чисто клиентских JavaScript-программ. Если обработчик события onclick возвращает false, стандартное действие этих кнопок не выполняется. Обработчик onclick элемента Submit позволяет проверить введенные в форме значения, но обычно это делается в обработчике onsubmit самой формы.

Создавать кнопки Button, Submit и Reset можно при помощи тега <button> вместо традиционного тега <input>. Тег <button> более гибок, т. к. выводит не просто текст, заданный атрибутом value, а любое HTML-содержимое (форматированный текст и/или изображения), присутствующее между тегами <button> и </button>. Тег <button> не обязательно должен находиться в пределах тега <form> и может размещаться в любом месте HTML-документа.

Объект Button, созданный тегом <button>, формально отличается от созданного тегом <input>, но оба имеют одинаковые значения поля type, да и в остальном их поведение довольно схоже. Основное отличие состоит в том, что тег <button> не использует значение свойства value для определения внешнего вида кнопки, т. е. внешний вид кнопки нельзя изменить путем установки свойства value.

В четвертой части книги нет описания элемента Button. Это описание, включая описание элементов, создаваемых с помощью тега <button>, вы найдете в разделе, посвященном элементу Input.

18.3.5. Элементы Checkbox и Radio

Элементы Checkbox (флажок) и Radio (переключатель) имеют два визуально различимых состояния: они могут быть либо установлены, либо сброшены. Пользователь может изменить состояние такого элемента, щелкнув на нем. Переключатели объединяются в группы связанных элементов, имеющих одинаковые значения HTML-атрибутов name. При установке одного из переключателей в группе другие переключатели сбрасываются. Флажки тоже часто составляют группы с одним значением атрибута name, и ссылаясь на них по имени, необходимо помнить, что объект, на который вы ссылаетесь, представляет собой массив элементов с одинаковыми именами. В примере 18.1 имеется три объекта Checkbox с именами extras, и мы можем ссылаться на массив из трех этих элементов следующим образом:

```
document.everything.extras
```

Для ссылки на отдельный флажок мы должны указать его индекс в массиве:

```
document.everything.extras[0] // Первый элемент формы с именем "extras"
```

У флажков и переключателей есть свойство checked. Это доступное для чтения и записи логическое значение определяет, установлен ли в данный момент элемент. Свойство defaultChecked представляет собой доступное только для чтения логическое значение, содержащее значение HTML-атрибута checked; оно определяет, должен ли элемент устанавливаться при первой загрузке страницы.

Флажки и переключатели сами не отображают какой-либо текст и обычно выводятся вместе с прилегающим к ним HTML-текстом (или со связанным тегом

<label>). Это значит, что установка свойства value элемента Checkbox или Radio не изменяет внешнего вида элемента, как это происходит с элементами Button (кнопка), создаваемыми с помощью тега <input>. Данное свойство можно установить, но это изменит лишь строку, отсылаемую на веб-сервер при передаче данных формы.

Когда пользователь щелкает на флажке или переключателе, элемент вызывает свой обработчик onclick для уведомления JavaScript-программы об изменении своего состояния. Современные броузеры вызывают также для этих элементов обработчик onchange. Оба обработчика передают в основном одну и ту же информацию, но обработчик onclick характеризуется лучшей переносимостью.

18.3.6. Элементы Text, Textarea, Password и File

Элемент Text (однострочное текстовое поле) применяется в HTML-формах и JavaScript-программах, пожалуй, чаще других. В однострочное текстовое поле пользователь может ввести короткий текст. Свойство value представляет текст, введенный пользователем. Установив это свойство, можно явно задать выводимый текст. Обработчик события onchange вызывается, когда пользователь вводит новый текст или редактирует существующий, а затем указывает, что он завершил редактирование, убрав фокус ввода из текстового поля.

Элемент Textarea (многострочное текстовое поле) очень похож на элемент Text за исключением того, что разрешает пользователю ввести (а JavaScript-программе вывести) многострочный текст. Многострочное текстовое поле создается тегом <textarea>, при этом синтаксис существенно отличается от синтаксиса тега <input>, используемого для создания однострочного текстового поля. (Подробнее об этом см. в разделе с описанием элемента Textarea в четвертой части книги.) Тем не менее эти два типа элементов ведут себя очень похожим образом. Свойство value и обработчик события onchange для многострочного текстового поля можно использовать точно так же, как для однострочного.

Элемент Password (поле ввода пароля) – это модификация однострочного текстового поля, в котором вместо вводимого пользователем текста отображаются символы звездочек. Эта особенность позволяет вводить пароли, не беспокоясь о том, что другие прочитают их через плечо. Обратите внимание: элемент Password защищает введенные пользователем данные от любопытных глаз, но при отправке данных формы эти данные никак не шифруются (если только отправка не выполняется по безопасному HTTPS-соединению) и при передаче по сети могут быть перехвачены.

И наконец, элемент File (поле ввода имени файла) предназначен для ввода пользователем имени файла, который должен быть загружен на сервер. По существу, это однострочное текстовое поле, совмещенное со встроенной кнопкой, выводящей диалоговое окно выбора файла. У элемента File, как и у однострочного текстового поля, есть обработчик события onchange. Однако в отличие от текстового поля ввода, свойство value объекта File доступно только для чтения. Это не дает злонамеренным JavaScript-программам обмануть пользователя, загрузив файл, не предназначенный для отправки на сервер.

Консорциум W3C еще не утвердил стандарт для объектов событий и обработчиков событий от клавиатуры. Тем не менее современные броузеры определяют об-

работчики событий onkeypress, onkeydown и onkeyup. Эти обработчики могут задаваться для любого объекта документа, но наиболее полезны они при назначении их текстовым полям и подобным им элементам, в которые пользователь вводит реальные данные. Можно вернуть false из обработчиков событий onkeypress или onkeydown, чтобы запретить обработку нажатой пользователем клавиши. Это может быть полезно, например, когда требуется заставить пользователя вводить только цифры. Этот прием демонстрируется в примере 17.5.

18.3.7. Элементы Select и Option

Элемент Select представляет собой набор вариантов (представленных элементами Option), которые могут быть выбраны пользователем. Броузеры обычно отображают элементы Select в виде выпадающих меню или списков. Элемент Select может работать двумя сильно различающимися способами, и значение свойства type зависит от того, как он настроен. Если в теге <select> определен атрибут multiple, пользователь может выбрать несколько вариантов, и свойство type объекта Select равно "select-multiple". В противном случае, если атрибут multiple отсутствует, может быть выбран только один вариант, и свойство type равно "select-one".

В некотором отношении элемент с возможностью множественного выбора похож на набор флажков, а элемент без такой возможности – на набор переключателей. Элемент Select отличается от них тем, что единственный элемент Select предоставляет полный набор вариантов выбора, которые задаются в HTML с помощью тега <option> и представлены в JavaScript объектами Option, хранящимися в массиве options[] элемента Select. Поскольку элемент Select предлагает набор вариантов, у него нет свойства value, как у других элементов формы. Вместо этого, о чем мы вскоре поговорим, свойство value определяется в каждом объекте Option, содержащемся в элементе Select.

Когда пользователь выбирает вариант или отменяет выбор, элемент Select вызывает свой обработчик события onchange. Для элементов Select без возможности множественного выбора значение доступного только для чтения свойства selectedIndex равно номеру выбранного в данный момент варианта. Для элементов Select с возможностью множественного выбора одного свойства selectedIndex недостаточно для представления полного набора выбранных вариантов. В этом случае для определения выбранных вариантов следует в цикле перебрать элементы массива options[] и проверить значения свойства selected каждого объекта Option.

Помимо свойства selected у элемента Option есть свойство text, задающее строку текста, которая отображается в элементе Select для данного варианта. Можно установить значение этого свойства так, чтобы изменить предлагаемый пользователю текст. Свойство value также представляет доступную для чтения и записи строку текста, который отсылается на веб-сервер при передаче данных формы. Даже если вы пишете чисто клиентскую программу, и передачи данных вашей формы никогда не происходит, свойство value (или соответствующий ей HTML-атрибут value) может применяться для хранения данных, которые потребуются при выборе пользователем определенного варианта. Обратите внимание, что элемент Option не определяет связанных с формой обработчиков событий; используйте вместо этого обработчик onchange соответствующего элемента Select.

Помимо установки свойства text объектов Option есть другие способы динамического изменения выводимых в элементе Select вариантов. Можно обрезать мас-

сив элементов Option, установив свойство options.length равным требуемому количеству вариантов, или удалить все объекты Option, установив значение свойства options.length равным нулю. Предположим, на форме с именем «address» имеется объект Select с именем «country». Тогда удалить все варианты выбора из элемента можно следующим образом:

```
document.address.country.options.length = 0; // Удаление всех вариантов
```

Мы также можем удалить отдельный объект Option элемента Select, записав в соответствующий элемент массива options[] значение null. Сделав это, вы удалите объект Option, и все элементы массива options[], индексы которых больше, автоматически сместятся для заполнения освободившейся ячейки:

```
// Удаление одного объекта Option из элемента Select
// Элемент Option, ранее расположенный в ячейке options[11],
// переместится в options[10] ...
document.address.country.options[10] = null;
```

Описание элемента Option есть в четвертой части книги. Дополнительно ознакомьтесь с описанием метода Select.add() стандарта DOM уровня 2, предоставляющего альтернативную возможность добавления новых вариантов выбора.

И наконец, элемент Option определяет конструктор Option(), который можно использовать для динамического создания новых элементов Option. Таким образом, можно добавить новые варианты в элемент Select, добавив их в конец массива options[]. Например:

```
// Создать новый объект Option
var zaire = new Option("Zaire", // Свойство text
                       "zaire", // Свойство value
                       false,   // Свойство defaultSelected
                       false);  // Свойство selected

// Отобразить вариант в элементе Select, добавив его к массиву options:
var countries = document.address.country; // Получить объект Select
countries.options[countries.options.length] = zaire;
```

Для группировки связанных вариантов внутри элемента Select может использоваться тег <optgroup>. В теге <optgroup> имеется атрибут label, задающий текст, который будет присутствовать в элементе Select. Однако несмотря на свое видимое присутствие в наборе элементов выбора, тег <optgroup> не может быть выбран пользователем, и объекты, соответствующие тегу <optgroup>, никогда не окажутся в массиве options[] элемента Select.

18.3.8. Элемент Hidden

Как следует из его имени, элемент Hidden (скрытый) не имеет визуального представления на форме. Он призван обеспечить передачу произвольного текста на сервер при отправке данных формы. Программы на стороне сервера используют элементы Hidden в качестве средства сохранения информации о состоянии, передаваемой в серверную программу при отправке данных формы. Элементы Hidden не видны на странице, поэтому они не могут генерировать события и не имеют обработчиков событий. Свойство value позволяет читать и записывать текстовое значение, связанное с элементом Hidden, но, как правило, элементы Hidden практически не используются в клиентском JavaScript-программировании.

18.3.9. Элемент Fieldset

В дополнение к описанному набору элементов, HTML-формы могут включать теги `<fieldset>` и `<label>`, которые могут играть важную роль для веб-дизайнеров, но с точки зрения клиентского JavaScript-программирования они не представляют большого интереса. Вам следует знать о теге `<fieldset>` просто потому, что при размещении такого тега в форме в массив `elements[]` добавляется соответствующий ему объект. (Однако этого не происходит при размещении тега `<label>`.) В отличие от всех других объектов из массива `elements[]`, объект, представляющий тег `<fieldset>`, не имеет свойства `value`, что может вызывать проблемы в программном коде, который предполагает наличие такого свойства.

18.4. Пример верификации формы

Мы завершим обсуждение форм расширенным примером, демонстрирующим порядок использования ненавязчивого JavaScript-кода для верификации формы.[1] Модуль программного JavaScript-кода из примера 18.3, который будет представлен чуть позже, позволяет выполнять автоматическую проверку на стороне клиента. Чтобы воспользоваться этим модулем, достаточно просто подключить его к HTML-странице, определить CSS-стили для выделения полей, содержащих некорректную информацию, и добавить дополнительные атрибуты к элементам формы. Чтобы сделать поле обязательным к заполнению, достаточно просто добавить к нему атрибут `required`. Чтобы выполнить проверку правильности с помощью регулярного выражения, следует добавить атрибут `pattern` и присвоить ему текст регулярного выражения. Пример 18.2 демонстрирует использование этого модуля, а на рис. 18.2 показано, что произойдет при попытке отправить некорректные данные формы.

Пример 18.2. Добавление модуля проверки в HTML-форму

```
<script src="Validate.js"></script> <!-- подключить модуль проверки -->
<style>
/*
 * Модуль Validate.js требует, чтобы были определены стили класса "invalid"
 * для отображения полей с некорректными данными, давая тем самым
 * пользователю отличать их визуально.
 * Для полей с корректными данными можно также определить необязательные стили.
 */
input.invalid { background: #faa; } /* Красноватый фон для полей с ошибками */
input.valid { background: #afa; }   /* Зеленоватый фон для полей, */
                                    /* заполненных правильно */
```

[1] Следует отметить, что проверка правильности заполнения полей формы на стороне клиента очень удобна для пользователей: это позволяет обнаружить и исправить ошибки еще до того, как форма будет отправлена на сервер. Однако наличие программного кода, выполняющего верификацию на стороне клиента, не гарантирует, что на сервер всегда будут отправляться корректные данные, потому что некоторые пользователи отключают режим исполнения JavaScript-кода в своих броузерах. Кроме того, проверка правильности на стороне клиента не представляет серьезной защиты от злонамеренного пользователя. По этим причинам проверка на стороне клиента никогда не сможет заменить проверку на стороне сервера.

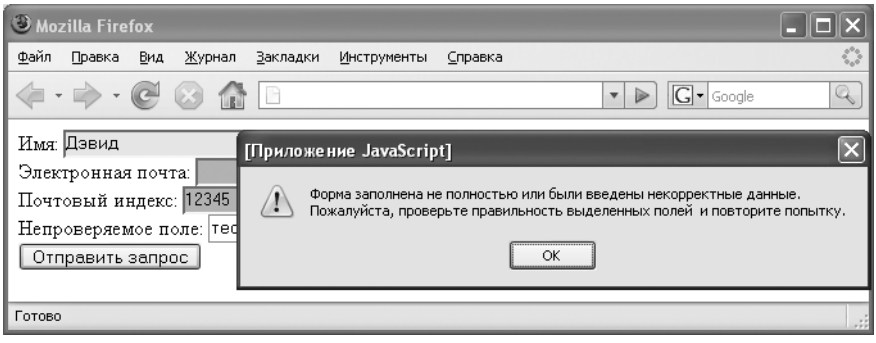

***Рис. 18.2.** Форма, не прошедшая проверку*

```
</style>

<!--
    Теперь, чтобы включить проверку полей формы, нужно просто установить
    атрибут required или pattern.
-->
<form>
<!-- Это поле должно быть заполнено -->
Имя: <input type="text" name="name" required><br>
<!--
    \s* - означает необязательный пробел.
    \w+ - один или более алфавитно-цифровых символов
-->
Электронная почта: <input type="text" name="email" pattern="^\s*\w+@\w+\.\w+\s*$"><br>
<!-- \d{6} означает, что должно быть введено ровно шесть цифр -->
Почтовый индекс: <input type="text" name="zip" pattern="^\s*\d{6}\s*$"><br>
<!-- следующее поле не проверяется -->
Непроверяемое поле: <input type="text"><br>
<input type="submit">
</form>
```

Пример 18.3 содержит программный код модуля верификации формы. Обратите внимание: при подключении этого модуля к HTML-файлу в глобальное пространство имен не добавляется ни одного имени, кроме того, модуль автоматически регистрирует обработчик события `onload`, который выполняет обход всех форм документа, отыскивает атрибуты `required` и `pattern` и в случае необходимости добавляет обработчики событий `onchange` и `onsubmit`. Эти обработчики устанавливают значение свойства `className` каждого элемента формы, который подвергается проверке, в значение `"invalid"` или `"valid"`, потому необходимо предусмотреть определение хотя бы для «неправильного» (invalid) CSS-класса,[1] чтобы обеспечить визуальное отличие полей с корректными и некорректными данными.

[1] Ко времени написания этих строк существующее средство автоматического заполнения полей панели инструментов Google использует CSS-стили для установки цвета фона некоторых текстовых полей. Расширение для броузеров, выпущенное компанией Google, делает светло-желтым цвет фона для полей, куда автоматически могут быть подставлены значения.

Пример 18.3. Автоматическая проверка формы

```
/**
 * Validate.js: ненавязчивая проверка HTML-форм.
 *
 * После загрузки документа данный модуль сканирует документ в поисках
 * HTML-форм и текстовых полей в формах. Если обнаруживаются элементы
 * с атрибутом "required" или "pattern", к ним добавляются соответствующие
 * обработчики событий, выполняющие проверку данных формы.
 *
 * Если элемент формы имеет атрибут "pattern", значение этого атрибута
 * используется как регулярное JavaScript-выражение, а элементу назначается
 * обработчик события onchange, который проверяет ввод пользователя с помощью
 * этого шаблона. Если данные не соответствуют шаблону, цвет фона элемента
 * ввода изменяется, чтобы привлечь внимание пользователя.
 * По умолчанию текстовое поле должно содержать некоторую подстроку, которая
 * соответствует шаблону. Если требуется указать более строгое соответствие,
 * используйте якорные элементы ^ и $ в начале и конце шаблона.
 *
 * Элемент формы с атрибутом "required" должен содержать какое-либо значение.
 * Если быть более точным, атрибут "required" является краткой формой атрибута
 * pattern="\S". То есть этот атрибут требует, чтобы поле содержало хотя бы
 * один символ, отличный от пробела
 *
 * Если элемент формы прошел проверку, в атрибут "class" этого элемента
 * записывается значение "valid". В противном случае - значение "invalid".
 * Чтобы извлечь из этого модуля выгоду, необходимо вместе с ним использовать
 * таблицу CSS-стилей, где определяются стили для "неправильного" класса.
 * Например:
 *
 * <!-- для привлечения внимания окрасить фон элементов формы, содержащих
 *      ошибки, в оранжевый цвет -->
 * <style>input.invalid { background: #fa0; }</style>
 *
 * Перед отправкой формы текстовые поля, требующие проверки, подвергаются
 * повторной верификации. Если обнаруживаются ошибки, отправка формы
 * блокируется и выводится диалоговое окно, в котором пользователю сообщается
 * о том, что форма заполнена не полностью или содержит ошибки.
 *
 * Этот модуль не может использоваться для проверки форм или полей, в которых
 * вы определили собственный обработчик событий onchange или onsubmit,
 * а также полей, для которых вы определили свое значение атрибута class.
 *
 * Весь программный код модуля размещается внутри анонимной функции
 * и не определяет ни одного имени в глобальном пространстве имен.
 */
(function() { // Все, что требуется, выполняется в этой анонимной функции
    // По окончании загрузки документа вызвать функцию init()
    if (window.addEventListener) window.addEventListener("load", init, false);
    else if (window.attachEvent) window.attachEvent("onload", init);

    // Устанавливает обработчики событий для форм и элементов форм,
    // где это необходимо.
    function init() {
```

```
    // Цикл по всем формам в документе
    for(var i = 0; i < document.forms.length; i++) {
        var f = document.forms[i]; // Текущая форма

        // Предположить, что форма не требует проверки
        var needsValidation = false;

        // Цикл по всем элементам на форме
        for(j = 0; j < f.elements.length; j++) {
            var e = f.elements[j]; // Текущий элемент

            // Интерес представляют только поля <input type="text">
            if (e.type != "text") continue;

            // Проверить, имеются ли атрибуты, требующие проверки
            var pattern = e.getAttribute("pattern");
            // Можно было бы использовать e.hasAttribute(),
            // но IE не поддерживает его.
            var required = e.getAttribute("required") != null;

            // Атрибут required - это лишь краткая форма записи
            // атрибута pattern
            if (required && !pattern) {
                pattern = "\\S";
                e.setAttribute("pattern", pattern);
            }

            // Если элемент требует проверки,
            if (pattern) {
                // проверять при каждом изменении содержимого элемента
                e.onchange = validateOnChange;
                // Запомнить, чтобы потом добавить обработчик onsubmit
                needsValidation = true;
            }
        }

        // Если хотя бы один элемент формы требует проверки,
        // то необходимо установить обработчик события onsubmit формы
        if (needsValidation) f.onsubmit = validateOnSubmit;
    }
}

// Эта функция - обработчик события onchange для текстового поля, которое
// требует проверки. Не забывайте, что в функции init() мы преобразовали
// атрибут required в pattern.
function validateOnChange() {
    var textfield = this;              // Текстовое поле
    var pattern = textfield.getAttribute("pattern");    // Шаблон
    var value = this.value;            // Данные, введенные пользователем

    // Если значение не соответствует шаблону, установить значение
    // атрибута class равным "invalid".
    if (value.search(pattern) == -1) textfield.className = "invalid";
    else textfield.className = "valid";
}

// Эта функция - обработчик события onsubmit для любой формы,
// требующей проверки.
```

```
    function validateOnSubmit() {
        // Перед отправкой формы выполнить проверку всех полей в форме
        // и установить их свойства className в соответствующее значение.
        // Если хотя бы одно из этих полей содержит ошибку, вывести диалоговое
        // окно и заблокировать отправку данных формы.
        var invalid = false; // Предполагаем, что все правильно

        // Цикл по всем элементам формы
        for(var i = 0; i < this.elements.length; i++) {
            var e = this.elements[i];

            // Если элемент – это текстовое поле, для которого установлен
            // наш обработчик события onchange
            if (e.type == "text" && e.onchange == validateOnChange) {
                e.onchange( ); // Вызвать обработчик для повторной проверки

                // Если проверка не пройдена, это означает, что
                // вся форма не прошла проверку
                if (e.className == "invalid") invalid = true;
            }
        }

        // Если форма не прошла проверку, вывести диалоговое окно
        // и заблокировать отправку формы
        if (invalid) {
            alert("Форма заполнена не полностью " +
                  "или были введены некорректные данные.\n" +
                  "Пожалуйста, проверьте правильность выделенных полей " +
                  " и повторите попытку.");
            return false;
        }
    }
})();
```

19

Cookies и механизм сохранения данных на стороне клиента

В объекте `Document` имеется не обсуждавшееся в главе 15 свойство под названием `cookie`. На первый взгляд кажется, что это свойство представляет собой простое строковое значение; однако свойство `cookie` – это много больше, чем просто строка: оно позволяет JavaScript-коду сохранять данные на жестком диске в пользовательской системе и извлекать сохраненные ранее данные. Cookies представляют собой простой способ наделить веб-приложения долговременной памятью, например, веб-сайт может сохранять личные предпочтения пользователя и затем использовать их при повторном посещении страницы.

Кроме того, cookies могут использоваться сценариями на стороне сервера и являются стандартным расширением протокола HTTP. Все современные броузеры поддерживают cookies и предоставляют к ним доступ через свойство `Document.cookie`. Существует еще один более мощный механизм сохранения данных на стороне клиента, но он плохо стандартизован. Подробнее этот механизм обсуждается в конце главы.

19.1. Обзор cookies

Cookie – это небольшой объем именованных данных, сохраняемых веб-броузером и связанных с определенной веб-страницей или веб-сайтом.[1] Cookies играют

[1] Особого смысла у термина «cookie» (булочка) нет, тем не менее появился он не «с потолка». В туманных анналах истории компьютеров термин «cookie», или «magic cookie», использовался для обозначения небольшой порции данных, в частности, привилегированных или секретных данных, вроде пароля, подтверждающих подлинность или разрешающих доступ. В JavaScript cookie-файлы применяются для сохранения информации о состоянии и могут служить средством идентификации веб-броузера, хотя они не шифруются и никак не связаны с безопасностью (впрочем, это не относится к передаче их через защищенное соединение по протоколу HTTPS).

роль памяти веб-броузера, чтобы сценарии и программы на стороне сервера могли на одной странице работать с данными, введенными на другой странице, или чтобы броузер мог вспомнить пользовательские параметры или другие переменные состояния, когда возвращается на страницу, посещенную им ранее. Cookies первоначально предназначались для разработки серверных сценариев и на низшем уровне реализованы как расширение протокола HTTP. Данные cookie автоматически передаются между веб-броузером и веб-сервером, так что серверные сценарии могут читать и записывать значения cookie, сохраняемые на стороне клиента. Как мы увидим, JavaScript также может работать с cookie с помощью свойства `cookie` объекта `Document`.

Свойство `cookie` – это строковое свойство, позволяющее читать, создавать, изменять и удалять cookies, связанные с текущей веб-страницей. Хотя `cookie` с первого взгляда может показаться обычным доступным для чтения и записи строковым свойством, фактически его поведение сложнее. Читая значение свойства `cookie`, мы получаем строку, содержащую имена и значения всех cookies, связанных с документом. Можно создавать, изменять и удалять cookies, устанавливая значение свойства `cookie`. В следующих разделах этой главы подробно объясняется, как это делать. Однако для того чтобы работа со свойством `cookie` была эффективной, надо больше знать о cookies и о том, как они работают.

Помимо имени и значения каждый cookie имеет четыре необязательных атрибута, управляющих временем его жизни, видимостью и безопасностью. По умолчанию cookies являются временными – их значения сохраняются на период сеанса веб-броузера и теряются при закрытии сеанса пользователем. Чтобы cookie сохранялся после окончания сеанса, необходимо сообщить броузеру, как долго он должен храниться. Изначально для этого использовался атрибут `expires`, указывающий дату окончания действия cookie. И хотя этот атрибут по-прежнему может применяться, он начинает вытесняться другим атрибутом – `max-age`, который определяет срок хранения cookie в секундах. Установка значения любого из этих атрибутов заставляет броузер сохранить cookie в локальном файле, чтобы он мог быть прочитан при следующем посещении пользователем веб-страницы. После того как будет достигнута дата окончания действия или истечет период `max-age`, броузер автоматически удалит cookie-файл.

Еще один немаловажный атрибут cookie, `path`, задает веб-страницы, с которыми связан cookie. По умолчанию cookie связывается с создавшей его веб-страницей и доступен этой же странице, а также любой другой странице из того же каталога или любых его подкаталогов. Если, например, веб-страница *http://www.example.com/catalog/index.html* создает cookie, то этот cookie будет также видим страницам *http://www.example.com/catalog/order.html* и *http://www.example.com/catalog/widgets/index.html*, но не видим странице *http://www.example.com/about.html*.

Этого правила видимости, принятого по умолчанию, обычно вполне достаточно. Тем не менее иногда значения cookie-файла требуется использовать на всем многостраничном веб-сайте независимо от того, какая страница создала cookie. Например, если пользователь ввел свой адрес в форму на одной странице, целесообразно сохранить этот адрес как адрес, применяемый по умолчанию. Тогда этим адресом можно будет воспользоваться при следующем посещении тем же пользователем этой страницы, а также при заполнении им совершенно другой формы на любой другой странице, где требуется ввести адрес, например для выставления

счета. Чтобы это можно было сделать, для cookie-файла задается значение `path`. Тогда любая страница того же веб-сервера, содержащая указанное значение в своем URL, сможет использовать cookie-файл. Например, если для cookie, установленного страницей *http://www.example.com/catalog/widgets/index.html*, для атрибута `path` задано значение `"/catalog"`, этот cookie будет также виден для страницы *http://www.example.com/catalog/order.html*. А если атрибут `path` установлен в `"/"`, то cookie-файл будет виден для любой страницы на веб-сервере *www.example.com*.

По умолчанию cookies доступны только страницам, загружаемым с того веб-сервера, который их установил. Однако большим веб-сайтам может потребоваться возможность совместного использования cookies несколькими веб-серверами. Например, серверу *order.example.com* может потребоваться прочитать значения cookie, установленного сервером *catalog.example.com*. В этой ситуации поможет третий атрибут cookie-файла – `domain`. Если cookie, созданный страницей с сервера *catalog.example.com*, установил свой атрибут `path`, равным `"/"`, а атрибут `domain` – равным `".example.com"`, этот cookie будет доступен всем веб-страницам серверов *catalog.example.com*, *orders.example.com* и любых других серверов в домене *example.com*. Если атрибут `domain` для cookie не установлен, значением по умолчанию будет имя веб-сервера, на котором находится страница. Обратите внимание, что нельзя сделать так, чтобы домен cookie-файла отличался от домена вашего сервера.

Последний атрибут cookie – это логический атрибут с именем `secure`, определяющий, как значения cookie-файла передаются по сети. По умолчанию cookie не защищен, т. е. передается по обычному незащищенному HTTP-соединению. Однако если cookie помечен как защищенный, он передается, только когда обмен между броузером и сервером организован по протоколу HTTPS или другому защищенному протоколу.

Обратите внимание: атрибуты `expires`, `max-age`, `path`, `domain` и `secure` не являются свойствами JavaScript-объекта. Далее в этой главе мы увидим, как установить эти атрибуты cookie-файла.

Cookies пользуются дурной славой у многих пользователей Всемирной паутины, поскольку сторонние производители часто недобросовестно применяют cookies, связанные не с самой веб-страницей, а с изображениями на ней. Например, cookies сторонних производителей позволяют компаниям, предоставляющим услуги рекламного характера, отслеживать перемещение пользователей с одного сайта на другой, что вынуждает многих пользователей по соображениям безопасности отключать режим сохранения cookies в своих веб-броузерах. Поэтому, прежде чем использовать cookie в сценариях JavaScript, следует проверить, не отключен ли режим их сохранения. В большинстве броузеров это можно сделать, проверив свойство `navigator.cookieEnabled`. Если оно содержит значение `true`, значит работа с cookie разрешена, а если `false` – запрещена (хотя при этом могут быть разрешены временные cookie-файлы, срок жизни которых ограничивается продолжительностью сеанса работы броузера). Это свойство не является стандартным, поэтому если сценарий вдруг обнаружит, что оно не определено, придется проверить, поддерживаются ли cookies, попытавшись записать, прочитать и удалить тестовый cookie-файл. Как это делается, описано далее в этой главе, а в примере 19.2 вы найдете программный код, выполняющий такую проверку.

Те, кто интересуется техническими подробностями работы cookies (на уровне протокола HTTP), могут обратиться к спецификации RFC 2965 на странице *http://*

www.ietf.org/rfc/rfc2965.txt. Изначально cookie были разработаны в компании Netscape и их первичная спецификация, подготовленная в Netscape, может попрежнему представлять интерес. Хотя некоторые ее части уже сильно устарели, она много короче и проще, чем формальный документ RFC. Отыскать этот старый документ можно на странице *http://wp.netscape.com/newsref/std/cookie_spec.html*.

В следующих разделах обсуждаются вопросы доступа к значениям cookie из JavaScript-сценариев и порядок работы с атрибутами `expires`, `path`, `domain` и `secure`. Затем рассматриваются альтернативные способы сохранения данных на стороне клиента.

19.2. Сохранение cookie

Чтобы связать временное значение cookie-файла с текущим документом, достаточно установить свойство `cookie` равным строке следующего формата:

```
имя=значение
```

Например:

```
document.cookie = "version=" + encodeURIComponent(document.lastModified);
```

При следующем чтении свойства `cookie` сохраненная пара «имя–значение» будет включена в список cookie-файлов документа. Значения cookie не могут содержать точки с запятой, запятые или символы-разделители. По этой причине для кодирования значения перед сохранением его в cookie-файле, возможно, потребуется использовать JavaScript-функцию `encodeURIComponent()`. В этом случае при чтении значения cookie-файла надо будет вызвать соответствующую функцию `decodeURIComponent()`. (Нередко можно встретить программный код, использующий для тех же целей устаревшие функции `escape()` и `unescape()`.)

Записанный указанным способом cookie сохраняется в текущем сеансе работы веб-броузера, но теряется при закрытии броузера пользователем. Чтобы создать cookie, сохраняющийся между сеансами броузера, необходимо указать время жизни (в секундах) с помощью атрибута `max-age`. Это можно сделать, установив значение свойства `cookie` равным строке следующего формата:

```
name=value; max-age=seconds
```

Например, чтобы создать cookie, сохраняющийся в течение года, можно использовать следующий фрагмент:

```
document.cookie = "version=" + document.lastModified + "; max-age=" + (60*60*24*365);
```

Кроме того, существует возможность указать время жизни cookie с помощью устаревшего атрибута `expires`, в который необходимо записать дату в формате, возвращаемом функцией `Date.toGMTString()`. Например:

```
var nextyear = new Date();
nextyear.setFullYear(nextyear.getFullYear() + 1);
document.cookie = "version=" + document.lastModified +
                  "; expires=" + nextyear.toGMTString();
```

Аналогичным образом можно установить атрибуты `path`, `domain` и `secure`, дописав к значению cookie-файла строки следующего формата перед его записью в свойство `cookie`:

```
; path=путь
; domain=домен
; secure
```

Чтобы изменить значение cookie, установите его значение снова, указав то же имя и новое значение. При изменении значения cookie-файла можно также переопределить время жизни, указав новые значения для атрибута `max-age` или `expires`.

Чтобы удалить cookie, установите произвольное (возможно пустое) значение с тем же именем, а в атрибут `max-age` запишите 0 (или в атрибут `expires` запишите уже прошедшую дату). Обратите внимание: броузер не обязан удалять cookie немедленно, так что он может сохраниться броузером и после даты окончания его действия.

19.2.1. Ограничения cookie

Cookie-файлы рассчитаны на то, чтобы изредка сохранять небольшие объемы данных. Они не являются универсальным средством взаимодействия или механизмом передачи данных, поэтому следует проявлять умеренность при их использовании. Спецификации RFC 2965 рекомендуют производителям броузеров не ограничивать число и размеры сохраняемых cookie-файлов. Однако следует знать, что стандарты не требуют, чтобы веб-броузеры сохраняли более 300 cookies и 20 cookies на один веб-сервер (на весь веб-сервер, а не только на вашу страницу или сайт на сервере) или по 4 Кбайт данных на один cookie (в этом ограничении учитываются и значение cookie-файла и его имя). На практике современные броузеры позволяют сохранять гораздо больше 300 cookies, но ограничение на размер 4 Кбайт для одного cookie в некоторых броузерах по-прежнему соблюдается.

19.3. Чтение cookies

Когда свойство `cookie` используется в JavaScript-выражении, возвращаемое им значение содержит все cookie-файлы, относящиеся к текущему документу. Эта строка представляет собой список пар *имя–значение*, разделенных точками с запятой, где *имя* – это имя cookie-файла, а *значение* – его строковое значение. Это значение не включает каких-либо атрибутов, которые могли быть установлены для cookie. Для получения значения cookie с определенным именем могут использоваться методы `String.indexOf()` и `String.substring()`, а для разбиения строки cookie-файла на отдельные составляющие – метод `String.split()`.

После извлечения значений cookie-файла из свойства `cookie` их требуется интерпретировать, основываясь на том формате или кодировке, которые были указаны создателем cookie. Например, в одном cookie может храниться несколько единиц информации в полях, разделенных двоеточиями. В этом случае придется для извлечения различных фрагментов информации обратиться к соответствующим строковым методам. Не забудьте вызвать функцию `decodeURIComponent()` для значения cookie, если оно было закодировано функцией `encodeURIComponent()`.

Следующий фрагмент демонстрирует, как выполняется чтение свойства `cookie`, как из него извлекается отдельное значение и как потом можно использовать это значение:

```
// Прочитать свойство cookie. В результате будут получены все cookies данного документа.
```

```
var allcookies = document.cookie;
// Отыскать начало cookie-файла с именем "version"
var pos = allcookies.indexOf("version=");

// Если cookie с данным именем найден, извлечь и использовать его значение
if (pos != -1) {
    var start = pos + 8;                          // Начало значения cookie
    var end = allcookies.indexOf(";", start);     // Конец значения cookie
    if (end == -1) end = allcookies.length;
    var value = allcookies.substring(start, end); // Извлекаем значение
    value = decodeURIComponent (value);           // Декодируем его

    // Теперь, получив значение cookie-файла, мы можем его использовать.
    // В данном случае значение было установлено равным дате изменения
    // документа, поэтому мы можем использовать это значение, чтобы узнать,
    // был ли документ изменен с момента последнего посещения пользователем.
    if (value != document.lastModified)
        alert("Документ был изменен с момента вашего последнего посещения");
}
```

Обратите внимание: строка, полученная при чтении значения свойства `cookie`, не содержит какой-либо информации о различных атрибутах cookie-файла. Свойство `cookie` позволяет установить эти атрибуты, но не дает возможности прочитать их.

19.4. Пример работы с cookie

Дискуссия о cookies завершается примером 19.2, где определяется вспомогательный класс, предназначенный для работы с cookies. Конструктор `Cookie()` читает значение cookie с заданным именем. Метод `store()` записывает данные в этот cookie, при этом устанавливаются значения атрибутов, определяющих срок жизни, путь и домен, а метод `remove()` удаляет cookie, записывая значение 0 в атрибут `max-age`.

Класс `Cookie`, определяемый в этом примере, сохраняет имена и значения нескольких переменных состояния в единственном cookie. Чтобы записать данные в cookie, достаточно просто установить значения свойств объекта `Cookie`. Когда вызывается метод `store()`, имена и значения свойств, добавленные к объекту, становятся значением сохраняемого cookie-файла. Аналогично при создании нового объекта `Cookie` конструктор `Cookie()` ищет существующий cookie с заданным именем, и если находит, его значение интерпретируется как набор пар имя–значение, после чего создаются соответствующие свойства нового объекта `Cookie`.

Чтобы помочь разобраться с примером 19.2, сначала рассмотрим пример 19.1, который представляет собой простую веб-страницу, использующую класс `Cookie`.

Пример 19.1. Порядок использования класса Cookie

```
<script src="Cookie.js"></script><!-- подключить класс Cookie -->
<script>
// Создать cookie, который будет использоваться для сохранения
// информации о состоянии данной веб-страницы.
var cookie = new Cookie("vistordata");
```

```
// Сначала попытаться прочитать данные, хранящиеся в cookie.
// Если он еще не существует (или не содержит требуемые данные),
// запросить данные у пользователя
if (!cookie.name || !cookie.color) {
    cookie.name = prompt("Введите ваше имя:", "");
    cookie.color = prompt("Какой цвет вы предпочитаете:", "");
}

// Запомнить число посещений страницы пользователем
if (!cookie.visits) cookie.visits = 1;
else cookie.visits++;

// Сохранить данные в cookie, куда включается счетчик числа посещений.
// Определить срок жизни cookie в 10 дней. Поскольку атрибут path
// не определяется, cookie будет доступен всем веб-страницам того же
// каталога и вложенных каталогов. Поэтому необходимо гарантировать,
// что имя cookie "visitordata" будет уникальным для всех этих страниц.
cookie.store(10);

// Теперь можно воспользоваться данными, полученными из cookie
// (или от пользователя), чтобы приветствовать пользователя по имени,
// подсветив приветствие цветом, который предпочитает пользователь.
document.write('<h1 style="color:' + cookie.color + '">' +
        'Добро пожаловать, ' + cookie.name + '!' + '</h1>' +
        '<p>Вы посетили нас ' + cookie.visits + ' раз.' +
        '<button onclick="window.cookie.remove();">Забыть обо мне </button>');
</script>
```

Собственно определение класса `Cookie` приводится в примере 19.2.

Пример 19.2. Вспомогательный класс Cookie

```
/**
 * Это функция-конструктор Cookie().
 *
 * Данный конструктор отыскивает cookie с заданным именем для текущего документа.
 * Если cookie существует, его значение интерпретируется как набор пар имя-значение,
 * после чего эти значения сохраняются в виде свойств вновь созданного объекта.
 *
 * Чтобы сохранить в cookie новые данные, достаточно просто установить
 * значение свойства объекта Cookie. Избегайте использования свойств
 * с именами "store" и "remove", поскольку эти имена зарезервированы для методов объекта.
 * Чтобы сохранить данные cookie в локальном хранилище веб-броузера,
 * следует вызвать метод store().
 * Чтобы удалить данные cookie из хранилища броузера, нужно вызвать метод remove().
 *
 * Статический метод Cookie.enabled() возвращает значение true,
 * если использование cookies разрешено в броузере, в противном случае
 * возвращается значение false.
 */
function Cookie(name) {
    this.$name = name; // Запомнить имя данного cookie

    // Прежде всего необходимо получить список всех cookies,
    // принадлежащих этому документу.
    // Для этого следует прочесть содержимое свойства Document.cookie.
```

```
    // Если ни одного cookie не найдено, ничего не предпринимать.
    var allcookies = document.cookie;
    if (allcookies == "") return;

    // Разбить строку на отдельные cookies, а затем выполнить
    // цикл по всем полученным строкам в поисках требуемого имени.
    var cookies = allcookies.split(';');
    var cookie = null;
    for(var i = 0; i < cookies.length; i++) {
        // Начинается ли текущая строка cookie с искомого имени?
        if (cookies[i].substring(0, name.length+1) == (name + "=")) {
            cookie = cookies[i];
            break;
        }
    }

    // Если cookie с требуемым именем не найден, вернуть управление
    if (cookie == null) return;

    // Значение cookie находится вслед за знаком равенства
    var cookieval = cookie.substring(name.length+1);

    // После того как значение именованного cookie-файла получено,
    // необходимо разбить его на отдельные пары имя-значение переменных
    // состояния. Пары имя-значение отделяются друг от друга символом
    // амперсанда, а имя от значения внутри пары отделяется двоеточиями.
    // Для интерпретации значения cookie-файла используется метод split().
    var a = cookieval.split('&');   // Превратить в массив пар имя-значение
    for(var i=0; i < a.length; i++) // Разбить каждую пару в массиве
        a[i] = a[i].split(':');

    // Теперь, после того как закончена интерпретация значения cookie,
    // необходимо определить свойства объекта Cookie и установить их значения.
    // Обратите внимание: значения свойств необходимо декодировать,
    // потому что метод store() кодирует их.
    for(var i = 0; i < a.length; i++) {
        this[a[i][0]] = decodeURIComponent(a[i][1]);
    }
}

/**
 * Данная функция - метод store() объекта Cookie.
 *
 * Аргументы:
 *
 * daysToLive: срок жизни cookie-файла в сутках. Если установить значение
 * этого аргумента равным нулю, cookie будет удален. Если установить
 * значение null или опустить этот аргумент, срок жизни cookie будет
 * ограничен продолжительностью сеанса и сам cookie не будет сохранен
 * броузером по завершении работы. Этот аргумент используется
 * для установки значения атрибута max-age в cookie-файле.
 * path: значение атрибута path в cookie
 * domain: значение атрибута domain в cookie
 * secure: если передается значение true, устанавливается
 * атрибут secure в cookie-файле
 */
```

```
Cookie.prototype.store = function(daysToLive, path, domain, secure) {
    // Сначала нужно обойти в цикле свойства объекта Cookie и объединить
    // их в виде значения cookie-файла. Поскольку символы знака равенства
    // и точки с запятой используются для нужд оформления cookies,
    // для отделения друг от друга переменных состояния, составляющих
    // значение cookie-файла, используются символы амперсанда,
    // а для отделения имен и значений внутри пар - двоеточия.
    // Обратите внимание: значение каждого свойства необходимо
    // кодировать на случай, если в них присутствуют знаки пунктуации
    // или другие недопустимые символы.
    var cookieval = "";
    for(var prop in this) {
        // Игнорировать методы, а также имена свойств, начинающиеся с '$'
        if ((prop.charAt(0) == '$') || ((typeof this[prop]) == 'function'))
            continue;
        if (cookieval != "") cookieval += '&';
        cookieval += prop + ':' + encodeURIComponent(this[prop]);
    }

    // Теперь, когда получено значение cookie-файла, можно создать полную
    // строку cookie-файла, которая включает имя и различные атрибуты,
    // заданные при создании объекта Cookie
    var cookie = this.$name + '=' + cookieval;
    if (daysToLive || daysToLive == 0) {
        cookie += "; max-age=" + (daysToLive*24*60*60);
    }

    if (path) cookie += "; path=" + path;
    if (domain) cookie += "; domain=" + domain;
    if (secure) cookie += "; secure";

    // Теперь нужно сохранить cookie, установив свойство Document.cookie
    document.cookie = cookie;
}

/**
 * Эта функция - метод remove() объекта Cookie; он удаляет свойства объекта
 * и сам cookie из локального хранилища броузера.
 *
 * Все аргументы этой функции являются необязательными, но чтобы удалить
 * cookie, необходимо передать те же значения, которые передавались методу store().
 */
Cookie.prototype.remove = function(path, domain, secure) {
    // Удалить свойства объекта Cookie
    for(var prop in this) {
        if (prop.charAt(0) != '$' && typeof this[prop] != 'function')
            delete this[prop];
    }

    // Затем сохранить cookie со сроком жизни, равным 0
    this.store(0, path, domain, secure);
}

/**
 * Этот статический метод пытается определить, разрешено ли использование cookies
 * в броузере. Возвращает значение true, если разрешено, и false - в противном случае.
```

```
 * Возвращаемое значение true не гарантирует, что сохранение cookies
 * фактически разрешено. Временные cookies сеанса по-прежнему могут быть
 * доступны, даже если этот метод возвращает значение false.
 */
Cookie.enabled = function() {
    // Воспользоваться свойством navigator.cookieEnabled, если оно определено в броузере
    if (navigator.cookieEnabled != undefined) return navigator.cookieEnabled;

    // Если значение уже было помещено в кэш, использовать это значение
    if (Cookie.enabled.cache != undefined) return Cookie.enabled.cache;

    // Иначе создать тестовый cookie с некоторым временем жизни
    document.cookie = "testcookie=test; max-age=10000"; // Установить cookie

    // Теперь проверить - был ли сохранен cookie-файл
    var cookies = document.cookie;
    if (cookies.indexOf("testcookie=test") == -1) {
        // Cookie не был сохранен
        return Cookie.enabled.cache = false;
    }
    else {
        // Cookie был сохранен, поэтому его нужно удалить перед выходом
        document.cookie = "testcookie=test; max-age=0"; // Удалить cookie
        return Cookie.enabled.cache = true;
    }
}
```

19.5. Альтернативы cookies

Cookies имеют пару недостатков, которые осложняют их использование для хранения данных на стороне клиента:

- Размер cookie ограничен значением 4 Кбайт.
- Даже когда cookies используются исключительно для нужд клиентских сценариев, они все равно загружаются на веб-сервер при запросе любых страниц, с которыми они связаны. Если cookies не используются на сервере, они понапрасну расходуют пропускную способность.

Есть две альтернативы использованию cookies. В Internet Explorer и в подключаемом Flash-модуле имеются фирменные (соответственно от Microsoft и Adobe) механизмы сохранения данных на стороне клиента. Хотя они не стандартизованы, тем не менее эти механизмы получили достаточно широкое распространение, а это означает, что, по крайней мере, один из них будет доступен в большинстве основных броузеров. Механизмы сохранения данных, предлагаемые IE и Flash, коротко описываются в следующих разделах, а в конце главы приводится расширенный пример программы, выполняющей сохранение данных с применением механизмов IE, Flash или cookies.

19.5.1. Механизм сохранения userData в IE

Internet Explorer позволяет сохранять информацию на стороне клиента средствами DHTML. Для доступа к этому механизму необходимо заставить элемент (такой как `<div>`) вести себя особым образом. Одно из таких средств – CSS:

```
<!-- Данная таблица стилей определяет класс с именем "persistent" -->
<style>.persistent { behavior:url(#default#userData);}</style>
<!-- Данный элемент <div> является членом этого класса -->
<div id="memory" class="persistent"></div>
```

Поскольку атрибут `behavior` не предусматривается стандартами CSS, остальные броузеры просто игнорируют его. Атрибут `behavior` стиля элемента можно также установить из JavaScript-сценария:

```
var memory = document.getElementById("memory");
memory.style.behavior = "url('#default#userData')";
```

Когда HTML-элементу назначается поведение «userData»[1], для этого элемента становятся доступными новые методы (определяемые атрибутом `behavior`). Чтобы сохранить данные в постоянном хранилище, необходимо установить значения атрибутов элемента методом `setAttribute()`, а затем сохранить эти атрибуты вызовом метода `save()`:

```
var memory = document.getElementById("memory"); // Получить ссылку на элемент,
                                                // сохраняющий данные
memory.setAttribute("username", username);      // Определить данные как атрибуты
memory.setAttribute("favoriteColor", favoriteColor);
memory.save("myPersistentData");                // Сохранить данные
```

Обратите внимание: метод `save()` принимает строковый аргумент – это имя (произвольное), под которым будут сохраняться данные. Чтобы потом прочитать эти данные, необходимо использовать то же самое имя.

Для данных, сохраняемых с помощью механизмов Internet Explorer, можно определить дату окончания срока действия точно так же, как и для cookie-файла. Для этого перед вызовом метода `save()` достаточно установить свойство `expires`. В это свойство должна быть записана строка в той же форме, в которой она возвращается методом `Date.toUTCString()`. Например, чтобы определить срок действия в 10 дней, начиная от текущего момента, в предыдущий фрагмент нужно добавить следующие строки:

```
var now = (new Date()).getTime();                    // Текущий момент
                                                     // в миллисекундах
var expires = now + 10 * 24 * 60 * 60 * 1000;        // 10 суток от текущего
                                                     // момента в миллисекундах
memory.expires = (new Date(expires)).toUTCString(); // Преобразовать в строку
```

Чтобы прочитать сохраненные ранее данные, нужно выполнить те же шаги в обратном порядке: вызвать метод `load()`, чтобы загрузить данные, а затем вызовом `getAttribute()` получить значения атрибутов:

```
var memory = document.getElementById("memory");   // Получить ссылку на элемент,
                                                  // сохраняющий данные
memory.load("myPersistentData");                  // Прочитать данные по имени
```

[1] Поведение userData – лишь один из четырех доступных в Internet Explorer вариантов поведения, связанных с сохранением данных на стороне клиента. Подробное описание механизмов сохранения информации в Internet Explorer вы найдете по адресу: *http://msdn.microsoft.com/workshop/author/persistence/overview.asp*.

```
var user = memory.getAttribute("username");       // Прочитать значения
var color = memory.getAttribute("favoriteColor"); // атрибутов
```

19.5.1.1. Сохранение данных с иерархической структурой

Возможности механизма userData не ограничиваются сохранением и восстановлением значений атрибутов. Любой элемент, для которого определяется поведение userData, имеет полноценный XML-документ, связанный с этим элементом. Применение этого поведения к HTML-элементу создает свойство `XMLDocument` в этом элементе, а значением этого свойства является DOM-объект `Document`.

Для добавления содержимого к этому документу перед вызовом метода `save()` или для извлечения данных после вызова метода `load()` можно использовать DOM-методы (см. главу 15). Например:

```
var memory = document.getElementById("memory"); // Получить ссылку на элемент,
                                                 // сохраняющий данные
var doc = memory.XMLDocument;                    // Получить ссылку на документ
var root = doc.documentElement;                  // Корневой элемент документа
root.appendChild(doc.createTextNode("data here")); // Сохранить текст в документе
```

Благодаря возможности использования XML-документов можно сохранять данные с иерархической структурой, например преобразовать дерево JavaScript-объектов в дерево XML-элементов.

19.5.1.2. Ограничения

Механизм сохранения данных IE позволяет хранить гораздо большие объемы данных, чем cookie-файлы. Каждая страница может сохранить до 64 Кбайт, а каждый веб-сервер – до 640 Кбайт. Сайты в доверенных локальных сетях могут сохранять еще большие объемы. Для конечного пользователя нет документированного способа изменить ограничения на размер хранилища или вообще отключить механизм сохранения.

19.5.1.3. Совместное использование сохраненных данных

Подобно cookies данные, сохраненные с помощью механизмов IE, доступны всем веб-страницам одного каталога. Однако, в отличие от cookies, при использовании механизма IE веб-страница не может получить доступ к данным, сохраненным страницами из родительского каталога. Кроме того, механизм сохранения данных IE не имеет атрибутов, эквивалентных атрибутам cookies `path` и `domain`. По этой причине отсутствует возможность расширить круг страниц, обладающих доступом к одним и тем же данным. Наконец, данные, сохраненные в IE, могут совместно использоваться только страницами из одного каталога и загруженными только с помощью одного и того же протокола. То есть данные, сохраненные страницей, загруженной по протоколу HTTPS, будут недоступны для страниц, загруженных по протоколу HTTP.

19.5.2. Механизм сохранения SharedObject подключаемого Flash-модуля

Начиная с версии 6 подключаемый Flash-модуль позволяет сохранять данные на стороне клиента с помощью класса `SharedObject`, которым можно управлять из

программного ActionScript-кода во Flash-роликах.[1] Чтобы воспользоваться этим механизмом, нужно с помощью программного кода на ActionScript создать объект SharedObject примерно так, как показано ниже. Обратите внимание: для сохраняемых данных необходимо указать имя (как и для cookie):

```
var so = SharedObject.getLocal("myPersistentData");
```

В классе SharedObject отсутствует метод load(), аналогичный методу механизма сохранения IE. Дело в том, что когда создается объект SharedObject, все данные, сохраненные ранее под указанным именем, загружаются автоматически. Все объекты SaredData имеют свойство data. Это свойство ссылается на обычный ActionScript-объект, а собственно данные доступны через свойства этого объекта. Чтобы прочитать или сохранить данные, достаточно просто прочитать или записать значения свойств объекта data:

```
var name = so.data.username;    // Прочитать сохраненные ранее данные
so.data.favoriteColor = "red"; // Записать сохраняемые данные
```

В свойства объекта data можно записывать не только значения элементарных типов, такие как числа или строки, но и такие значения, например, как массивы.

Хотя объект SharedObject не имеет метода save(), зато у него есть метод flush(), который выполняет немедленное сохранение текущего состояния SharedObject. Однако вызывать этот метод совершенно не обязательно: свойства объекта data сохраняются автоматически при выгрузке Flash-ролика. Кроме того, следует отметить, что объект SharedObject не предоставляет возможности определить дату окончания срока действия или время жизни хранимых данных.

Имейте в виду, что весь программный код, продемонстрированный в этом разделе, не является исполняемым броузером JavaScript-кодом – это ActionScript-код, который исполняется Flash-модулем. Если вам потребуется использовать предлагаемый Flash механизм сохранения данных из JavaScript-сценариев, вам придется организовать взаимодействие между JavaScript-сценарием и Flash-модулем. Как это сделать, рассказывается в главе 23. В примере 22.12 демонстрируется порядок использования класса ExternalInterface (доступен в подключаемом Flash-модуле версии 8 или выше), который упрощает вызов ActionScript-методов из JavaScript-сценариев. В примерах 19.3 и 19.4 демонстрируются низкоуровневые механизмы взаимодействия между JavaScript и ActionScript. Методы GetVariable() и SetVariable() из подключаемого объекта Flash-модуля позволяют JavaScript-сценариям получать и записывать значения ActionScript-переменных, а с помощью ActionScript-функции fscommand() можно передать данные в JavaScript-сценарий.

---

[1] Полное описание класса SharedObject и механизма сохранения данных Flash-модуля вы найдете на сайте Adobe по адресу: *http://www.adobe.com/support/flash/action_scripts/local_shared_object/*. О существовании механизма сохранения данных на базе Flash я узнал от Бреда Ньюберга (Brad Neuberg), который впервые начал использовать его из JavaScript-сценариев в своем проекте AMASS (*http://codinginparadise.org/projects/storage/README.html*). К моменту написания этих строек проект продолжал свое развитие; дополнительные сведения вы можете получить на персональной странице Бреда (*http://codinginparadise.org*).

19.5.2.1. Ограничения

По умолчанию Flash-проигрыватель позволяет сохранять до 100 Кбайт данных с одного веб-сайта. Пользователь может менять это значение в пределах от 10 Кбайт до 10 Мбайт. Кроме того, он может вообще снять ограничение или полностью запретить возможность сохранения данных. Если веб-сайт попытается сохранить данные, объем которых превышает установленное ограничение, Flash-проигрыватель запросит у пользователя разрешение на увеличение объема хранилища для данного веб-сайта.

19.5.2.2. Совместное использование сохраненных данных

По умолчанию сохраненные данные доступны только тому Flash-ролику, который их создал. Однако существует возможность ослабить это ограничение и позволить различным роликам из одного каталога или с одного сервера совместно использовать сохраненные данные. Делается это способом, очень похожим на применение атрибута `path` в cookie. При создании объекта `SharedObject` методом `SharedObject.getLocal()` вторым аргументом можно передать путь, который должен представлять собой начальную часть фактического пути из URL-адреса ролика. После этого любой другой ролик, расположенный по тому же пути, сможет получить доступ к данным, сохраненным текущим роликом. Например, в следующем фрагменте создается объект `SharedObject`, который может использоваться всеми Flash-роликами того же веб-сервера:

```
var so = SharedObject.getLocal("My/Shared/Persistent/Data", // Имя объекта
                               "/");                        // Путь
```

При работе с классом `SharedObject` из JavaScript-сценариев вас вряд ли заинтересует возможность совместного использования данных разными Flash-роликами, скорее разными веб-страницами, управляющими одним роликом (см. пример 19.3 в следующем разделе).

19.5.3. Пример: хранимые объекты

Данный раздел завершается расширенным примером, в котором определяется унифицированный прикладной интерфейс доступа к трем механизмам сохранения данных, рассматриваемым в этой главе. В примере 19.3 определяется класс, позволяющий сохранять объекты. Класс `PObject` очень напоминает класс `Cookie` из примера 19.2. Сначала с помощью конструктора `PObject()` создается хранимый объект. Конструктору передаются имя объекта, набор значений по умолчанию и функция-обработчик события `onload`. Конструктор создает новый JavaScript-объект и предпринимает попытку загрузить данные, сохраненные ранее с указанным именем. Если данные обнаружены, выполняется их интерпретация в виде пар имя–значение, после чего эти пары преобразуются в свойства вновь созданного объекта. Если данные не обнаружены, используются значения, заданные как свойства, предлагаемые по умолчанию. В любом случае указанная функция-обработчик вызывается асинхронно, когда данные готовы к использованию.

После вызова обработчика события `onload` хранимые данные становятся доступными в виде свойств объекта `PObject`, как если бы это были свойства обычного JavaScript-объекта. Чтобы сохранить новые данные, сначала нужно установить требуемые свойства объекта `PObject` (логического, числового или строкового ти-

па), а затем вызвать метод `save()` объекта `PObject`, указав при желании время жизни данных (в сутках). Чтобы удалить хранимые данные, следует вызвать метод `forget()` объекта `PObject`.

Определяемый ниже класс `PObject` при работе в IE использует механизм сохранения данных этого броузера. В противном случае он проверяет доступность подходящей версии Flash-модуля и при его наличии использует механизм сохранения Flash-модуля. Если ни один из перечисленных вариантов недоступен, сохранение выполняется с помощью cookies.[1]

Обратите внимание: класс `PObject` допускает сохранение значений только элементарных типов и преобразует значения числовых и логических типов в строки. Есть возможность реализовать сериализацию массивов и объектов в строки, а при загрузке данных преобразовывать эти строки обратно в массивы и объекты (подробнее об этом можно прочитать по адресу: *http://www.json.org*), но в данном примере это не предусмотрено.

Пример 19.3 достаточно большой, но он содержит подробные комментарии, поэтому разобраться в нем не составит большого труда. Обязательно прочитайте вводный комментарий, где описывается класс `PObject` и его прикладной интерфейс (API).

Пример 19.3. PObject.js: хранимые объекты для JavaScript-объекта PObject

```
/**
 * PObject.js: JavaScript-объекты, которые позволяют сохранять данные между
 * сеансами работы с броузером и могут совместно использоваться веб-страницами
 * одного каталога с одного и того же сервера.
 *
 * Данный модуль определяет конструктор PObject(), с помощью которого
 * создается хранимый объект.
 * Объекты PObject имеют два общедоступных метода. Метод save() сохраняет
 * текущие значения свойств объекта, а метод forget() удаляет сохраненные
 * значения свойств объекта. Чтобы определить хранимое свойство в объекте
 * PObject, достаточно просто установить свойство, как если бы это был
 * обычный JavaScript-объект, и затем вызвать метод save(), чтобы сохранить
 * текущее состояние объекта. Вы не должны использовать имена "save"
 * и "forget" для определения своих свойств, точно так же вы не должны
 * использовать имена, начинающиеся с символа $. Объект PObject предполагает,
 * что значения всех свойств будут иметь строковый тип. Хотя при этом
 * допускается сохранять числовые и логические значения, но при получении
 * данных они будут преобразованы в строки.
 *
 * В процессе создания PObject хранимые данные загружаются и сохраняются
 * во вновь созданном объекте в виде обычных JavaScript-свойств, и вы можете
 * использовать PObject точно так же, как обычный JavaScript-объект.
 * Обратите внимание: к тому моменту, когда конструктор PObject() вернет
 * управление, хранимые свойства могут быть еще не готовы к использованию
```

[1] Кроме того, можно определить класс, который задействовал бы cookies как основной механизм сохранения данных, а возможности, предоставляемые IE и Flash-модулем, применялись бы только в том случае, если пользователь заблокировал cookie-файлы.

```
* и вам нужно подождать, пока в виде асинхронного вызова функции-обработчика
* события onload не будет получено извещение о готовности,
* которое передается конструктору.
*
* Конструктор:
*    PObject(name, defaults, onload):
*
* Аргументы:
*
*    name Имя, идентифицирующее хранимый объект. Одна страница может хранить
*         данные в нескольких объектах PObject, и каждый объект PObject
*         доступен для всех страниц из одного каталога, поэтому данное имя
*         должно быть уникальным в пределах каталога. Если этот аргумент
*         содержит значение null или отсутствует, будет использовано
*         имя файла (а не каталога), содержащего веб-страницу.
*
*    defaults Необязательный JavaScript-объект. Если сохраненные ранее
*             значения свойств хранимого объекта найдены не будут (что
*             может произойти, когда объект PObject создается впервые)
*             свойства данного объекта будут скопированы во вновь созданный
*             объект PObject.
*
*    onload Функция, которая вызывается (асинхронно), когда хранимые значения
*           загрузятся в объект PObject и будут готовы к использованию.
*           Данная функция вызывается с двумя аргументами: ссылкой на объект
*           PObject и именем объекта PObject. Эта функция вызывается уже
*           *после* того, как конструктор PObject() вернет управление.
*           До этого момента свойства PObject не должны использоваться.
*
* Метод PObject.save(lifetimeInDays):
*    Сохраняет свойства объекта PObject и гарантирует, их хранение по меньшей
*    мере указанное число суток.
*
* Метод PObject.forget():
*    Удаляет свойства объекта PObject. После этого сохраняет "пустой" объект
*    PObject в хранилище, и если это возможно, определяет, что срок хранения
*    этого объекта уже истек.
*
* Примечания к реализации:
*
* Этот модуль определяет единый прикладной интерфейс (API) объекта
* PObject, который предоставляет три различные реализации этого
* интерфейса. В Internet Explorer используется механизм сохранения
* "UserData". В любых других броузерах, в которых установлен Flash-модуль
* от Adobe, используется механизм сохранения SharedObject. В броузерах,
* отличных от IE и не имеющих Flash-модуля расширения, используется
* реализация на базе cookie. Обратите внимание: Flash-реализация не
* поддерживает возможность определения даты окончания срока действия
* хранимых данных, поэтому в данной реализации хранимые данные существуют
* до тех пор, пока не будут явно удалены.
*
* Совместное использование объектов PObject:
*
* Данные, сохраняемые в объекте PObject из одной страницы, будут доступны
```

```
 * из других страниц, расположенных в том же каталоге на том же веб-сервере.
 * При использовании реализации на базе cookie страницы, расположенные
 * во вложенных каталогах, смогут читать (но не писать) свойства объектов
 * PObject, созданных страницами из родительского каталога. В случае
 * реализации на базе Flash-модуля любые страницы с того же веб-сервера смогут
 * совместно использовать данные, если задействуют модифицированную
 * версию этого модуля.
 *
 * Различные броузеры сохраняют свои cookie-файлы в разных хранилищах,
 * потому данные, сохраненные как cookie-файлы в одном броузере, будут
 * недоступны в других броузерах. Однако если два броузера применяют
 * одну и ту же установленную копию Flash-модуля, они смогут совместно
 * использовать данные, сохраненные с помощью реализации на базе Flash-модуля.
 *
 * Сведения о безопасности:
 *
 * Данные, сохраняемые в виде объекта PObject, хранятся в незашифрованном
 * виде на жестком диске локальной системы. Приложения, работающие на этом
 * компьютере, могут прочитать эти данные, поэтому PObject не подходит
 * для хранения частной информации, такой как номера кредитных карт,
 * паролей или номеров банковских счетов.
 */
// Это конструктор
function PObject(name, defaults, onload) {
    if (!name) { // Если имя не задано, использовать последний компонент URL
        name = window.location.pathname;
        var pos = name.lastIndexOf("/");
        if (pos != -1) name = name.substring(pos+1);
    }
    this.$name = name; // Запомнить имя

    // Вызов делегированного частного метода init(),
    // определяемого реализацией.
    this.$init(name, defaults, onload);
}

// Сохраняет текущее состояние объекта PObject на заданное число суток.
PObject.prototype.save = function(lifetimeInDays) {
    // Для начала преобразовать свойства объекта в одну строку
    var s = "";                                 // Изначально строка пустая
    for(var name in this) {                     // Цикл по свойствам объекта
        if (name.charAt(0) == "$") continue; // Пропустить частные свойства,
                                                // имена которых начинаются с $
        var value = this[name];                 // Получить значение свойства
        var type = typeof value;                // Получить тип свойства
        // Пропустить свойства-объекты и функции
        if (type == "object" || type == "function") continue;
        if (s.length > 0) s += "&";             // Отделить свойства знаком &
        // Добавить имя свойства и кодированное значение
        s += name + ':' + encodeURIComponent(value);
    }

    // Затем вызвать делегированный метод, определяемый реализацией,
    // для фактического сохранения строки.
    this.$save(s, lifetimeInDays);
```

```
};
POBject.prototype.forget = function() {
    // Сначала удалить сериализуемые свойства данного объекта с помощью
    // тех же критериев отбора свойств, что использовались в методе save().
    for(var name in this) {
        if (name.charAt(0) == '$') continue;
        var value = this[name];
        var type = typeof value;
        if (type == "function" || type == "object") continue;
        delete this[name]; // Удалить свойство
    }

    // Затем стереть сохраненные ранее данные, записав пустую строку
    // и установив время жизни равным 0.
    this.$save("", 0);
};

// Преобразовать строку в пары имя/значение и превратить их в свойства объекта this.
// Если строка не определена или пустая, скопировать свойства из объекта по умолчанию.
// Данный частный метод используется реализациями $init().
POBject.prototype.$parse = function(s, defaults) {
    if (!s) { // Если строка отсутствует, использовать объект по умолчанию
        if (defaults) for(var name in defaults) this[name] = defaults[name];
        return;
    }

    // Пары имя-значение отделяются символом амперсанда, а имя и значение
    // внутри каждой пары - символом двоеточия.
    // Все преобразования выполняются с помощью метода split().
    var props = s.split('&');// Преобразовать строку в массив пар имя/значение
    for(var i = 0; i < props.length; i++) { // Цикл по парам имя/значение
        var p = props[i];
        var a = p.split(':'); // Разбить каждую пару по символу двоеточия
        this[a[0]] = decodeURIComponent(a[1]); // Декодировать и сохранить
                                               // в виде свойства
    }
};

/*
 * Далее находится часть модуля, зависящая от реализации.
 * Для каждой из реализаций определяется метод $init(), который загружает
 * хранимые данные, и метод $save(), который эти данные сохраняет.
 */

// Определить, исполняется ли данный программный код под управлением IE,
// если нет - проверить, установлен ли Flash-модуль и имеет ли он
// достаточно высокий номер версии
var isIE = navigator.appName == "Microsoft Internet Explorer";
var hasFlash7 = false;
if (!isIE && navigator.plugins) { // Если используется архитектура
                                  // модуля для Netscape
    var flashplayer = navigator.plugins["Shockwave Flash"];
    if (flashplayer) { // Если Flash-модуль установлен
        // Извлечь номер версии
        var flashversion = flashplayer.description;
```

```
        var flashversion = flashversion.substring(flashversion.search("\\d"));
        if (parseInt(flashversion) >= 7) hasFlash7 = true;
    }
}

if (isIE) { // Если броузер - IE
    // Делегировать конструктору PObject() эту функцию инициализации
    PObject.prototype.$init = function(name, defaults, onload) {
        // Создать скрытый элемент с поведением userData для сохранения данных
        var div = document.createElement("div"); // Создать тег <div>
        this.$div = div;                         // Запомнить ссылку на него
        div.id = "PObject" + name;               // присвоить имя
        div.style.display = "none";              // Сделать невидимым

        // Далее следует специфичная для IE реализация сохранения.
        // Поведение "userData" добавляет методы getAttribute(),
        // setAttribute(), load() и save() к элементу <div>.
        // они потребуются позже.
        div.style.behavior = "url('#default#userData')";
        document.body.appendChild(div); // Добавить элемент в документ

        // Теперь нужно получить сохраненные ранее данные.
        div.load(name); // Загрузить данные, сохраненные под указанным именем
        // Данные представляют собой набор атрибутов. Нам нужен лишь один
        // из них. Мы произвольно выбрали для атрибута имя "data".
        var data = div.getAttribute("data");

        // Преобразовать полученные данные в свойства объекта
        this.$parse(data, defaults);

        // Если была определена функция обратного вызова onload, запланировать
        // асинхронный вызов этой функции после того, как конструктор
        // PObject() закончит работу.
        if (onload) {
            var pobj = this; // Во вложенных функциях нельзя использовать
                             // ключевое слово "this"
            setTimeout(function() { onload(pobj, name);}, 0);
        }
    }

    // Сохраняет текущее состояние хранимого объекта
    PObject.prototype.$save = function(s, lifetimeInDays) {
        if (lifetimeInDays) { // Если время жизни указано, преобразовать
                              // его в конечную дату срока действия
            var now = (new Date()).getTime();
            var expires = now + lifetimeInDays * 24 * 60 * 60 * 1000;
            // Установить конечную дату срока действия как
            // строковое свойство элемента <div>
            this.$div.expires = (new Date(expires)).toUTCString();
        }

        // Теперь сохранить данные
        this.$div.setAttribute("data", s); // Установить значение текстового
                                           // атрибута элемента <div>
        this.$div.save(this.$name);        // И сохранить этот атрибут
    };
}
```

```
else if (hasFlash7) { // Реализация на базе Flash
    PObject.prototype.$init = function(name, defaults, onload) {
        var moviename = "PObject_" + name; // идентификатор тега <embed>
        var url = "PObject.swf?name=" + name; // URL-адрес файла ролика

        // Когда Flash-проигрыватель запустится и получит наши данные,
        // он пошлет извещение с помощью FSCommand. Поэтому
        // нам нужно определить обработчик этого события.
        var pobj = this; // Для использования во вложенной функции
        // Flash требует, чтобы имя функции было глобальным
        window[moviename + "_DoFSCommand"] = function(command, args) {
            // Теперь известно, что данные загружены Flash-модулем,
            // следовательно, можно прочитать их
            var data = pobj.$flash.GetVariable("data")
            pobj.$parse(data, defaults);// Преобразовать данные в свойства
                                        // или скопировать данные по умолчанию
            if (onload) onload(pobj, name); // Вызвать обработчик onload,
                                            // если он определен
        };

        // Создать тег <embed> для хранения Flash-ролика. Использование тега
        // <object> точнее соответствует стандартам, но он вызывает проблемы
        // при приеме FSCommand. Обратите внимание: мы никогда не используем
        // Flash в IE, что существенно упрощает реализацию.
        var movie = document.createElement("embed"); // Элемент с роликом
        movie.setAttribute("id", moviename);         // Идентификатор элемента
        movie.setAttribute("name", moviename);       // И имя
        movie.setAttribute("type", "application/x-shockwave-flash");
        movie.setAttribute("src", url); // Это URL-адрес ролика
        // Сделать ролик малозаметным и переместить его в правый верхний угол
        movie.setAttribute("width", 1); // Если установить равным 0,
                                        // это работать не будет
        movie.setAttribute("height", 1);
        movie.setAttribute("style", "position:absolute; left:0px; top:0px;");
        document.body.appendChild(movie); // Добавить ролик в документ
        this.$flash = movie; // И запомнить для дальнейшего использования
    };

    PObject.prototype.$save = function(s, lifetimeInDays) {
        // Чтобы сохранить данные, достаточно определить их как переменные
        // Flash-ролика. ActionScript-код ролика сохранит их.
        // Обратите внимание: механизм сохранения Flash-модуля
        // не поддерживает возможность определения времени жизни.
        this.$flash.SetVariable("data", s); // Попросить Flash сохранить текст
    };
}
else { /* Если это не IE и отсутствует модуль Flash 7, использовать cookies */
    PObject.prototype.$init = function(name, defaults, onload) {
        var allcookies = document.cookie;   // Получить все cookies
        var data = null;                    // Предположить отсутствие cookie
        var start = allcookies.indexOf(name + '='); // Отыскать начало cookie
        if (start != -1) {                  // Найдено
            start += name.length + 1;       // Пропустить имя cookie
            var end = allcookies.indexOf(';', start);// Отыскать конец cookie
            if (end == -1) end = allcookies.length;
```

```
            data = allcookies.substring(start, end); // Извлечь данные
        }
        this.$parse(data, defaults); // Преобразовать значение cookie в свойства
        if (onload) {                // Вызвать асинхронно обработчик onload
            var pobj = this;
            setTimeout(function() { onload(pobj, name); }, 0);
        }
    };

    PObject.prototype.$save = function(s, lifetimeInDays) {
        var cookie = this.$name + '=' + s; // Имя и значение cookie
        if (lifetimeInDays != null)        // Добавить конечную дату
            cookie += "; max-age=" + (lifetimeInDays*24*60*60);
        document.cookie = cookie;          // Сохранить cookie
    };
}
```

19.5.3.1. ActionScript-код для работы с механизмом сохранения Flash

Программный код примера 19.3 не полон, поскольку его реализация сохранения данных на базе механизма Flash предполагает использование Flash-ролика с именем *PObject.swf*. Этот ролик не более чем скомпилированный ActionScript-файл. ActionScript-код приводится в примере 19.4.

Пример 19.4. ActionScript-код для сохранения данных на базе механизма Flash

```
class PObject {
    static function main() {
        // Объект SharedObject существует в Flash 6, но он не защищен
        // от атак типа межсайтовый скриптинг, поэтому нам нужен
        // Flash-проигрыватель версии 7.
        var version = getVersion();
        version = parseInt(version.substring(version.lastIndexOf(" ")));
        if (isNaN(version) || version < 7) return;

        // Создать объект SharedObject, который будет содержать хранимые
        // данные. Имя объекта передается в строке URL-адреса ролика
        // примерно так: PObject.swf?name=name
        _root.so = SharedObject.getLocal(_root.name);

        // Получить начальные данные и сохранить их в _root.data.
        _root.data = _root.so.data.data;

        // Следить за переменной. При изменении - сохранить ее новое значение.
        _root.watch("data", function(propName, oldValue, newValue) {
                    _root.so.data.data = newValue;
                    _root.so.flush();
                   });

        // Известить JavaScript-код, что хранимые данные получены.
        fscommand("init");
    }
}
```

Программный ActionScript-код достаточно прост. Он начинается с создания объекта `SharedObject`, используя при этом имя, заданное (из JavaScript-сценария)

в виде строки запроса в URL-адресе объекта ролика. При создании объекта `SharedObject` загружаются хранимые данные, которые в данном случае представлены в виде единственной строки. Эта строка передается обратно JavaScript-сценарию с помощью функции `fscommand()`, которая вызывает определенный в сценарии обработчик `doFSCommand`. Кроме того, ActionScript-код устанавливает функцию-обработчик, которая будет вызываться при изменении свойства `data` корневого объекта. Изменение значения свойства `data` из JavaScript-кода производится с помощью функции `SetVariable()`, а этот ActionScript-обработчик вызывается в ответ и сохраняет данные.

ActionScript-код из файла *PObject.as*, который приводится в примере 19.4, необходимо скомпилировать в файл *PObject.swf*, прежде чем он сможет использоваться с Flash-плеером. Сделать это можно с помощью свободно распространяемого компилятора ActionScript с именем *mtasc* (доступен по адресу: *http://www.mtasc.org*). Вызывается компилятор следующим образом:

```
mtasc -swf PObject.swf -main -header 1:1:1 PObject.as
```

Результатом работы компилятора *mtasc* является файл формата SWF, который будет вызывать метод `PObject.main()` из первого кадра ролика. Однако если вы пользуетесь интегрированной средой разработки Flash, то можете явно определить вызов метода `PObject.main()` из первого кадра. Как вариант – можно просто скопировать код из метода `main()` и вставить его в первый кадр.

19.6. Хранимые данные и безопасность

В начале примера 19.3 были отмечены некоторые проблемы с безопасностью, которые следует иметь в виду, сохраняя данные на стороне клиента. Запомните: любые данные, сохраняемые на жестком диске клиента, записываются в открытом незашифрованном виде. Поэтому они могут быть доступны как любопытствующим пользователям, имеющим доступ к компьютеру, так и злонамеренному программному обеспечению (например, разнообразным шпионским программам), функционирующему на этом компьютере. Поэтому механизмы сохранения данных на стороне пользователя никогда не должны применяться для хранения частной информации: паролей, номеров банковских счетов и тому подобного. Помните: если пользователь ввел какую-либо информацию в поле формы при взаимодействии с вашим веб-сайтом, это еще не значит, что он хотел бы сохранить копию этих данных у себя на жестком диске. Как пример, представьте себе номер кредитной карты. Это частная информация, которую люди прячут от посторонних глаз в своих бумажниках. Сохранять такого рода информацию на стороне клиента – все равно, что написать номер кредитной карты на листочке и приклеить этот листочек на клавиатуру компьютера пользователя. Поскольку шпионские программы получили широкое распространение (по крайней мере, в Windows), это все равно, что отправить эту информацию в открытом виде через Интернет.

Кроме того, следует учесть, что многие веб-сайты используют cookies и другие механизмы сохранения данных для отслеживания перемещений пользователей Всемирной паутины, что вызывает недоверие у последних. Задействуйте механизмы хранения данных, описываемые в этой главе, чтобы сделать ваш сайт более удобным, но не применяйте их для сбора сведений о пользователях.

20

Работа с протоколом HTTP

Протокол передачи гипертекста (Hypertext Transfer Protocol, HTTP) определяет, как веб-броузеры должны запрашивать документы, как они должны передавать информацию веб-серверам и как веб-серверы должны отвечать на эти запросы и передачи. Вполне очевидно, что веб-броузеры очень много работают с протоколом HTTP. Тем не менее, как правило, сценарии не работают с протоколом HTTP, когда пользователь щелкает по ссылке, отправляет форму или вводит URL в адресной строке. В то же время обычно, хотя и не всегда, JavaScript-код *способен* работать с протоколом HTTP.

HTTP-запросы могут инициироваться, когда сценарий устанавливает значение свойства `location` объекта `Window` или вызывает метод `submit()` объекта `Form`. В обоих случаях броузер загружает в окно новую страницу, затирая любой исполнявшийся сценарий. Такого рода взаимодействие с протоколом HTTP может быть вполне оправданным в веб-страницах, состоящих из нескольких фреймов, но в этой главе мы будем говорить совсем о другом. Здесь мы рассмотрим такое взаимодействие JavaScript-кода с веб-сервером, при котором веб-броузер не перезагружает текущую веб-страницу.

Теги `<img>`, `<frame>` и `<script>` имеют свойство `src`. Когда сценарий записывает в это свойство URL-адрес, инициируется HTTP-запрос GET и выполняется загрузка содержимого с этого URL-адреса. Таким образом, сценарий может отправлять информацию веб-серверу, добавляя ее в виде строки запроса в URL-адрес изображения и устанавливая свойство `src` элемента `<img>`. В ответ на этот запрос веб-сервер должен вернуть некоторое изображение, которое, например, может быть невидимым: прозрачным и размером 1×1 пиксел.[1]

---

[1] Такие изображения иногда называют *веб-жучками* (*web bugs*). Они пользуются дурной славой из-за проблем с безопасностью, т. к. могут применяться для обмена информацией (подсчета числа посещений и анализа трафика) со сторонним сервером (не тем, откуда была загружена страница). Если же веб-страница устанавливает свойство `src` изображения для передачи информации обратно на тот сервер, с которого она была загружена, особых проблем с безопасностью не возникает.

Теги <iframe> достаточно недавно появились в HTML и они более универсальны, чем теги <img>, т. к. позволяют веб-серверу вернуть результат не в виде двоичного файла с изображением, а в удобочитаемом виде, который может быть проверен сценарием. При использовании тега <iframe> сценарий сначала добавляет в URL-адрес информацию, предназначенную для веб-сервера, а затем записывает этот URL-адрес в свойство src тега <iframe>. Сервер создает HTML-документ, содержащий ответ на запрос, и отправляет его обратно веб-броузеру, который выводит ответ в теге <iframe>. При этом элемент <iframe> необязательно должен быть видимым для пользователя – он может быть сокрыт, например средствами таблиц стилей. Сценарий может проанализировать ответ сервера, выполнив обход документа в элементе <iframe>. Обратите внимание: взаимодействие с документом ограничивается политикой общего происхождения, о которой рассказывается в разделе 13.8.2.

Даже изменение свойства src тега <script> может использоваться для инициирования динамического HTTP-запроса. Использование тегов <script> для взаимодействия с протоколом HTTP выглядит особенно привлекательно, потому что когда ответ сервера принимает форму JavaScript-кода, он не требует дополнительного анализа – интерпретатор JavaScript просто исполняет его.

Несмотря на то, что существует возможность использования тегов <img>, <iframe> и <script> для взаимодействия с протоколом HTTP, реализовать такую возможность на практике переносимым образом гораздо сложнее, чем это выглядит на словах, и в этой главе мы сосредоточимся на другом, более мощном способе достижения тех же результатов. Объект XMLHttpRequest прекрасно поддерживается всеми современными броузерами и предоставляет полный доступ к протоколу HTTP, включая возможность отправлять запросы методами POST и HEAD в дополнение к обычному запросу методом GET. Объект XMLHttpRequest может возвращать ответ веб-сервера синхронно или асинхронно, в виде простого текста или в виде DOM-документа. Несмотря на свое название объект XMLHttpRequest не ограничивается использованием XML-документов – он в состоянии принимать любые текстовые документы. Объект XMLHttpRequest является ключевым элементом архитектуры веб-приложений, известной как Ajax[1]. Об Ajax-приложениях мы поговорим после ознакомления с тем, как работает объект XMLHttpRequest.

В конце главы мы вернемся к теме использования тега <script> для организации взаимодействия с протоколом HTTP, и там я продемонстрирую, как можно применить эту методику, когда объект XMLHttpRequest недоступен.

20.1. Использование объекта XMLHttpRequest

Процесс взаимодействия с протоколом HTTP с использованием объекта XMLHttpRequest делится на три этапа:

- Создание объекта XMLHttpRequest.

---

[1] Эта глава – лишь введение в предмет; исчерпывающее описание архитектуры Ajax с детальными примерами реализации можно найти, например, в книгах:
Закас, Мак-Пик, Фосетт «Ajax для профессионалов». – Пер. с англ. – СПб.: Символ-Плюс, 2007;
Дари, Бринзаре, Черчез-Тоза, Бусика «AJAX и PHP. Разработка динамических веб-приложений». – Пер. с англ. – СПб.: Символ-Плюс, 2007. – *Примеч. науч. ред.*

- Определение и передача HTTP-запроса на веб-сервер.
- Синхронный или асинхронный прием ответа сервера.

Каждый из этих этапов более подробно рассматривается в следующих подразделах.

Все примеры этой главы представляют собой часть одного большого модуля. В них определяются вспомогательные функции, входящие в пространство имен `HTTP` (см. главу 10). Однако в приведенных здесь примерах вы не найдете программный код, фактически создающий пространство имен. В пакет с примерами, который можно загрузить с сайта издательства, входит файл с именем `http.js`, который включает в себя программный код создания пространства имен, но вы можете в рассматриваемые здесь примеры просто добавить одну строку:

```
var HTTP = {};
```

20.1.1. Создание объекта запроса

Объект `XMLHttpRequest` никогда не стандартизировался, и процесс его создания в Internet Explorer отличается от такового в других платформах. (К счастью, прикладной интерфейс для работы с объектом `XMLHttpRequest` после его создания одинаков для всех платформ.)

В большинстве броузеров объект `XMLHttpRequest` создается простым вызовом конструктора:

```
var request = new XMLHttpRequest();
```

В IE до появления версии 7 конструктор `XMLHttpRequest()` попросту отсутствовал. В IE 5 и 6 `XMLHttpRequest` представляет собой объект ActiveX и должен создаваться обращением к конструктору `ActiveXObject()`, которому передается имя создаваемого объекта:

```
var request = new ActiveXObject("Msxml2.XMLHTTP");
```

К сожалению, в разных версиях библиотеки XML HTTP компании Microsoft объект имеет различные имена. В зависимости от версии библиотеки, установленной у клиента, иногда приходится использовать следующий программный код для создания объекта:

```
var request = new ActiveXObject("Microsoft.XMLHTTP");
```

Пример 20.1 представляет собой платформонезависимую вспомогательную функцию с именем `HTTP.newRequest()`, которая создает объекты `XMLHttpRequest`.

Пример 20.1. Вспомогательная функция HTTP.newRequest( )

```
// Попробуем использовать следующие функции, создающие объект XMLHttpRequest.
HTTP._factories = [
    function() { return new XMLHttpRequest(); },
    function() { return new ActiveXObject("Msxml2.XMLHTTP"); },
    function() { return new ActiveXObject("Microsoft.XMLHTTP"); }
];

// Когда будет обнаружена работоспособная функция, она будет сохранена здесь.
HTTP._factory = null;
```

```
// Создает и возвращает новый объект XMLHttpRequest.
//
// При первом обращении к функции опробываются все функции из списка, пока
// не будет найдена та, что вернет непустое значение и не возбудит исключение.
// После того как будет обнаружена работоспособная функция, ссылка на нее
// запоминается для последующего использования.
//
HTTP.newRequest = function() {
    if (HTTP._factory != null) return HTTP._factory();

    for(var i = 0; i < HTTP._factories.length; i++) {
        try {
            var factory = HTTP._factories[i];
            var request = factory();
            if (request != null) {
                HTTP._factory = factory;
                return request;
            }
        }
        catch(e) {
            continue;
        }
    }

    // Если попав сюда, сценарию не удалось обнаружить подходящую функцию для создания
    // объекта, необходимо возбудить исключение в этом и всех последующих вызовах.
    HTTP._factory = function() {
        throw new Error("Объект XMLHttpRequest не поддерживается");
    }
    HTTP._factory(); // Возбудить исключение
}
```

20.1.2. Отправка запроса

После того как объект `XMLHttpRequest` создан, начинается следующий этап – отправка запроса веб-серверу. Этот процесс сам по себе также состоит из нескольких этапов. В первую очередь нужно вызвать метод `open()`, которому передается URL запроса и *метод* выполнения HTTP-запроса. Большая часть HTTP-запросов выполняется методом GET, который просто загружает содержимое по заданному URL-адресу. Другой, не менее полезный метод – POST; этот метод используется в основном HTML-формами: он позволяет включать в текст запроса имена и значения переменных. Еще один интересный метод – HEAD: он просто запрашивает у сервера заголовки, соответствующие заданному URL-адресу. Это позволяет сценариям проверять, например, время последнего изменения документа без загрузки содержимого этого документа. Указать метод и URL-адрес запроса можно следующим образом:

```
request.open("GET", url, false);
```

По умолчанию метод `open()` настраивает объект `XMLHttpRequest` на выполнение асинхронного запроса. Если передать ему в третьем аргументе значение `false`, запрос будет выполнен синхронно. Вообще, предпочтительнее использовать

асинхронные запросы, но синхронные запросы выполняются проще, поэтому наше рассмотрение мы начнем с них.

Помимо третьего необязательного аргумента метод `open()` может принимать имя и пароль в четвертом и пятом аргументах. Они используются для выполнения запроса к серверу, требующему авторизации.

Метод `open()` не отправляет запрос, он просто сохраняет свои аргументы для последующего использования, когда будет производиться фактическая отправка запроса. Прежде чем отправить запрос, необходимо настроить некоторые заголовки запроса. Вот несколько примеров:[1]

```
request.setRequestHeader("User-Agent", "XMLHttpRequest");
request.setRequestHeader("Accept-Language", "en");
request.setRequestHeader("If-Modified-Since", lastRequestTime.toString());
```

Обратите внимание: веб-броузер автоматически добавляет к создаваемому запросу все необходимые cookies. Явно настраивать заголовок `"Cookie"` может потребоваться только при необходимости отправить на сервер подложный cookie.

Наконец, после создания объекта запроса вызовом метода `open()` и установки необходимых заголовков можно выполнить отправку запроса:

```
request.send(null);
```

В качестве аргумента функции `send()` передается тело запроса. Для HTTP-запросов GET всегда используется значение `null`. Однако для запросов POST аргумент должен содержать данные формы, отправляемой на сервер (см. пример 20.5). Пока мы просто будем передавать значение `null`. (Обратите внимание: значение `null` должно передаваться обязательно. Объект `XMLHttpRequest` является клиентским, по крайней мере в броузере Firefox, его методы не допускают отсутствия аргументов, что вполне допустимо в обычных JavaScript-функциях.)

20.1.3. Получение синхронного ответа

Объект `XMLHttpRequest` хранит не только информацию о HTTP-запросе, но и ответ сервера. Если методу `open()` третьим аргументом передается значение `false`, метод `send()` выполнит запрос синхронно: он не вернет управление до тех пор, пока не будет получен ответ сервера.[2]

Метод `send()` не возвращает код состояния. После того как он вернет управление, можно проверить код состояния HTTP, возвращаемый сервером в свойстве `status` объекта запроса. Возможные значения кода состояния определяются протоколом HTTP. Код состояния 200 означает успешное завершение запроса и доступность ответа. В то же время код состояния 404 означает ошибку «не найдено», которая возникает в случае, когда указанный URL-адрес не существует.

---

[1] Подробное описание протокола HTTP выходит за рамки темы этой книги. За дополнительной информацией об этих и других заголовках, используемых при выполнении HTTP-запросов, обращайтесь к техническому описанию протокола HTTP.

[2] Объект `XMLHttpRequest` обладает поистине удивительными возможностями, но его прикладной программный интерфейс продуман недостаточно. Например, логическое значение, определяющее синхронное или асинхронное поведение, в действительности должно было бы быть аргументом метода `send()`.

Объект XMLHttpRequest возвращает ответ сервера в виде строки, доступной через свойство responseText объекта запроса. Если ответ представляет собой XML-документ, к нему можно обращаться как к DOM-объекту Document через свойство responseXML. Обратите внимание: чтобы объект XMLHttpRequest воспринял и преобразовал ответ сервера в объект Document, сервер должен явно идентифицировать его как XML-документ, указав MIME-тип "text/xml".

Когда запрос выполняется синхронно, программный код, следующий за вызовом метода send(), обычно выглядит как-то так:

```
if (request.status == 200) {
    // Ответ сервера получен. Отобразить текст ответа.
    alert(request.responseText);
}
else {
    // Что-то пошло не так. Отобразить код ошибки и сообщение.
    alert("Error " + request.status + ": " + request.statusText);
}
```

Помимо кода состояния и ответа сервера в виде текста или документа объект XMLHttpRequest предоставляет доступ к HTTP-заголовкам, полученным от сервера. Метод getAllResponseHeaders() возвращает заголовки ответа в виде одного сплошного блока текста, а метод getResponseHeader() возвращает значение заголовка по его имени. Например:

```
if (request.status == 200) { // Убедиться в отсутствии ошибок
    // Убедиться, что ответ - это XML-документ
    if (request.getResponseHeader("Content-Type") == "text/xml") {
        var doc = request.responseXML;
        // Теперь обработать полученный документ
    }
}
```

При использовании объекта XMLHttpRequest в синхронном режиме существует одна серьезная проблема: если веб-сервер не ответит на запрос, метод send() окажется заблокированным на достаточно продолжительное время. Исполнение JavaScript-сценария прекратится, создавая ощущение, что веб-броузер «повис» (разумеется, это во многом зависит от типа платформы). Когда сервер прекращает процесс передачи обычной страницы, пользователь может просто щелкнуть на кнопке Остановить и попробовать перейти по другой ссылке или ввести другой URL-адрес. Однако на объект XMLHttpRequest кнопка Остановить никакого воздействия не оказывает. Метод send() не предоставляет возможности определить максимальное время ожидания, а однопоточная модель исполнения сценариев в JavaScript не позволяет прервать работу объекта XMLHttpRequest в синхронном режиме после того как запрос отправлен.

Решение этой проблемы заключается в использовании объекта XMLHttpRequest в асинхронном режиме.

20.1.4. Обработка асинхронного ответа

Чтобы использовать объект XMLHttpRequest в асинхронном режиме, необходимо передать методу open() в третьем аргументе значение true (или просто опустить

третий аргумент, поскольку значение `true` используется по умолчанию). В этом случае метод `send()` отправит запрос серверу и сразу же вернет управление. Когда придет ответ от сервера, он будет доступен через те же свойства объекта `XMLHttpRequest`, которые были описаны выше.

Асинхронный ответ от сервера – это как асинхронный щелчок мыши, сделанный пользователем: вам потребуется извещение, сообщающее об этом. Роль такого извещения может выполнить обработчик события. В случае объекта `XMLHttpRequest` такой обработчик события устанавливается в свойство `onreadystatechange`. Как следует из имени свойства, функция-обработчик вызывается при изменении значения свойства `readyState`. Свойство `readyState` – это целое число, которое определяет код состояния HTTP-запроса, а его возможные значения перечислены в табл. 20.1. Объект `XMLHttpRequest` не определят символических констант ни для одного из пяти значений, перечисленных в таблице.

Таблица 20.1. Значения свойства readyState объекта XMLHttpRequest

readyState	Значение
0	Метод `open()` еще не вызывался
1	Метод `open()` уже был вызван, но метод `send()` еще не вызывался
2	Метод `send()` был вызван, но ответ от сервера еще не получен
3	Идет прием данных от сервера. Значение 3 свойства `readyState` в Firefox и Internet Explorer отличаются; подробности см. в разделе 20.1.4.1
4	Ответ сервера получен полностью[a]

[a] Запрос успешно завершен. – *Примеч. науч. ред.*

Поскольку объект `XMLHttpRequest` имеет всего один обработчик события, он вызывается для обработки всех возможных событий. Обычно обработчик `onreadystatechange` вызывается один раз после вызова метода `open()` и один раз после вызова метода `send()`. Еще раз он вызывается, когда от сервера начинает поступать ответ, и последний раз – когда ответ сервера полностью принят. В отличие от большинства событий в клиентском JavaScript, обработчику `onreadystatechange` не передается объект события. Чтобы определить причину вызова обработчика, необходимо проверить свойство `readyState` объекта `XMLHttpRequest`. К сожалению, обработчику не передается даже сам объект `XMLHttpRequest`, поэтому необходимо определять функцию-обработчик в той области видимости, откуда ей будет доступен объект запроса. Типичный обработчик асинхронного запроса выглядит примерно следующим образом:

```
// Создать XMLHttpRequest с помощью описанной ранее функции
var request = HTTP.newRequest();

// Зарегистрировать обработчик события для приема асинхронных извещений.
// Этот код выполняет обработку ответа и размещается во вложенной функции
// еще до того, как будет отправлен запрос.
request.onreadystatechange = function() {
    if (request.readyState == 4) {       // Если прием запроса завершился
        if (request.status == 200)       // Если запрос увенчался успехом
            alert(request.responseText); // отобразить ответ сервера
```

```
    }
}

// Создать запрос GET для заданного URL-адреса. Третий аргумент опущен,
// поэтому запрос будет выполнен асинхронно
request.open("GET", url);

// Здесь в случае необходимости можно было бы определить дополнительные
// заголовки в запросе.

// Передать запрос. Поскольку это запрос GET, в качестве тела запроса
// передается значение null. Так как это асинхронный запрос, метод send()
// не блокируется и сразу же возвращает управление.
request.send(null);
```

20.1.4.1. Дополнительные замечания о значении 3 свойства readyState

Объект `XMLHttpRequest` еще не стандартизован, поэтому броузеры по-разному обрабатывают значение 3 свойства `readyState`. Например, при загрузке достаточно длинного ответа броузер Firefox несколько раз вызывает обработчик события `onreadystatechange` для значения 3 в свойстве `readyState` с целью обеспечить обратную связь в процессе загрузки. Сценарии могут использовать это обстоятельство для демонстрации пользователю процесса загрузки. С другой стороны, Internet Explorer очень точно интерпретирует имя обработчика события и вызывает его только в случае фактического изменения значения свойства `readyState`. Это означает, что в Internet Explorer обработчик вызывается всего один раз для значения 3 в свойстве `readyState` независимо от того, как долго продолжается загрузка документа.

Броузеры также по-разному реагируют на значение 3 в свойстве `readyState`. Несмотря на то, что значение 3 означает, что какая-то часть ответа уже принята, тем не менее в документации компании Microsoft к объекту `XMLHttpRequest` явно указывается, что в этом состоянии обращение к свойству `responseText` рассматривается как ошибка. В других броузерах, похоже, свойство `responseText` возвращает ту часть ответа, которая уже доступна.

К сожалению, ни один из основных производителей броузеров не предоставил адекватную документацию к своему объекту `XMLHttpRequest`. До тех пор пока `XMLHttpRequest` не будет стандартизован или хотя бы достаточно ясно документирован, лучше всего игнорировать любые значения `readyState`, отличные от 4.

20.1.5. Безопасность объекта XMLHttpRequest

Будучи субъектом политики общего происхождения (см. раздел 13.8.2), объект `XMLHttpRequest` может отправлять HTTP-запросы только тому серверу, откуда был получен документ, использующий этот объект. Это вполне разумное ограничение, но его можно преодолеть, если на стороне сервера разместить сценарий, выполняющий функции прокси, который будет получать содержимое URL-адресов, расположенных за пределами сайта.

Это ограничение безопасности `XMLHttpRequest` имеет одно очень важное следствие: объект `XMLHttpRequest` выполняет HTTP-запросы и не может работать с другими схемами URL-адресации. Например, он не в состоянии работать с такими префиксами URL-адреса, как `file://`. Это значит, что нет никакой возможности прове-

рить работоспособность сценария, использующего объект XMLHttpRequest в локальной файловой системе. Вам придется загрузить тестовый сценарий на веб-сервер (или запустить веб-сервер на своем локальном компьютере). Чтобы сценарий мог выполнить HTTP-запрос, он должен быть загружен броузером через HTTP.

20.2. Примеры и утилиты с объектом XMLHttpRequest

В начале этой главы был представлен пример вспомогательной функции HTTP.newRequest(), которая позволяет получить объект XMLHttpRequest в любом броузере. Аналогичным образом с помощью других вспомогательных функций можно существенно упростить работу с объектом XMLHttpRequest. В следующих подразделах приводятся примеры таких вспомогательных функций.

20.2.1. Основные утилиты для работы с запросами GET

В примере 20.2 приводится очень простая функция, обрабатывающая наиболее общий случай использования объекта XMLHttpRequest: просто передайте ей требуемый URL-адрес и функцию, которая примет текст ответа.

Пример 20.2. Вспомогательная функция HTTP.getText( )

```
/**
 * Использует объект XMLHttpRequest для получения содержимого по заданному
 * URL-адресу методом GET. Получив ответ, передает его
 * (в виде простого текста) указанной функции обратного вызова.
 *
 * Эта функция не блокируется и не имеет возвращаемого значения.
 */
HTTP.getText = function(url, callback) {
    var request = HTTP.newRequest();
    request.onreadystatechange = function() {
        if (request.readyState == 4 && request.status == 200)
            callback(request.responseText);
    }
    request.open("GET", url);
    request.send(null);
};
```

В примере 20.3 приводится тривиальный вариант функции, которая принимает XML-документ и передает его функции обратного вызова.

Пример 20.3. Вспомогательная функция HTTP.getXML( )

```
HTTP.getXML = function(url, callback) {
    var request = HTTP.newRequest();
    request.onreadystatechange = function() {
        if (request.readyState == 4 && request.status == 200)
            callback(request.responseXML);
    }
    request.open("GET", url);
    request.send(null);
};
```

20.2.2. Получение только заголовков

Одна из особенностей объекта `XMLHttpRequest` заключается в том, что он позволяет определить используемый HTTP-метод. HTTP-метод HEAD запрашивает у сервера только заголовки для заданного URL-адреса без содержимого, расположенного по этому адресу. Эта возможность может использоваться, например, для проверки даты последнего изменения ресурса, прежде чем загружать его.

В примере 20.4 демонстрируется, как можно выполнить запрос HEAD. Он включает функцию, которая выполняет анализ пар имя–значение в HTTP-заголовке и сохраняет их в виде свойств JavaScript-объекта. Здесь также имеется функция обработки ошибок, которая вызывается в случае получения от сервера кода состояния 404 и других кодов ошибок.

Пример 20.4. Вспомогательная функция HTTP.getHeaders( )

```
/**
 * Использует HTTP-запрос HEAD для получения заголовков с указанного
 * URL-адреса. После получения заголовков анализирует их с помощью функции
 * HTTP.parseHeaders() и передает получившийся объект указанной функции
 * обратного вызова. Если сервер вернет код ошибки, вызывает указанную
 * функцию errorHandler. Если обработчик ошибок не задан, передает значение
 * null функции обратного вызова.
 */
HTTP.getHeaders = function(url, callback, errorHandler) {
    var request = HTTP.newRequest();
    request.onreadystatechange = function() {
        if (request.readyState == 4) {
            if (request.status == 200) {
                callback(HTTP.parseHeaders(request));
            }
            else {
                if (errorHandler) errorHandler(request.status,
                                               request.statusText);
                else callback(null);
            }
        }
    }
    request.open("HEAD", url);
    request.send(null);
};

// Анализирует заголовки ответа, полученные в XMLHttpRequest, и возвращает
// имена и значения в виде свойств нового объекта.
HTTP.parseHeaders = function(request) {
    var headerText = request.getAllResponseHeaders(); // Текст от сервера
    var headers = {}; // Это возвращаемое значение
    var ls = /^\s*/;  // Регулярное выражение, удаляющее начальные пробелы
    var ts = /\s*$/;  // Регулярное выражение, удаляющее конечные пробелы

    // Разбить заголовки на строки
    var lines = headerText.split("\n");
    // Цикл по всем строкам
    for(var i = 0; i < lines.length; i++) {
        var line = lines[i];
```

```
            if (line.length == 0) continue; // Пропустить пустые строки
            // Разбить каждую строку по первому двоеточию и удалить лишние пробелы
            var pos = line.indexOf(':');
            var name = line.substring(0, pos).replace(ls, "").replace(ts, "");
            var value = line.substring(pos+1).replace(ls, "").replace(ts, "");
            // Сохранить пару имя-значение в виде свойства JavaScript-объекта
            headers[name] = value;
        }
        return headers;
    };
```

20.2.3. HTTP-метод POST

HTML-формы по умолчанию отправляются на сервер методом POST. При выполнении запроса POST данные передаются на сервер в теле запроса, а не в строке URL-адреса. Поскольку параметры запроса в методе GET приходится вставлять в URL, метод GET пригоден только для случаев, когда запрос не вызывает побочных эффектов на стороне сервера, т, е. когда повторные запросы GET с тем же самым URL-адресом и с теми же параметрами приводят к получению тех же самых результатов. Если запрос сопровождается побочными эффектами (например, сервер сохраняет некоторые из параметров в базе данных), должен использоваться запрос POST.

Пример 20.5 демонстрирует порядок выполнения запросов POST с помощью объекта `XMLHttpRequest`. Метод `HTTP.post()` вызывает функцию `HTTP.encodeFormData()` для преобразования свойств объекта в строковую форму, которая может использоваться в качестве тела запроса POST. Затем полученная строка передается методу `XMLHttpRequest.send()` и становится телом запроса. (Кроме того, строка, созданная с помощью функции `HTTP.encodeFormData()`, может добавляться в URL-адрес метода GET; достаточно лишь отделить URL-адрес и данные символом вопросительного знака.) Помимо этого в примере 20.5 используется метод `HTTP._getResponse()`. Данный метод анализирует ответ сервера на основе его типа. Реализация этого метода приводится в следующем разделе.

Пример 20.5. Вспомогательная функция HTTP.post( )

```
/**
 * Отправляет HTTP-запрос POST по указанному URL-адресу,
 * используя имена и значения свойств объекта в качестве тела запроса.
 * Анализирует ответ сервера на основе его типа и передает
 * полученное значение функции обратного вызова.
 * В случае появления HTTP-ошибки вызывает заданную
 * функцию errorHandler или передает значение null
 * функции обратного вызова, если обработчик ошибок не определен.
 **/
HTTP.post = function(url, values, callback, errorHandler) {
    var request = HTTP.newRequest();
    request.onreadystatechange = function() {
        if (request.readyState == 4) {
            if (request.status == 200) {
                callback(HTTP._getResponse(request));
            }
            else {
```

```
                if (errorHandler) errorHandler(request.status,
                                               request.statusText);
                else callback(null);
            }
        }
    }
    request.open("POST", url);

    // Этот заголовок сообщает серверу, как интерпретировать тело запроса.
    request.setRequestHeader("Content-Type",
                             "application/x-www-form-urlencoded");
    // Вставить в тело запроса имена и значения свойств объекта
    // и отправить их в теле запроса.
    request.send(HTTP.encodeFormData(values));
};

/**
 * Интерпретирует имена и значения свойств объекта, как если бы они были
 * значениями элементов формы, использует формат application/x-www-form-urlencoded
 */
HTTP.encodeFormData = function(data) {
    var pairs = [];
    var regexp = /%20/g; // Регулярное выражение, соответствующее закодированному пробелу
    for(var name in data) {
        var value = data[name].toString();
        // Создать пару имя/значение, но сначала имя и значение закодировать.
        // Практически все, что нам требуется, выполняет глобальная функция
        // encodeURIComponent, но она превращает пробелы в виде %20 вместо
        // требуемого нам "+". Исправить это можно с помощью String.replace()
        var pair = encodeURIComponent(name).replace(regexp,"+") + '=' +
                   encodeURIComponent(value).replace(regexp,"+");
        pairs.push(pair);
    }

    // Объединить все пары в строку, разделяя их символами &
    return pairs.join('&');
};
```

Еще один вариант выполнения запроса POST с помощью объекта `XMLHttpRequest` приводится в примере 21.14. Код в этом примере вызывает веб-службу, но вместо значений элементов формы в теле запроса передает XML-документ.

20.2.4. Ответы в форматах HTML, XML и JSON

В большинстве примеров, продемонстрированных выше, ответ сервера на HTTP-запрос интерпретировался как простой текст. Это вполне законно, и никто не может заявить, что веб-серверы не могут возвращать документы с содержимым типа «text/plain». Анализировать такой ответ из JavaScript-сценария можно с помощью разнообразных строковых методов и делать с ним все, что потребуется.

Ответ сервера всегда можно интерпретировать как простой текст, даже если его содержимое имеет другой тип. Если сервер, например, возвращает HTML-документ, можно извлечь содержимое этого документа с помощью свойства `responseText` и затем вставить его в какой-либо элемент документа с помощью свойства `innerHTML`.

Однако существуют и другие способы обработки ответа, полученного от сервера. Как уже отмечалось в начале главы, если сервер отправляет ответ с содержимым типа «text/xml», можно получить преобразованное представление XML-документа из свойства `responseXML`. Значением этого свойства является DOM-объект `Document`, поэтому для работы с таким документом допускается использовать DOM-методы.

Однако следует заметить, что использование формата XML для передачи данных может оказаться далеко не лучшим выбором. Если сервер передает данные, которые будут обрабатываться на стороне клиента JavaScript-сценарием, весьма неэффективным будет сначала преобразовать эти данные в формат XML на стороне сервера, затем с помощью объекта `XMLHttpRequest` преобразовать эти данные в DOM-дерево узлов, а потом в сценарии выполнять обход этого дерева для извлечения данных. Более короткий путь заключается в том, чтобы на стороне сервера преобразовать данные в литералы объектов и массивов, а затем передать полученный исходный текст на языке JavaScript веб-броузеру. После этого сценарий сможет «проанализировать» ответ, просто передав его методу `eval()`.

Преобразование данных в форму JavaScript-объектов известно под названием JSON (JavaScript Object Notation – нотация JavaScript-объектов).[1] Вот примеры данных в форматах XML и JSON:

```
<!-- Формат XML -->
<author>
  <name>Wendell Berry</name>
  <books>
    <book>The Unsettling of America</book>
    <book>What are People For?</book>
  </books>
</author>

// Формат JSON
{
    "name": "Wendell Berry",
    "books": [
        "The Unsettling of America",
        "What are People For?"
    ]
}
```

Функция `HTTP.post()`, приводившаяся в примере 20.5, вызывает функцию `HTTP._getResponse()`. Эта функция отыскивает заголовок `Content-Type` и с его помощью определяет форму представления ответа. В примере 20.6 приводится реализация `HTTP_getResponse()`, которая возвращает XML-документ в виде объекта `Document`, интерпретирует с помощью `eval()` содержимое JavaScript- или JSON-документов, а документы любых других типов возвращает в виде обычного текста.

---

[1] Подробнее узнать о JSON можно на сайте *http://json.org*. Идея принадлежит Дугласу Крокфорду (Douglas Crockford); на его веб-сайте можно найти ссылки на утилиты преобразования данных в/из формата JSON, написанные на различных языках программирования. Такой способ кодирования данных может оказаться полезным даже для тех, кто не пользуется JavaScript.

Пример 20.6. HTTP._getResponse( )

```
HTTP._getResponse = function(request) {
    // Проверить тип содержимого, полученного от сервера
    switch(request.getResponseHeader("Content-Type")) {
    case "text/xml":
        // Если это XML-документ, вернуть объект Document.
        return request.responseXML;
    case "text/json":
    case "text/javascript":
    case "application/javascript":
    case "application/x-javascript":
        // Если это JavaScript-код или документ в формате JSON, вызвать eval(),
        // чтобы выполнить преобразование текста в JavaScript-значение.
        // Обратите внимание: делать это следует только в том случае,
        // если добропорядочность сервера не вызывает сомнений!
        return eval(request.responseText);
    default:
        // В противном случае интерпретировать ответ как простой текст
        // и вернуть его как строку.
        return request.responseText;
    }
};
```

Не используйте метод `eval()` для обработки данных в формате JSON, как это делается в примере 20.6, если не уверены в том, что сервер никогда не пришлет злонамеренный программный код вместо данных в формате JSON. Более безопасная альтернатива методу `eval()` – выполнить разбор объектов-литералов JavaScript «вручную», без вызова `eval()`.

20.2.5. Ограничение времени ожидания запроса

Недостатком объекта `XMLHttpRequest` является отсутствие возможности ограничить время ожидания исполнения запроса. Этот недостаток особенно критичен для синхронных запросов. Если связь с сервером пропадет, веб-броузер окажется заблокированным в методе `send()` и не будет реагировать на действия пользователя. В случае асинхронных запросов такого не происходит, поскольку метод `send()` не блокируется, и веб-броузер может продолжать реагировать на действия пользователя. Однако и здесь существует проблема ограничения времени выполнения запроса. Предположим, что приложение с помощью объекта `XMLHttpRequest` запустило HTTP-запрос, когда пользователь щелкнул на кнопке. Чтобы предотвратить возможность отправки нескольких запросов, нелишне сделать кнопку неактивной до того момента, как прибудет ответ сервера. Но что если сервер остановится или произойдет нечто, что помешает получению ответа на запрос? Броузер не будет заблокирован, но возможности приложения из-за неактивной кнопки окажутся ограниченными.

Чтобы избежать проблем подобного рода, было бы удобно иметь возможность устанавливать собственные тайм-ауты с помощью функции `Windows.setTimeout()` при выполнении HTTP-запросов. В обычной ситуации ответ приходит до того, как будет вызван обработчик события таймера, – в этом случае можно просто вызвать функцию `Window.clearTimeout()`, чтобы отменить срабатывание таймера. С другой стороны, если обработчик события от таймера будет вызван раньше,

чем свойство readyState получит значение 4, исполнение запроса можно будет отменить с помощью метода XMLHttpRequest.abort(). После этого обычно следует известить пользователя о том, что попытка выполнения запроса потерпела неудачу (например, методом Window.alert()). Если в этом гипотетическом примере перед запуском запроса кнопка была деактивирована, ее можно будет повторно активировать по истечении предельного времени ожидания.

В примере 20.7 определяется функция HTTP.get(), которая демонстрирует только что описанный прием организации тайм-аута. Она представляет собой усовершенствованную версию функции HTTP.getText() из примера 20.2 и поддерживает многие из возможностей, демонстрировавшихся в предыдущих примерах, включая обработку ошибок, параметры запроса и метод HTTP._getResponse(), описанный выше. Кроме того, она допускает возможность указать необязательную функцию обратного вызова, которая будет вызываться всякий раз, когда произойдет событие onreadystatechange со значением свойства readyState, отличным от 4. В таких броузерах, как Firefox, обработчик этого события может вызываться неоднократно со значением 3, и данная функция обратного вызова позволяет сценарию демонстрировать пользователю индикатор процесса загрузки.

Пример 20.7. Вспомогательная функция HTTP.get( )

```
/**
 * Отправляет HTTP-запрос GET с заданным URL. В случае успешного
 * получения ответа он преобразуется в объект на основе заголовка
 * Content-Type и передается указанной функции обратного вызова.
 * Дополнительные аргументы могут быть переданы в виде свойств объекта options.
 *
 * Если получен ответ с сообщением об ошибке (например, сообщение
 * 404 Not Found), код состояния и сообщение передаются функции
 * options.errorHandler. Если обработчик ошибок не определен, вызывается
 * функция обратного вызова со значением null в аргументе.
 *
 * Если объект options.parameters определен, его свойства интерпретируются
 * как имена и значения параметров запроса. С помощью HTTP.encodeFormData()
 * они преобразуются в строку, которую можно вставить в URL, после чего эта
 * строка добавляется в конец URL вслед за символом '?'.
 *
 * Если определена функция options.progressHandler, она будет вызываться
 * всякий раз, когда свойство readyState обретает новое значение, меньшее 4.
 * Каждый раз этой функции будет передаваться количество вызовов этой функции.
 *
 * Если указано значение options.timeout, работа объекта XMLHttpRequest будет
 * прервана, если запрос не будет исполнен до истечения заданного числа миллисекунд.
 * Если предельное время ожидания истекло и определена функция
 * options.timeoutHandler, она будет вызвана со строкой
 * URL запроса в виде аргумента.
 *
 **/
HTTP.get = function(url, callback, options) {
    var request = HTTP.newRequest();
    var n = 0;
    var timer;
    if (options.timeout)
```

```
            timer = setTimeout(function() {
                                    request.abort();
                                    if (options.timeoutHandler)
                                        options.timeoutHandler(url);
                                },
                                options.timeout);
        request.onreadystatechange = function() {
            if (request.readyState == 4) {
                if (timer) clearTimeout(timer);
                if (request.status == 200) {
                    callback(HTTP._getResponse(request));
                }
                else {
                    if (options.errorHandler)
                        options.errorHandler(request.status,
                                             request.statusText);
                    else callback(null);
                }
            }
            else if (options.progressHandler) {
                options.progressHandler(++n);
            }
        }
        var target = url;
        if (options.parameters)
            target += "?" + HTTP.encodeFormData(options.parameters)
        request.open("GET", target);
        request.send(null);
    };
```

20.3. Ajax и динамические сценарии

Термин *Ajax* обозначает архитектуру веб-приложений, которая основана на взаимодействии с протоколом HTTP и объектом `XMLHttpRequest`. (В действительности для многих объект `XMLHttpRequest` и Ajax являются синонимами.) Ajax – это акроним от «Asynchronous JavaScript and XML» (асинхронный JavaScript и XML).[1] Термин был придуман Джессом Джеймсом Гарреттом (Jesse James Garrett) и впервые появился в феврале 2005 года в его статье «Ajax: A New Approach to Web Applications» (Ajax: новый подход к разработке веб-приложений), которую можно найти по адресу *http://www.adaptivepath.com/publications/essays/archives/000385.php*.

[1] Значимость архитектуры Ajax сложно переоценить, и наличие простого названия лишь послужило катализатором начала революции в разработке веб-приложений. Однако, как оказывается, этот акроним недостаточно полно описывает технологии, используемые Ajax-приложениями. Все клиентские JavaScript-сценарии используют механизм обработки событий и потому являются асинхронными. Кроме того, применение XML в веб-приложениях, разработанных в стиле Ajax, часто бывает удобным, но это совершенно не обязательно. Главная особенность Ajax-приложений – взаимодействие с протоколом HTTP, но это никак не отражено в акрониме.

Объект `XMLHttpRequest`, на котором основана технология Ajax, был доступен в броузерах Microsoft и Netscape/Mozilla еще за четыре года до появления статьи Гарретта, но он никогда до этого момента не привлекал такого внимания.[1] Все изменилось в 2004 году, когда компания Google выпустила новую версию почтового веб-приложения Gmail, использующего объект `XMLHttpRequest`. Сочетание этого высококлассного приложения, выполненного на профессиональном уровне, и статьи Гарретта, вышедшей в начале 2005 года, открыло шлюзы для бурного интереса к Ajax.

Ключевой особенностью любого Ajax-приложения является взаимодействие с веб-сервером по протоколу HTTP без необходимости полной перезагрузки страницы. Поскольку объемы передаваемых данных невелики и броузер не тратит время на анализ и отображение целого документа (а также связанных с ним таблиц стилей и сценариев), время отклика таких приложений оказывается очень небольшим. В результате веб-приложения стали напоминать традиционные настольные приложения.

Необязательной особенностью Ajax-приложений является использование формата XML для представления данных во время обмена между клиентом и сервером. В главе 21 демонстрируется, как можно манипулировать XML-данными из JavaScript-сценариев, включая исполнение XPath-запросов и выполнение XSL-преобразований XML-документов в формат HTML. В некоторых Ajax-приложениях для отделения содержимого (данные в формате XML) от представления (HTML-форматирование, выполненное с помощью таблиц стилей XSL) используется язык XSLT. Такой подход дает дополнительные преимущества, уменьшая объемы данных, передаваемых от сервера клиенту, и перенося выполнение необходимых преобразований на сторону клиента.

Существует возможность формализовать Ajax в терминах RPC-механизма[2]. В такой формулировке веб-разработчики используют низкоуровневые Ajax-библиотеки как на стороне сервера, так и на стороне клиента, чтобы облегчить высокоуровневое взаимодействие между клиентом и сервером. В данной главе не описываются никакие библиотеки, реализующие RPC средствами Ajax, потому что основное внимание здесь уделяется низкоуровневым технологиям, обеспечивающим работу архитектуры Ajax.

Ajax – достаточно молодая прикладная архитектура, и описывающая ее статья Гарретта заканчивается призывом к действию, который стоит того, чтобы привести его здесь:

> Самые большие сложности в разработке Ajax-приложений лежат вовсе не в технической плоскости. Технологии, составляющие основу Ajax, достаточно

---

[1] Я сожалею, что не взялся за описание объекта `XMLHttpRequest` в четвертом издании этой книги. То издание было во многом основано на стандартах, и объект `XMLHttpRequest` не был включен в него просто потому, что он никогда не был стандартизован. Если бы в то время я осознавал мощные возможности, которые предоставляет работа с протоколом HTTP, я нарушил бы свои правила и включил бы описание объекта в книгу.

[2] Аббревиатура RPC происходит от Remote Procedure Call (вызов удаленных процедур) и описывает стратегию, используемую в распределенных вычислениях для упрощения взаимодействий между клиентом и сервером.

зрелые, устоявшиеся и понятные. Главные проблемы заключаются в том, что разработчики таких приложений забывают думать о существующих ограничениях Всемирной паутины и начинают воображать себе более широкий, более богатый диапазон возможностей.

Это будет забавно.

20.3.1. Пример применения Ajax

Примеры, приводившиеся до сих пор в этой главе, представляли собой вспомогательные функции, демонстрирующие порядок использования объекта `XMLHttpRequest`. Они не показывали, *зачем* может потребоваться этот объект или *какие* выгоды он дает. Как отмечалось в цитате из статьи Гарретта, архитектура Ajax открывает массу новых возможностей, которые только начали исследоваться. Следующий пример достаточно прост, но в нем демонстрируются некоторые вспомогательные функции и ряд возможностей, предоставляемых архитектурой Ajax.

Пример 20.8 представляет собой ненавязчивый сценарий, регистрирующий обработчики событий в ссылках документа, чтобы с их помощью отображать всплывающие подсказки при наведении на них указателя мыши. Для ссылок, указывающих на тот же сервер, откуда был загружен сам документ, сценарий выполняет HTTP-запрос HEAD с помощью объекта `XMLHttpRequest`. Из возвращаемых сервером заголовков извлекаются тип содержимого, размер и дата последнего изменения документа, на который указывает ссылка, и эта информация отображается в виде всплывающей подсказки (рис. 20.1). Тем самым всплывающие подсказки предоставляют своего рода средство предварительной оценки целевого документа, что может помочь пользователям в принятии решения о том, стоит ли щелкать на этой ссылке или нет.

В основе реализации лежит класс `Tooltip`, разработанный в примере 16.4 (здесь не требуется расширенная версия класса, которая приводилась в примере 17.3). Кроме того, здесь используется модуль `Geometry` из примера 14.2 и вспомогательная функция `HTTP.getHeaders()` из примера 20.4. В программный код заложено несколько уровней асинхронности: в форме это обработчик события `onload`, обработчик события `onmouseover`, таймер и функция обратного вызова для объекта `XMLHttpRequest`. Все это приводит к созданию глубоко вложенных функций.

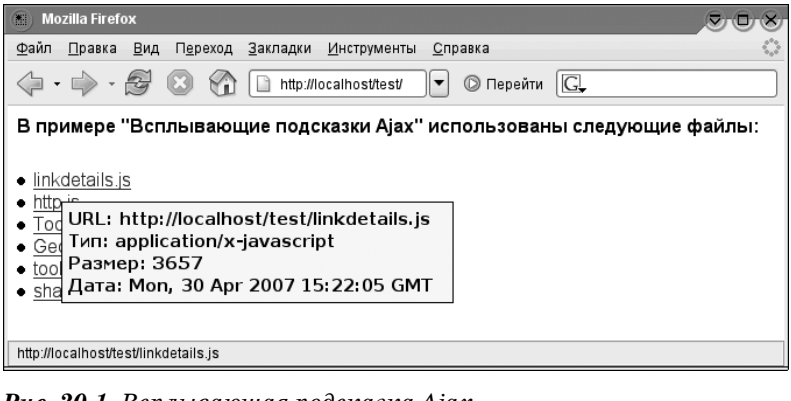

Рис. 20.1. *Всплывающая подсказка Ajax*

Пример 20.8. Всплывающие подсказки Ajax

```
/**
 * linkdetails.js
 *
 * Данный модуль добавляет обработчики событий к ссылкам в документе,
 * с помощью которых отображаются всплывающие подсказки при задержке
 * указателя мыши над этими ссылками в течение полусекунды. Если ссылка
 * указывает на документ на том же сервере, что и исходный документ, всплывающая
 * подсказка будет включать в себя информацию о типе, размере и дате, которая
 * извлекается с помощью HTTP-запроса HEAD, выполняемого объектом XMLHttpRequest.
 *
 * Данный модуль требует наличия модулей Tooltip.js, HTTP.js и Geometry.js
 */
(function() { // Анонимная функция, которая содержит все необходимые имена
    // Создает объект подсказки
    var tooltip = new Tooltip();

    // Настроить вызов функции init() после загрузки документа
    if (window.addEventListener) window.addEventListener("load", init, false);
    else if (window.attachEvent) window.attachEvent("onload", init);

    // Вызывается после загрузки документа
    function init() {
        var links = document.getElementsByTagName('a');

    // Цикл по всем ссылкам и добавление обработчиков событий
    for(var i = 0; i < links.length; i++)
        if (links[i].href) addTooltipToLink(links[i]);
    }

    // Эта функция добавляет обработчики событий
    function addTooltipToLink(link) {
        // Добавить обработчики событий
        if (link.addEventListener) { // Стандартный прием
            link.addEventListener("mouseover", mouseover, false);
            link.addEventListener("mouseout", mouseout, false);
        }
        else if (link.attachEvent) { // Для IE
            link.attachEvent("onmouseover", mouseover);
            link.attachEvent("onmouseout", mouseout);
        }

        var timer; // Используется в вызовах функций setTimeout/clearTimeout

        function mouseover(event) {
            var e = event || window.event;
            // Получить положение указателя мыши, преобразовать
            // в координаты документа и добавить смещение
            var x = e.clientX + Geometry.getHorizontalScroll() + 25;
            var y = e.clientY + Geometry.getVerticalScroll() + 15;

            // Если запланирован вывод подсказки, отменить его
            if (timer) window.clearTimeout(timer);

            // Запланировать вывод подсказки через полсекунды
            timer = window.setTimeout(showTooltip, 500);
```

```
            function showTooltip() {
                // Если HTTP-ссылка указывает на тот же хост, откуда был
                // загружен этот сценарий, использовать объект XMLHttpRequest
                // для получения дополнительной информации.
                if (link.protocol == "http:" && link.host == location.host) {
                    // Выполнить запрос заголовков по ссылке
                    HTTP.getHeaders(link.href, function(headers) {
                        // Собрать строку из заголовков
                        var tip = "URL: " + link.href + "<br>" +
                            "Тип: " + headers["Content-Type"] + "<br>" +
                            "Размер: " + headers["Content-Length"] + "<br>" +
                            "Дата: " + headers["Last-Modified"];
                        // И отобразить ее в виде всплывающей подсказки
                        tooltip.show(tip, x, y);
                    });

                }
                else {
                    // Иначе, если это ссылка на другой сайт,
                    // отобразить в подсказке лишь URL-адрес ссылки
                    tooltip.show("URL: " + link.href, x, y);
                }
            }
        }

        function mouseout(e) {
            // Когда указатель мыши смещается со ссылки, отменить отображение
            // запланированной подсказки или скрыть ее, если она уже отображается.
            if (timer) window.clearTimeout(timer);
            timer = null;
            tooltip.hide();
        }
    }
})();
```

20.3.2. Одностраничные приложения

Под термином *одностраничное приложение* (*single-page application*) понимается именно то, что он обозначает: управляемое JavaScript-сценарием веб-приложение, которое требует загрузки единственной страницы. Некоторые одностраничные приложения после загрузки вообще не взаимодействуют с сервером. Примером таких приложений могут служить DHTML-игры, где взаимодействие с пользователем приводит лишь к модификации загруженного документа.

Объект `XMLHttpRequest` и архитектура Ajax открывают массу дополнительных возможностей. Веб-приложения могут использовать эти технологии для обмена данными с сервером и оставаться одностраничными приложениями. Веб-приложение, разработанное в соответствии с этими положениями, может содержать небольшой объем JavaScript-кода, выполняющего начальную загрузку, и «экранную заставку» в формате HTML, которая отображается в процессе инициализации приложения. После вывода экранной заставки запускающий JavaScript-код мог бы с помощью объекта `XMLHttpRequest` загрузить фактический JavaScript-код приложения, который можно было бы запустить методом `eval()`. Этот Java-

Script-код мог бы взять на себя обязанности по загрузке требуемых данных с помощью `XMLHttpRequest` и с использованием DHTML отобразить эти данные перед пользователем.

20.3.3. Удаленное взаимодействие

Термин *удаленное взаимодействие* (*remote scripting*) появился более чем за четыре года до термина *Ajax* и представляет собой всего лишь менее броское название одной и той же идеи: использование протокола HTTP для обеспечения тесной интеграции (и уменьшения времени отклика) клиента и сервера. В статье компании Apple, вышедшей в 2002 году и получившей широкую известность, описывается, как с помощью тега `<iframe>` отправлять веб-серверу HTTP-запросы (*http://developer.apple.com/internet/webcontent/iframe.html*). В данной статье отмечается, что если веб-сервер отправляет обратно HTML-файл, содержащий тег `<script>`, то JavaScript-код из этого тега будет исполнен броузером и сможет вызывать методы, определенные в окне, содержащем этот тег `<iframe>`. Таким способом сервер может посылать клиенту прямые команды в форме JavaScript-инструкций.

20.3.4. Предостережения по использованию архитектуры Ajax

Подобно любой другой архитектуре, Ajax имеет свои ловушки. В этом разделе описываются три основные проблемы, о которых следует знать при разработке Ajax-приложений.

Первая проблема – визуальная обратная связь. Когда пользователь щелкает на традиционной гиперссылке, веб-броузер обеспечивает индикацию процесса загрузки содержимого ссылки. Такая обратная связь предоставляется еще до того, как содержимое будет готово к отображению, поэтому пользователь явно видит, что броузер работает над выполнением его запроса. Однако, когда HTTP-запрос запускается из объекта `XMLHttpRequest`, броузер не предоставляет никакой обратной связи. Даже при подключении к магистральным линиям инерционность сети часто вызывает заметные задержки между посылкой HTTP-запроса и получением ответа. Поэтому для Ajax-приложений особенно важно обеспечивать визуальную обратную связь (например, в виде простой DHTML-анимации; подробнее об этом см. в главе 16), пока приложение ожидает получение ответа от `XMLHttpRequest`.

Обратите внимание: в примере 20.8 не учитывается совет по обеспечению визуальной обратной связи просто потому, что в данном примере для запуска HTTP-запроса пользователь не предпринимает никаких активных действий. Запрос выполняется, когда пользователь (пассивно) наводит указатель мыши на ссылку. Пользователь явно не требует от приложения выполнить какое-либо действие и потому не нуждается в обратной связи.

Вторая проблема связана с URL. В традиционных веб-приложениях переход из одного состояния в другое сопровождается загрузкой новой страницы, причем каждая страница имеет свой уникальный URL-адрес. Это не относится к Ajax-приложениям: когда Ajax-приложения используют протокол HTTP для загрузки и отображения нового содержимого, URL в адресной строке не изменяется. У пользователя может появиться желание сделать закладку на приложение

в конкретном состоянии, но с помощью механизма закладок броузера сделать это будет невозможно. Более того, пользователю не поможет даже копирование URL из адресной строки броузера.

Эта проблема и ее решение прекрасно иллюстрирует приложение Google Maps (*http://local.google.com*). При изменении масштаба карты или ее прокрутке между клиентом и сервером передаются огромные объемы информации, но URL-адрес, отображаемый в адресной строке броузера, не изменяется. Компания Google решила проблему установки закладок, добавив в каждую страницу ссылку «link to this page» (ссылка на эту страницу). Щелчок на этой ссылке генерирует URL-адрес на отображаемую в данный момент карту и вызывает перезагрузку страницы с этим URL-адресом. После того как загрузка завершится, ссылка на карту в текущем состоянии может быть помещена в закладки, отправлена по электронной почте и тому подобное. Главное, что должны извлечь разработчики из этого урока: все, что имеет существенное значение для описания состояния веб-приложения, должно инкапсулироваться в URL-адресе и этот URL-адрес должен быть доступен пользователю в случае необходимости.

Третья проблема, которая часто упоминается при обсуждении Ajax, связана с кнопкой броузера Назад. Отобрав у броузера контроль над протоколом HTTP, сценарии, использующие объект `XMLHttpRequest`, обходят механизм хранения истории броузера. Пользователи привыкли применять кнопки Назад и Вперед для навигации по Всемирной паутине. Если Ajax-приложение средствами HTTP получает и отображает существенные объемы содержимого документа, пользователи могут попробовать с помощью этих кнопок перемещаться между различными состояниями приложения. Но когда они попробуют щелкнуть на кнопке Назад, то с удивлением обнаружат, что эта кнопка отправляет их за пределы приложения, вместо того чтобы вернуть к его предыдущему состоянию.

В свое время неоднократно предпринимались попытки решить проблему кнопки Назад за счет добавления URL-адресов в историю броузера. Однако, как правило, эти попытки увязали в трясине программного кода, учитывающего специфические особенности каждого броузера, и не давали достаточно удовлетворительных результатов. И даже когда удавалось добиться положительных результатов, найденные решения противоречили основной парадигме Ajax и вынуждали пользователя полностью перезагружать страницы, вместо того чтобы обеспечить прозрачное взаимодействие с сервером по протоколу HTTP.

На мой взгляд, проблема кнопки Назад не настолько серьезна, как ее пытаются представить, и ее отрицательное влияние может быть минимизировано за счет тщательного обдумывания дизайна веб-приложения. Элементы приложения, похожие на гиперссылки, должны вести себя как гиперссылки и действительно должны вызывать перезагрузку страницы. Это сделает их субъектами механизма истории броузера, как того и ожидает пользователь. Те же элементы приложения, которые инициируют взаимодействия с протоколом HTTP в обход механизма истории броузера, наоборот, не должны напоминать гиперссылки. Вернемся к приложению Google Maps еще раз. Когда пользователь прокручивает карту в окне броузера, он не ожидает, что кнопка Назад сможет отменить операцию прокрутки, точно так же, как он не ожидает, что кнопка Назад отменит операцию прокрутки обычной веб-страницы в окне броузера.

Следует с особой осторожностью подходить к использованию слов «вперед» и «назад» в Ajax-приложениях для обозначения внутренних элементов управления навигацией. Например, если интерфейс приложения реализован в стиле многостраничного мастера с кнопками Вперед и Назад для отображения следующего или предыдущего экрана, оно должно поддерживать традиционный способ загрузки страниц (вместо `XMLHttpRequest`), потому что в такой ситуации пользователь вполне оправданно ожидает, что кнопка броузера Назад будет вызывать точно такой же эффект, что и кнопка Назад в приложении.

В более широком смысле кнопка броузера Назад не должна восприниматься как кнопка Отмена в приложении. Ajax-приложения могут предусматривать собственную реализацию операций повторить/отменить, если они будут востребованы пользователем, но они должны совершенно четко отличаться от функций, выполняемых кнопками Назад и Вперед броузера.

20.4. Взаимодействие с протоколом HTTP с помощью тега <script>

В броузерах Internet Explorer версий 5 и 6 объект `XMLHttpRequest` представляет собой ActiveX-объект. Иногда из соображений безопасности пользователи запрещают применение ActiveX-объектов в Internet Explorer, и в такой ситуации сценарии лишены возможности создавать объекты `XMLHttpRequest`. В случае необходимости можно будет выполнять простые HTTP-запросы GET с помощью тегов `<iframe>` и `<script>`. Несмотря на то, что таким способом невозможно реализовать все функциональные возможности объекта `XMLHttpRequest`,[1] тем не менее удастся реализовать, по меньшей мере, вспомогательную функцию `HTTP.getText()`, которая работает без привлечения ActiveX.

Сгенерировать HTTP-запрос достаточно просто с помощью свойства `src` тегов `<script>` и `<iframe>`. Но гораздо сложнее извлечь данные из этих элементов без изменения этих данных броузером. Тег `<iframe>` ожидает, что в него будет загружен HTML-документ. Если попытаться загрузить в плавающий фрейм простой текст, вы обнаружите, что текст будет преобразован в формат HTML. Кроме того, некоторые версии Internet Explorer некорректно реализуют обработку событий `onload` и `onreadystatechange` в теге `<iframe>`, что еще больше осложняет ситуацию.

Подход, который здесь рассматривается, основан на использовании тега `<script>` и сценария на стороне сервера. В этом случае серверному сценарию сообщается URL-адрес, содержимое которого требуется получить, и имя функции на стороне клиента, которой это содержимое должно быть передано. Серверный сценарий берет содержимое с требуемого URL-адреса, преобразует его в JavaScript-строку (возможно, достаточно длинную) и возвращает клиентский сценарий, который передает эту строку указанной функции. Поскольку данный клиентский сценарий загружается в тег `<script>`, по окончании загрузки указанная функция вызывается автоматически.

В примере 20.9 приводится реализация серверного сценария на языке PHP.

---

[1] Полная замена объекта `XMLHttpRequest`, вероятно, потребует применения Java-апплета.

Пример 20.9. jsquoter.php

```
<?php
// Указать броузеру, что выполняется передача сценария
header("Content-Type: text/javascript");
// Извлечь аргументы из URL
$func = $_GET["func"];       // Функция вызова нашего JavaSript-кода
$filename = $_GET["url"];    // Файл или URL для передачи функции func
$lines = file($filename);    // Получить строки содержимого файла
$text = implode("", $lines); // Объединить их в одну строку
// Экранировать кавычки и символы перевода строки
$escaped = str_replace(array("'", "\"", "\n", "\r"),
                       array("\\'", "\\\"", "\\n", "\\r"),
                       $text);
// Отправить все это в виде одиночного вызова JavaScript-функции
echo "$func('$escaped');"
?>
```

Клиентская функция в примере 20.10 использует серверный сценарий *jsquoter.php* из примера 20.9 и работает на манер функции `HTTP.getText()` из примера 20.2.

Пример 20.10. Вспомогательная функция HTTP.getTextWithScript( )

```
HTTP.getTextWithScript = function(url, callback) {
    // Создать новый элемент-сценарий и добавить его в документ.
    var script = document.createElement("script");
    document.body.appendChild(script);

    // Получить уникальное имя для функции.
    var funcname = "func" + HTTP.getTextWithScript.counter++;

    // Определить функцию с этим именем, используя данную функцию как удобное
    // пространство имен. Сценарий, созданный на стороне сервера,
    // будет вызывать эту функцию.
    HTTP.getTextWithScript[funcname] = function(text) {
        // Передать текст функции обратного вызова
        callback(text);

        // Удалить тег script и созданную функцию.
        document.body.removeChild(script);
        delete HTTP.getTextWithScript[funcname];
    }

    // Создать URL-адрес, содержимое которого требуется получить, и имя функции
    // в качестве аргументов серверного сценария jsquoter.php. Установить
    // значение свойства src тега <script>, чтобы получить требуемый URL-адрес.
    script.src = "jsquoter.php" +
                 "?url=" + encodeURIComponent(url) + "&func=" +
                 encodeURIComponent("HTTP.getTextWithScript." + funcname);
}

// Этот счетчик используется для генерации уникальных имен функций обратного
// вызова на случай, если потребуется запланировать выполнение нескольких
// запросов одновременно.
HTTP.getTextWithScript.counter = 0;
```

21

JavaScript и XML

Наиболее важная особенность веб-приложений, созданных на основе архитектуры Ajax, состоит в их способности взаимодействовать по протоколу HTTP с использованием объекта `XMLHttpRequest`, о чем рассказывалось в главе 20. Символ X в акрониме «Ajax» означает XML, и для многих веб-приложений способность работать с данными в формате XML – это их вторая самая важная особенность.

В этой главе рассказывается о том, как работать с данными в формате XML из JavaScript-сценариев. Начинается она с демонстрации способов получения данных в формате XML: загрузка из сети, преобразование из строкового представления и получение их из *островков XML-данных* внутри HTML-документа. После обсуждения приемов получения XML-данных будут описаны базовые приемы работы с этими данными. Здесь рассматриваются вопросы использования прикладного интерфейса (API) модели W3C DOM, преобразования XML-данных с помощью таблиц XSL-стилей, выполнения запросов XML-данных с использованием выражений языка XPath и обратное преобразование XML-данных в строковую форму (сериализация).

После описания этих базовых приемов следуют два раздела, в которых демонстрируются приложения, использующие эти приемы. Сначала вы узнаете, как определять HTML-шаблоны и автоматически разворачивать их с данными из XML-документа средствами DOM и XPath. Затем вы узнаете, как на языке JavaScript разработать клиента веб-служб, опираясь на представленные в этой главе приемы применения XML.

И наконец, в заключении главы дается краткое введение в E4X – мощное расширение ядра языка JavaScript, предназначенное для работы с XML.

21.1. Получение XML-документов

В главе 20 было показано, как с помощью объекта `XMLHttpRequest` загрузить с веб-сервера XML-документ. После выполнения запроса свойство `responseXML` объекта `XMLHttpRequest` будет ссылаться на объект `Document`, являющийся представлением

XML-документа. Но это не единственный способ получения объекта Document с XML-документом. В следующих подразделах показано, как создать пустой XML-документ, как загрузить XML-документ без использования объекта XMLHttpRequest, как преобразовать XML-документ из строки и как получить XML-документ из островка XML-данных.

Как и многие другие расширенные возможности JavaScript-кода, приемы получения XML-данных во многом зависят от типа броузера. В следующих подразделах определяются вспомогательные функции, работающие как в Internet Explorer, так и в Firefox.

Эти вспомогательные функции представляют собой часть одного большого модуля и находятся в пространстве имен XML (см. главу 10). Однако в приведенных здесь примерах вы не найдете программный код, фактически создающий пространство имен. В пакет с примерами, который можно загрузить с сайта издательства, входит файл с именем xml.js, который включает в себя программный код создания пространства имен, но вы можете в рассматриваемые здесь примеры просто добавить одну строку:

```
var XML = {};
```

21.1.1. Создание нового документа

Создать пустой XML-документ (за исключением необязательного корневого элемента) в Firefox и родственных ему броузерах можно с помощью метода document.implementation.createDocument() модели DOM Level 2. То же самое в Internet Explorer можно сделать с помощью ActiveX-объекта MSXML2.DOMDocument. В примере 21.1 приводится определение вспомогательной функции XML.newDocument(), которая скрывает внутри себя различия между этими двумя подходами. От пустого XML-документа мало проку, но его создание – это лишь первый шаг в подготовке к загрузке документа и его преобразованию, что демонстрируется в следующих примерах.

Пример 21.1. Создание пустого XML-документа

```
/**
 * Создает новый объект Document. При отсутствии аргументов создает пустой
 * документ. Если указан корневой тег, документ будет содержать единственный
 * корневой тег. Если корневой тег имеет префикс пространства имен, второй аргумент
 * должен содержать URL-адрес, идентифицирующий это пространство имен.
 */
XML.newDocument = function(rootTagName, namespaceURL) {
    if (!rootTagName) rootTagName = "";
    if (!namespaceURL) namespaceURL = "";

    if (document.implementation && document.implementation.createDocument) {
        // Способ создания в соответствии со стандартом W3C
        return document.implementation.createDocument(namespaceURL,
                                                       rootTagName, null);
    }
    else { // Способ, специфичный для IE
        // Создать пустой документ как ActiveX-объект.
        // Если корневой элемент не определен, на этом создание
        // документа можно считать законченным
```

```
        var doc = new ActiveXObject("MSXML2.DOMDocument");

        // Если корневой элемент определен, инициализировать документ
        if (rootTagName) {
            // Проверить наличие префикса пространства имен
            var prefix = "";
            var tagname = rootTagName;
            var p = rootTagName.indexOf(':');
            if (p != -1) {
                prefix = rootTagName.substring(0, p);
                tagname = rootTagName.substring(p+1);
            }

            // Если пространство имен определено, должен быть префикс пространства имен.
            // Если пространство имен не определено, необходимо удалить
            // существующий префикс
            if (namespaceURL) {
                if (!prefix) prefix = "a0"; // Используется в Firefox
            }
            else prefix = "";
            // Создать корневой элемент (с необязательным пространством
            // имен) в виде текстовой строки
            var text = "<" + (prefix?(prefix+":"):"") + tagname +
                (namespaceURL
                 ?(" xmlns:" + prefix + '="' + namespaceURL +'"')
                 :"") +
                "/>";
            // И преобразовать текст в пустой документ
            doc.loadXML(text);
        }
        return doc;
    }
};
```

21.1.2. Загрузка документа из сети

В главе 20 было показано, как с помощью объекта `XMLHttpRequest` выполнить HTTP-запрос с целью загрузки текстового документа. В случае XML-документов свойство `responseXML` будет ссылаться на преобразованное представление документа в виде DOM-объекта `Document`. Несмотря на то, что объект `XMLHttpRequest` не стандартизован, он получил широкое распространение и, как правило, представляет собой наилучшее средство загрузки XML-документов.

Однако *существует* и другой способ. XML-объект `Document`, созданный способом, описанным в примере 21.1, может загружать и анализировать XML-документы, используя для этого менее известный прием. Пример 21.2 демонстрирует, как это делается. Что самое удивительное, в IE и броузерах, основанных на Mozilla, применяется один и тот же программный код.

Пример 21.2. Синхронная загрузка XML-документа

```
/**
 * Синхронно загружает XML-документ с заданного URL-адреса
 * и возвращает его в виде объекта Document
 */
```

```
XML.load = function(url) {
    // Создать пустой документ с помощью функции, определенной ранее
    var xmldoc = XML.newDocument();
    xmldoc.async = false; // Загрузка выполняется синхронно
    xmldoc.load(url);     // Загрузить и проанализировать
    return xmldoc;        // Вернуть документ
};
```

Подобно объекту `XMLHttpRequest`, представленный здесь метод `load()` не является стандартным. Он имеет несколько существенных отличий от `XMLHttpRequest`. Во-первых, он работает только с XML-документами, тогда как `XMLHttpRequest` может применяться для загрузки текстовых документов любого типа. Во-вторых, он не ограничен протоколом HTTP. В частности, он способен читать файлы из локальной файловой системы, что бывает удобно в процессе разработки и отладки веб-приложения. В-третьих, когда задействован протокол HTTP, метод `load()` генерирует запросы GET и не может использоваться для передачи данных веб-серверу методом POST.

Подобно `XMLHttpRequest`, метод `load()` может работать в асинхронном режиме. Фактически этот режим используется по умолчанию, если в свойство `async` явно не будет записано значение `false`. В примере 21.3 приводится асинхронная версия метода `XML.load()`.

Пример 21.3. Асинхронная загрузка XML-документа

```
/**
 * Асинхронно загружает и анализирует XML-документ с заданного URL-адреса.
 * Как только документ будет готов, он передается указанной функции обратного вызова.
 * Данная функция сразу же возвращает управление и не имеет возвращаемого значения.
 */
XML.loadAsync = function(url, callback) {
    var xmldoc = XML.newDocument();

    // Если XML-документ создан методом createDocument, использовать
    // onload для определения момента, когда он будет загружен
    if (document.implementation && document.implementation.createDocument){
        xmldoc.onload = function() { callback(xmldoc); };
    }
    // В противном случае использовать onreadystatechange, как
    // и в случае с объектом XMLHttpRequest
    else {
        xmldoc.onreadystatechange = function() {
            if (xmldoc.readyState == 4) callback(xmldoc);
        };
    }

    // Начать загрузку и анализ документа
    xmldoc.load(url);
};
```

21.1.3. Синтаксический анализ текста XML-документа

Иногда возникает необходимость просто проанализировать XML-документ, имеющий вид JavaScript-строки, а не загружать его из сети. В броузерах, реали-

зованных на базе Mozilla, для этих целей используется объект DOMParser, в IE – метод loadXML() объекта Document. (Если вы внимательно изучили программный код метода XML.newDocument() в примере 21.1, то могли заметить вызов этого метода.)

В примере 21.4 демонстрируется не зависящая от платформы функция, которая выполняет синтаксический разбор XML-документа и работает как в Mozilla, так и в IE. Для платформ, отличающихся от этих двух, она пытается выполнить синтаксический разбор текста, загрузив его с помощью объекта XMLHttpRequest с URL-адреса со спецификатором data:.

Пример 21.4. Синтаксический разбор XML-документа

```
/**
 * Выполняет синтаксический разбор XML-документа, содержащегося в строковом
 * аргументе, и возвращает представляющий его объект Document.
 */
XML.parse = function(text) {
    if (typeof DOMParser != "undefined") {
        // Mozilla, Firefox и родственные броузеры
        return (new DOMParser()).parseFromString(text, "application/xml");
    }
    else if (typeof ActiveXObject != "undefined") {
        // Internet Explorer.
        var doc = XML.newDocument();  // Создать пустой документ
        doc.loadXML(text);            // Выполнить синтаксический
                                      // разбор текста в документе
        return doc;                   // Вернуть документ
    }
    else {
        // Как последняя возможность - попытаться загрузить документ
        // с URL-адреса со спецификатором data:
        // Этот прием работает в Safari. Спасибо Маносу Батсису (Manos Batsis)
        // с его библиотекой Sarissa (sarissa.sourceforge.net).
        var url = "data:text/xml;charset=utf-8," + encodeURIComponent(text);
        var request = new XMLHttpRequest();
        request.open("GET", url, false);
        request.send(null);
        return request.responseXML;
    }
};
```

21.1.4. XML-документы в островках данных

Компания Microsoft расширила язык разметки HTML новым тегом <xml>, с помощью которого создаются островки данных в формате XML в окружающем их «море» HTML-разметки. Когда IE встречает тег <xml>, он интерпретирует его как отдельный XML-документ, который можно извлечь методом document.getElementById() или другими DOM-методами языка HTML. Если в теге <xml> определен атрибут src, тогда вместо того, чтобы анализировать содержимое тега <xml>, XML-документ загружается с URL-адреса, указанного в этом атрибуте.

Если для работы веб-приложения необходимы XML-данные и эти данные заранее известны, определенно есть смысл включить их прямо в HTML-страницу: данные будут доступны сразу же, и приложению не потребуется устанавливать новое со-

единение, чтобы их загрузить. Островки XML-данных – удобные объекты для этого. Существует возможность эмулировать островки данных в IE и в других броузерах с помощью программного кода, который приводится в примере 21.5.

Пример 21.5. Извлечение XML-документа из островка данных

```
/**
 * Возвращает объект Document, в котором хранится содержимое тега <xml>
 * с заданным идентификатором. Если тег <xml> имеет атрибут src, тогда
 * выполняется загрузка документа с этого URL-адреса.
 *
 * Поскольку островки данных часто используются неоднократно, данная
 * функция кэширует возвращаемые документы.
 */
XML.getDataIsland = function(id) {
    var doc;

    // Сначала проверить кэш
    doc = XML.getDataIsland.cache[id];
    if (doc) return doc;

    // Найти требуемый элемент
    doc = document.getElementById(id);

    // Если определен атрибут "src", загрузить документ с указанного URL-адреса
    var url = doc.getAttribute('src');
    if (url) {
        doc = XML.load(url);
    }
    // В противном случае, если атрибут src отсутствует, документ, который
    // следует вернуть, содержится внутри тега <xml>. В Internet Explorer
    // в переменной doc уже будет ссылка на требуемый объект документа.
    // В других броузерах переменная doc ссылается на HTML-элемент, и нам
    // нужно скопировать содержимое этого элемента в новый объект документа
    else if (!doc.documentElement) {// Если это еще не документ...

        // Прежде всего, нужно найти элемент документа в теге <xml>.
        // Это будет первый дочерний элемент тега <xml>, не являющийся текстом,
        // комментарием или исполняемой инструкцией
        var docelt = doc.firstChild;
        while(docelt != null) {
            if (docelt.nodeType == 1 /*Node.ELEMENT_NODE*/) break;
            docelt = docelt.nextSibling;
        }

        // Создать пустой документ
        doc = XML.newDocument();

        // Если узел <xml> имеет какое-то содержимое, импортировать его в новый документ
        if (docelt) doc.appendChild(doc.importNode(docelt, true));
    }

    // Поместить документ в кэш и вернуть его
    XML.getDataIsland.cache[id] = doc;
    return doc;
};
XML.getDataIsland.cache = {}; // Инициализация кэша
```

Этот программный код не совсем точно моделирует островки XML-данных в броузерах, не относящихся к IE. HTML-стандарты требуют, чтобы броузеры не выполняли синтаксический разбор неизвестных им тегов (а просто их игнорировали). Это означает, что броузеры не уничтожают XML-данные, расположенные в теге <xml>. Но это также означает, что любой текст, содержащийся в островках данных, по умолчанию будет отображаться броузером. Самый простой способ предотвратить это заключается в использовании следующей таблицы CSS-стилей:

```
<style type="text/css">xml { display: none; }</style>
```

Другая несовместимость с не относящимися к IE броузерами связана с тем, что они интерпретируют содержимое островков данных как HTML-текст, а не как XML-текст. Если, к примеру, воспользоваться сценарием из примера 21.5 в броузере Firefox и затем сериализовать получившийся документ (как это сделать, показано в этой главе далее), можно обнаружить, что имена тегов преобразованы в верхний регистр, поскольку Firefox предполагает, что имеет дело с HTML-тегами. В большинстве случаев это не вызывает проблем, но в некоторых случаях может стать источником неприятностей. Наконец, следует заметить, что пространства имен будут разрушены, если броузер интерпретирует XML-теги как HTML-теги. Это означает, что островки XML-данных не подходят для хранения таблиц XSL-стилей (более подробно об XSL будет рассказываться в этой главе далее), поскольку эти таблицы всегда используют пространства имен.

Если вы хотите воспользоваться преимуществами, которые дает включение XML-данных непосредственно в HTML-страницу, но не желаете бороться с несовместимостью броузеров из-за наличия островков XML-данных в тегах <xml>, тогда вам следует рассмотреть возможность включения в страницу XML-документов в виде JavaScript-строк, которые затем можно преобразовать в документы с помощью программного кода, приводившегося в примере 21.4.

21.2. Манипулирование XML–данными средствами DOM API

В предыдущих разделах был показан ряд приемов получения XML-данных в виде объекта Document. Объект Document определяется стандартом W3C DOM API и во многом напоминает объект HTMLDocument, на который ссылается свойство document броузера.

В следующих подразделах описываются некоторые существенные различия между моделями HTML DOM и XML DOM, а затем демонстрируется порядок использования прикладного интерфейса (API) модели DOM для извлечения данных из XML-документа и отображения этих данных в динамически создаваемых узлах HTML-документа.

21.2.1. Модели XML DOM и HTML DOM

Модель W3C DOM уже описывалась в главе 15, но там основное внимание уделялось ее использованию в JavaScript-сценариях для работы с HTML-документами на стороне клиента. Фактически консорциум W3C проектировал DOM API как не зависящий от языка прикладной интерфейс, предназначенный в первую очередь для работы с XML-документами, а работа с HTML-документами реализова-

на в нем через необязательный модуль расширения. Обратите внимание: в четвертой части книги есть отдельные разделы, посвященные интерфейсам Document и HTMLDocument, а также объектам Element и HTMLElement. Интерфейсы HTMLDocument и HTMLElement являются расширениями базовых XML-интерфейсов Document и Element. Если вы привыкли применять DOM-интерфейс для манипулирования HTML-документами, то при работе с XML-документами следует избегать использования прикладного интерфейса, специфичного для HTML.

Вероятно, самое существенное отличие между HTML и XML в модели DOM состоит в использовании метода getElementById(), который обычно бесполезен для XML-документов. В DOM уровня 1 этот метод фактически относится только к HTML и определен исключительно в интерфейсе HTMLDocument. В DOM Level 2 этот метод повышен до интерфейса Document, но здесь есть одно препятствие. В XML-документах метод getElementById() ищет элемент с заданным значением атрибута, *тип* которого – «id». Здесь недостаточно определить в элементе атрибут с *именем* «id», поскольку имя атрибута не имеет никакого значения – важен только тип атрибута. Типы атрибутов объявляются в определении типа документа (Document Type Definition, DTD), а DTD документа указывается в объявлении DOCTYPE. XML-документы, используемые веб-приложениями, часто не имеют этого объявления, потому метод getElementById() для таких документов всегда возвращает значение null. Обратите внимание: метод getElementsByTagName() интерфейсов Document и Element прекрасно работает с XML-документами. (Далее в этой главе я покажу, как делать запросы к XML-документу с помощью XPath-выражений; язык запросов XPath может использоваться для извлечения элементов на основе значения любого атрибута.)

Еще одно отличие между HTML- и XML-объектами Document заключается в наличии свойства body, которое ссылается на тег <body> в документе. В случае с XML-документами только свойство documentElement ссылается на элемент верхнего уровня в документе. Обратите внимание: этот элемент верхнего уровня доступен также через свойство childNodes[] документа, но указанный элемент может оказаться не первым и не единственным в этом массиве, поскольку XML-документ может также содержать объявление DOCTYPE, комментарии и исполняемые инструкции верхнего уровня.

Существует еще одно очень важное отличие между XML-интерфейсом Element и интерфейсом HTMLElement, который является его расширением. В модели HTML DOM стандартные HTML-атрибуты доступны в виде свойств интерфейса HTMLElement. Например, атрибут src тега <img> доступен в виде свойства src объекта HTMLImageElement, являющегося представлением тега <img>. Однако в модели XML DOM дело обстоит иначе: интерфейс Element имеет единственное свойство tagName. Получение и изменение значения атрибута XML-элемента должно выполняться методами getAttribute(), setAttribute() и другими родственными им методами.

В заключение следует отметить, что специальные атрибуты, которые являются значащими в любом HTML-теге, не имеют никакого значения для всех XML-элементов. Вспомните, что установка атрибута с именем «id» в XML-элементе не означает, что этот элемент может быть найден методом getElementById(). Аналогичным образом нельзя определить стиль XML-элемента, установив атрибут style. Точно так же невозможно ассоциировать XML-элемент с CSS-классом, установив атрибут class. Все эти атрибуты являются специфичными для HTML.

21.2.2. Пример: создание HTML-таблицы на основе XML-данных

В примере 21.7 определяется функция с именем makeTable(), которая использует модели XML DOM и HTML DOM для извлечения данных из XML-документа и добавления их в HTML-документ в виде таблицы. Функция ожидает получить в виде аргумента литерал JavaScript-объекта, который указывает, какие элементы XML-документа содержат данные для таблицы и как эти данные должны располагаться в таблице.

Прежде чем перейти к программному коду функции makeTable(), рассмотрим пример ее применения. В примере 21.6 приводится образец XML-документа, который мы используем в этом примере (а также в других примерах этой главы).

Пример 21.6. Файл с данными в формате XML

```
<?xml version="1.0"?>
<contacts>
  <contact name="Able Baker"><email>able@example.com</email></contact>
  <contact name="Careful Dodger"><email>dodger@example.com</email></contact>
  <contact name="Eager Framer" personal="true">
  <email>framer@example.com</email></contact>
</contacts>
```

Следующий фрагмент HTML-документа демонстрирует, как может использоваться функция makeTable() с этими XML-данными. Обратите внимание: объект schema ссылается на имена тегов и атрибутов из файла-образца.

```
<script>
// Эта функция обращается к makeTable()
function displayAddressBook() {
    var schema = {
        rowtag: "contact",
        columns: [
            { tagname: "@name", label: "Name" },
            { tagname: "email", label: "Address" }
        ]
    };

    var xmldoc = XML.load("addresses.xml"); // Прочитать XML-данные
    makeTable(xmldoc, schema, "addresses"); // Преобразовать в HTML-таблицу
}
</script>

<button onclick="displayAddressBook()">Показать адресную книгу</button>
<div id="addresses"><!-- таблица будет вставлена сюда --></div>
```

Реализация функции makeTable() приводится в примере 21.7.

Пример 21.7. Построение HTML-таблицы из данных в формате XML

```
/**
 * Извлекает данные из указанного XML-документа и формирует из них HTML-таблицу.
 * Добавляет таблицу в конец указанного HTML-элемента.
 * (Если аргумент element - строка, он интерпретируется как идентификатор,
 * по которому выполняется поиск требуемого элемента.)
```

```
 *
 * Аргумент schema - это JavaScript-объект, указывающий, какие данные должны
 * извлекаться и как они должны отображаться. Объект schema должен иметь
 * свойство с именем "rowtag", где указывается имя XML-тега, в котором
 * содержатся данные для одной строки таблицы. Кроме того, объект
 * schema должен иметь свойство с именем "columns", которое ссылается на массив.
 * Элементы этого массива определяют порядок следования и содержимое
 * столбцов таблицы. Каждый элемент массива может быть строкой или
 * JavaScript-объектом. Если элемент - строка, она интерпретируется как имя
 * тега XML-элемента, где содержатся данные для столбца таблицы, а также как
 * заголовок этого столбца. Если элемент массива columns[] - объект,
 * он должен иметь свойства с именами "tagname" и "label".
 * Свойство tagname используется для извлечения данных из XML-документа,
 * а свойство label - в качестве текста для заголовка столбца. Если значение
 * свойства tagname начинается с символа @, оно рассматривается как название
 * атрибута элемента строки, а не как дочерний элемент строки.
 */
function makeTable(xmldoc, schema, element) {
    // Создать элемент <table>
    var table = document.createElement("table");

    // Создать строку заголовка из элементов <th> внутри элемента <tr> в элементе <thead>
    var thead = document.createElement("thead");
    var header = document.createElement("tr");
    for(var i = 0; i < schema.columns.length; i++) {
        var c = schema.columns[i];
        var label = (typeof c == "string")?c:c.label;
        var cell = document.createElement("th");
        cell.appendChild(document.createTextNode(label));
        header.appendChild(cell);
    }
    // Вставить заголовок в таблицу
    thead.appendChild(header);
    table.appendChild(thead);

    // Остальные строки таблицы располагаются в теге <tbody>
    var tbody = document.createElement("tbody");
    table.appendChild(tbody);

    // Получить элементы XML-документа, в которых содержатся данные
    var xmlrows = xmldoc.getElementsByTagName(schema.rowtag);

    // Цикл по всем этим элементам. Каждый из них содержит строку таблицы.
    for(var r=0; r < xmlrows.length; r++) {
        // Этот XML-элемент содержит данные для всей строки
        var xmlrow = xmlrows[r];
        // Создать HTML-элемент для отображения данных в строке
        var row = document.createElement("tr");

        // Цикл по всем столбцам, указанным в объекте schema
        for(var c = 0; c < schema.columns.length; c++) {
            var sc = schema.columns[c];
            var tagname = (typeof sc == "string")?sc:sc.tagname;
            var celltext;
            if (tagname.charAt(0) == '@') {
                // Если значение tagname начинается с '@' - это имя атрибута
```

```
                celltext = xmlrow.getAttribute(tagname.substring(1));
            }
            else {
                // Найти XML-элемент, где хранятся данные для этого столбца
                var xmlcell = xmlrow.getElementsByTagName(tagname)[0];
                // Предположить, что элемент имеет текстовый узел, как
                // первый дочерний элемент
                var celltext = xmlcell.firstChild.data;
            }
            // Создать HTML-элемент для этой ячейки
            var cell = document.createElement("td");
            // Вставить текстовые данные в HTML-ячейку
            cell.appendChild(document.createTextNode(celltext));
            // Добавить ячейку в строку
            row.appendChild(cell);
        }

        // Добавить строку в тело таблицы
        tbody.appendChild(row);
    }

    // Установить значение HTML-атрибута для элемента table, записав его в свойство.
    // Обратите внимание: в случае с XML-документом мы должны были
    // бы использовать метод setAttribute().
    table.frame = "border";

    // Создание таблицы закончено, теперь ее нужно добавить к указанному
    // элементу. Если этот элемент - строка, интерпретировать ее как
    // значение атрибута ID элемента.
    if (typeof element == "string") element = document.getElementById(element);
    element.appendChild(table);
}
```

21.3. Преобразование XML-документа с помощью XSLT

После того как вы загрузили, выполнили синтаксический разбор или каким-то другим способом получили объект Document, представляющий XML-документ, одно из самых интереснейших действий, которые можно с ним выполнить, – это преобразовать документ с помощью таблицы XSLT-стилей. Аббревиатура XSLT происходит от XSL Transformations (XSL-преобразования), а XSL – от Extensible Stylesheet Language (расширяемый язык таблиц стилей). Таблицы XSL-стилей – это XML-документы, которые могут быть загружены и разобраны, как и любые другие XML-документы. Изучение XSL далеко выходит за рамки темы этой книги, тем не менее в примере 21.8 демонстрируется таблица стилей, которая может использоваться для преобразования в HTML-таблицу XML-документа, подобного представленному в примере 21.6.

Пример 21.8. Простейшая таблица XSL-стилей

```
<?xml version="1.0"?><!-- это XML-документ -->
<!-- объявить пространство имен xsl, чтобы отличать xsl-теги от html-тегов -->
<xsl:stylesheet version="1.0"
```

```
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="html"/>

  <!-- Когда будет найден корневой элемент, вывести каркас HTML-таблицы -->
  <xsl:template match="/">
    <table>
      <tr><th>Имя</th><th>Электронная почта</th></tr>
      <xsl:apply-templates/> <!-- и рекурсия по другим шаблонам -->
    </table>
  </xsl:template>

  <!-- Когда встретится элемент <contact>... -->
  <xsl:template match="contact">
    <tr> <!-- Начать новую строку таблицы -->
      <!-- Использовать атрибут name тега contact как первый столбец -->
      <td><xsl:value-of select="@name"/></td>
      <xsl:apply-templates/> <!-- и рекурсия по другим шаблонам -->
    </tr>
  </xsl:template>

  <!--
    Когда встретится элемент <email>, вывести его содержимое в другую ячейку
  -->
  <xsl:template match="email">
    <td><xsl:value-of select="."/></td>
  </xsl:template>
</xsl:stylesheet>
```

Правила в таблицах XSL-стилей используются для XSLT-преобразований XML-документов. В контексте клиентского JavaScript-кода это обычно означает преобразование XML-документов в HTML-документы. Многие архитектуры разработки веб-приложений используют XSLT на стороне сервера, но броузеры на базе Mozilla и броузеры линейки IE поддерживают XSLT-преобразования на стороне клиента, что может помочь снизить нагрузку на сервер и объем трафика, передаваемого по сети (потому что формат XML, как правило, более компактный, чем HTML).

Многие современные броузеры позволяют определять XML-стили с помощью таблиц CSS- или XSL-стилей. Если определить таблицу стилей в исполняемой инструкции `xml-stylesheet`, тогда можно загрузить XML-документ непосредственно в броузер, а броузер преобразует и отобразит его. Например, исполняемая инструкция могла бы выглядеть примерно так:

```
<?xml-stylesheet href="dataToTable.xml" type="text/xsl"?>
```

Обратите внимание: броузеры выполняют такого рода XSLT-преобразования автоматически, когда XML-документ, содержащий соответствующую исполняемую инструкцию, загружается в окно броузера. Это очень важно и очень удобно, но не это является темой данного раздела. Дальше я расскажу о том, как с помощью JavaScript выполнить динамическое XSLT-преобразование.

Консорциум W3C не определяет стандартного прикладного интерфейса для XSLT-преобразований DOM-объектов `Document` и `Element`. В броузерах на базе Mozilla прикладной интерфейс для XSLT-преобразований в JavaScript представляет объект `XSLTProcessor`. В IE XML-объекты `Document` и `Element` имеют метод `transform-`

Node(), выполняющий преобразования. В примере 21.9 демонстрируется использование обоих прикладных интерфейсов. В нем определяется класс XML.Transformer, который инкапсулирует таблицу XSL-стилей и позволяет использовать ее для преобразования более одного XML-документа. Метод transform() объекта XML.Transformer с помощью инкапсулированной таблицы стилей выполняет преобразование указанного XML-документа, а затем замещает содержимое заданного DOM-элемента результатом преобразования.

Пример 21.9. XSLT в Mozilla и Internet Explorer

```
/**
 * Этот класс XML.Transformer инкапсулирует таблицу XSL-стилей.
 * Если параметр stylesheet представляет собой URL-адрес, выполняется
 * загрузка таблицы. Иначе предполагается, что это ссылка
 * на соответствующий DOM-объект Document.
 */
XML.Transformer = function(stylesheet) {
    // Загрузить таблицу стилей, если это необходимо.
    if (typeof stylesheet == "string") stylesheet = XML.load(stylesheet);
    this.stylesheet = stylesheet;

    // В броузерах на базе Mozilla создать объект XSLTProcessor
    // и передать ему таблицу стилей.
    if (typeof XSLTProcessor != "undefined") {
        this.processor = new XSLTProcessor();
        this.processor.importStylesheet(this.stylesheet);
    }
};

/**
 * Это метод transform() класса XML.Transformer.
 * Выполняет преобразование указанного xml-узла с использованием
 * инкапсулированной таблицы стилей.
 * Предполагается, что в результате преобразования получается HTML-код,
 * которым следует заменить содержимое указанного элемента.
 */
XML.Transformer.prototype.transform = function(node, element) {
    // Если элемент указан по id, отыскать его.
    if (typeof element == "string") element = document.getElementById(element);

    if (this.processor) {
        // Если был создан объект XSLTProcessor (в броузерах на базе Mozilla),
        // использовать его.
        // Преобразовать узел в DOM-объект DocumentFragment.
        var fragment = this.processor.transformToFragment(node, document);
        // Стереть существующее содержимое элемента.
        element.innerHTML = "";
        // И вставить преобразованные узлы.
        element.appendChild(fragment);
    }
    else if ("transformNode" in node) {
        // Если узел имеет метод transformNode() (в IE), использовать его.
        // Обратите внимание: transformNode() возвращает строку.
        element.innerHTML = node.transformNode(this.stylesheet);
    }
```

```
    else {
        // В противном случае удача отвернулась от нас.
        throw "XSLT не поддерживается в этом броузере";
    }
};

/**
 * Эта вспомогательная функция, выполняющая XSLT-преобразование,
 * может быть удобна, когда таблица стилей должна использоваться всего один раз.
 */
XML.transform = function(xmldoc, stylesheet, element) {
    var transformer = new XML.Transformer(stylesheet);
    transformer.transform(xmldoc, element);
}
```

К моменту написания этих строк IE и броузеры на базе Mozilla были единственными из основных броузеров, предоставляющих API для XSLT-преобразований. Если для вас важно иметь поддержку и в других броузерах, вас наверняка заинтересует проект AJAXSLT – свободно распространяемая JavaScript-реализация XSLT-преобразований. Разработка проекта AJAXSLT была начата компанией Google, ознакомиться с ним можно на сайте проекта по адресу *http://goog-ajaxslt.sourceforge.net*.

21.4. Выполнение запросов к XML-документу с помощью XPath-выражений

XPath – это просто язык, с помощью которого можно обращаться к элементам, атрибутам и тексту внутри XML-документа. XPath-выражение может обратиться к XML-элементу по его положению в иерархии документа или выбрать элемент по значению некоторого атрибута (или просто по его присутствию). Подробное обсуждение языка XPath далеко выходит за рамки темы этой главы, тем не менее подраздел 21.4.1 представляет собой краткое руководство по языку XPath, в котором описываются, например, наиболее общие XPath-выражения.

Консорциумом W3C был выработан предварительный стандарт на API для выборки узлов в DOM-дереве документа с помощью XPath-выражения. Firefox и родственные броузеры реализуют этот прикладной интерфейс в виде метода `evaluate()` объекта `Document` (как для HTML-, так и для XML-документов). Кроме того, броузеры на базе Mozilla реализуют метод `Document.createExpression()`, который компилирует XPath-выражения в промежуточное представление, благодаря чему повышается их эффективность при многократном использовании.

В IE вычисление XPath-выражений выполняется с помощью методов `selectSingleNode()` и `selectNodes()` XML-объектов (но не HTML-объектов) `Document` и `Element`. Далее в этом разделе вы найдете пример, в котором используются оба прикладных интерфейса: и W3C, и IE.

Чтобы обеспечить поддержку XPath-выражений и в других броузерах, обдумайте возможность использования свободно распространяемого проекта AJAXSLT (*http://goog-ajaxslt.sourceforge.net*).

21.4.1. Примеры использования XPath-выражений

Если вы понимаете структуру DOM-дерева документа, то без труда разберетесь с простыми XPath-выражениями на примере. Тем не менее чтобы понимать эти примеры, вы должны знать, что XPath-выражение вычисляется относительно некоторого *контекстного* узла документа. Простейшие XPath-выражения просто ссылаются на узлы-потомки контекстного узла:

```
contact // Набор всех тегов <contact>, вложенных в контекстный узел
contact[1] // Первый тег <contact>, вложенный в контекстный узел
contact[last()] // Последний потомок <contact> контекстного узла
contact[last()-1] // Предпоследний потомок <contact> контекстного узла
```

Обратите внимание: индексация элементов массивов в языке XPath начинается с 1, а не с 0, как в JavaScript-массивах.

Слово «path» (путь) в названии «XPath» отражает тот факт, что этот язык интерпретирует уровни в иерархии XML-элементов как каталоги в файловой системе и для отделения уровней иерархии использует символ «/». Например:

```
contact/email // Все потомки <email> потомка <contact> контекстного узла
/contacts // Потомок <contacts> корневого (ведущий слэш) элемента документа
contact[1]/email // Потомок <email> первого потомка <contact>
contact/email[2] // Второй потомок <email> любого потомка <contact> контекста
```

Обратите внимание: выражение `contact/email[2]` возвращает множество элементов `<email>`, которые являются вторыми узлами `<email>` любого узла `<contact>`, вложенного в контекстный узел. Это не то же самое, что `contact[2]/email` или `(contact/email)[2]`.

Точка (`.`) в XPath-выражениях ссылается на контекстный элемент, а двойной слэш (`//`) предписывает игнорировать уровни иерархии и ссылается на любой узел-потомок, а не непосредственно на дочерний элемент. Например:

```
.//email // Все потомки <email> контекстного узла
//email // Все теги <email> документа (обратите внимание на ведущий слэш)
```

XPath-выражения могут ссылаться не только на XML-элементы, но и на их атрибуты. Для идентификации имени атрибута в качестве приставки используется символ `@`:

```
@id // Значение атрибута id контекстного узла
contact/@name // Значения атрибутов name потомков <contact>
```

Существует возможность отобрать множество элементов, возвращаемых XPath-выражением, по значению XML-атрибута. Например:

```
contact[@personal="true"] // Все теги <contact> с атрибутом personal="true"
```

Для извлечения текстового содержимого XML-элементов используется метод `text()`:

```
contact/email/text() // Текстовые узлы внутри тегов <email>
//text() // Все текстовые узлы в документе
```

Язык XPath различает пространства имен, благодаря чему существует возможность включать префиксы пространств имен в выражения:

```
//xsl:template // Отберет все элементы <xsl:template>
```

Разумеется, при вычислении XPath-выражения, использующего пространства имен, вы должны обеспечить отображение префиксов пространств имен на соответствующие им URL-адреса.

Эти примеры – всего лишь краткий обзор наиболее общих образцов применения XPath. Язык XPath имеет и другие синтаксические элементы и особенности, которые здесь не описаны. Как один из примеров – функция `count()`, которая возвращает количество узлов в получившемся множестве, а не само множество:

```
count(//email) // Число элементов <email> в документе
```

21.4.2. Выполнение XPath-выражений

В примере 21.10 приводится определение класса `XML.XPathExpression`, который одинаково работает и в IE, и в броузерах, соответствующих стандартам, таких как Firefox.

Пример 21.10. Выполнение XPath-выражений

```
/**
 * XML.XPathExpression - это класс, инкапсулирующий XPath-запрос
 * и ассоциированное с ним отображение префикса пространства имен в URL.
 * После того как объект XML.XPathExpression создан, он может
 * использоваться для многократного выполнения выражения (в одном
 * или более контекстах) посредством методов getNode() и getNodes().
 *
 * Первый аргумент конструктора класса - это текст XPath-выражения.
 *
 * Если выражение включает в себя какие-либо пространства имен XML, тогда
 * второй аргумент должен быть JavaScript-объектом, который отображает
 * префиксы пространств имен на URL-адреса, определяющие эти пространства
 * имен. Свойствами этого объекта должны быть префиксы, а значениями -
 * соответствующие им URL-адреса.
 */
XML.XPathExpression = function(xpathText, namespaces) {
    this.xpathText = xpathText;   // Сохранить текст выражения
    this.namespaces = namespaces; // И карту соответствий пространств имен

    if (document.createExpression) {
        // Если это W3C-совместимый броузер, использовать W3C API
        // для компиляции текста XPath-запроса
        this.xpathExpr =
            document.createExpression(xpathText,
                                      // Этой функции передается префикс
                                      // пространства имен, а она возвращает URL.
                                      function(prefix) {
                                          return namespaces[prefix];
                                      });
    }
    else {
        // Иначе предположить, что исполнение идет в IE и преобразовать
        // объект с пространствами имен в текстовую форму, как того требует IE
        this.namespaceString = "";
        if (namespaces != null) {
            for(var prefix in namespaces) {
```

```
                // Добавить пробел, если в строке уже что-то есть
                if (this.namespaceString) this.namespaceString += ' ';
                // И добавить пространство имен
                this.namespaceString += 'xmlns:' + prefix + '="' +
                    namespaces[prefix] + '"';
            }
        }
    }
};

/**
 * Метод getNodes() класса XML.XPathExpression. Он выполняет XPath-выражение
 * в указанном контексте. Аргумент context должен быть объектом Document
 * или Element. Возвращаемое значение - массив или объект, похожий на массив,
 * где содержатся узлы, соответствующие выражению.
 */
XML.XPathExpression.prototype.getNodes = function(context) {
    if (this.xpathExpr) {
        // Если это W3C-совместимый броузер, значит, выражение уже
        // скомпилировано в конструкторе. Осталось лишь вычислить
        // выражение в указанном контексте.
        var result =
            this.xpathExpr.evaluate(context,
                                    // Это - тип требуемого результата
                                    XPathResult.ORDERED_NODE_SNAPSHOT_TYPE,
                                    null);

        // Скопировать результаты в массив.
        var a = new Array(result.snapshotLength);
        for(var i = 0; i < result.snapshotLength; i++) {
            a[i] = result.snapshotItem(i);
        }
        return a;
    }
    else {
        // Если это не W3C-совместимый броузер, попытаться выполнить
        // выражение с использованием IE API.
        try {
            // Чтобы указать пространства имен, необходим объект Document
            var doc = context.ownerDocument;
            // Если контекст не имеет объекта ownerDocument, значит,
            // это и есть Document
            if (doc == null) doc = context;
            // Прием отображения префиксов на URL-адреса, характерный для IE
            doc.setProperty("SelectionLanguage", "XPath");
            doc.setProperty("SelectionNamespaces", this.namespaceString);

            // В IE объект Document не может быть контекстом - только Element,
            // таким образом, если контекст - это документ, использовать
            // вместо него documentElement
            if (context == doc) context = doc.documentElement;
            // Теперь с помощью IE-метода selectNodes() выполнить выражение
            return context.selectNodes(this.xpathText);
        }
        catch(e) {
```

```
            // Если IE API не работает, значит, нам просто не повезло
            throw "XPath не поддерживается этим броузером.";
        }
    }
}

/**
 * Метод getNode() класса XML.XPathExpression. Он выполняет XPath-выражение
 * в заданном контексте и возвращает единственный узел, соответствующий
 * выражению (или null, если совпадений не найдено). Если обнаружено
 * более одного совпадения, метод возвращает первое из них.
 * Реализация этого метода отличается от getNodes() только типом
 * возвращаемого значения.
 */
XML.XPathExpression.prototype.getNode = function(context) {
    if (this.xpathExpr) {
        var result =
            this.xpathExpr.evaluate(context,
                                    // Вернуть первое совпадение
                                    XPathResult.FIRST_ORDERED_NODE_TYPE,
                                    null);
        return result.singleNodeValue;
    }
    else {
        try {
            var doc = context.ownerDocument;
            if (doc == null) doc = context;
            doc.setProperty("SelectionLanguage", "XPath");
            doc.setProperty("SelectionNamespaces", this.namespaceString);
            if (context == doc) context = doc.documentElement;
            // В IE, вместо selectNodes, вызвать selectSingleNode
            return context.selectSingleNode(this.xpathText);
        }
        catch(e) {
            throw "XPath не поддерживается этим броузером.";
        }
    }
};

// Вспомогательная функция, которая создает объект XML.XPathExpression
// и вызывает его метод getNodes()
XML.getNodes = function(context, xpathExpr, namespaces) {
    return (new XML.XPathExpression(xpathExpr, namespaces)).getNodes(context);
};

// Вспомогательная функция, которая создает объект XML.XPathExpression
// и вызывает его метод getNode()
XML.getNode = function(context, xpathExpr, namespaces) {
    return (new XML.XPathExpression(xpathExpr, namespaces)).getNode(context);
};
```

21.4.3. Дополнительно о W3C XPath API

Из-за ограничений, присущих прикладному интерфейсу XPath в IE, программный код примера 21.10 способен обрабатывать только те запросы, которые воз-

вращают один или более узлов документа. В IE невозможно выполнить XPath-выражение, которое возвращало бы строку или число. Однако прикладной интерфейс для W3C позволяет это сделать с помощью, например, такого фрагмента:

```
// Определить количество тегов <p> в документе
var n = document.evaluate("count(//p)", document, null,
                          XPathResult.NUMBER_TYPE, null).numberValue;
// Извлечь текст второго абзаца
var text = document.evaluate("//p[2]/text()", document, null,
                             XPathResult.STRING_TYPE, null).stringValue;
```

По поводу этих двух простых примеров есть два замечания. Во-первых, для выполнения XPath-выражения без предварительной компиляции в них вызывается метод `document.evaluate()`. В противоположность этому в примере 21.10 применяется метод `document.createExpression()`, который компилирует XPath-выражение в форму, допускающую многократное использование скомпилированного выражения. Во-вторых, обратите внимание, что в этих примерах работа ведется с HTML-тегом `<p>` объекта `document`. В броузере Firefox XPath-выражения могут использоваться для работы как с XML-документами, так и с HTML-документами.

Дополнительные сведения о W3C XPath API вы найдете в четвертой части книги в разделах, посвященных объектам `Document`, `XPathExpression` и `XPathResult`.

21.5. Сериализация XML-документа

Иногда бывает удобно *сериализовать* XML-документ (или некоторые подэлементы документа), преобразовав его в строку. Это может понадобиться, например, чтобы отправить XML-документ в теле HTTP-запроса POST, сгенерированного с помощью объекта `XMLHttpRequest`. Нередко сериализация XML-документов и их элементов выполняется для использования в отладочных сообщениях!

В броузерах на базе Mozilla сериализация выполняется с помощью объекта `XMLSerializer`. В IE еще проще: с помощью свойства `xml` XML-объекта `Document` или `Element`, возвращающего содержимое документа или элемента в сериализованной форме.

В примере 21.11 приводится программный код, выполняющий сериализацию в броузерах Mozilla и IE.

Пример 21.11. Сериализация XML-документа

```
/**
 * Сериализует XML-документ или XML-элемент и возвращает его в виде строки.
 */
XML.serialize = function(node) {
    if (typeof XMLSerializer != "undefined")
        return (new XMLSerializer()).serializeToString(node);
    else if (node.xml) return node.xml;
    else throw "XML.serialize не поддерживается или не может сериализовать " + node;
};
```

21.6. Разворачивание HTML-шаблонов с использованием XML-данных

Одна из ключевых особенностей островков XML-данных заключается в том, что они могут использоваться механизмом автоматического разворачивания шаблонов, при котором данные из этих островков автоматически вставляются в HTML-элементы. Такого рода HTML-шаблоны в IE определяются добавлением в элементы атрибутов `datasrc` и `datafld` (где префикс «fld» означает «field» – поле).

В этом разделе описываются приемы работы с XML-данными, уже упоминавшиеся в начале главы, а также приемы создания средствами XPath и DOM улучшенного механизма разворачивания шаблонов, который будет работать в броузерах IE и Firefox. Шаблон – это любой HTML-элемент с атрибутом `datasource`. Значением этого атрибута должен быть идентификатор островка XML-данных или URL-адрес внешнего XML-документа. Кроме того, элемент шаблона должен иметь атрибут `foreach`, значением которого является XPath-выражение, возвращающее список XML-узлов, откуда должны извлекаться данные. Для каждого XML-узла, полученного в результате выполнения выражения `foreach`, в HTML-документ вставляется развернутая копия шаблона. Разворачивание шаблона производится в результате поиска внутри него всех элементов, имеющих атрибут `data`. Данный атрибут – это еще одно XPath-выражение, которое выполняется в контексте узла, полученного от выражения `foreach`. Выражение `data` выполняется вызовом метода `XML.getNode()`, а текст, содержащийся в полученном узле, используется как содержимое HTML-элемента, в котором этот атрибут определен.

Это описание станет понятнее после изучения конкретного примера. В примере 21.12 приводится простой HTML-документ, который включает островок XML-данных и использующий его шаблон. Разворачивание шаблона производится обработчиком события `onload`.

Пример 21.12. Островок XML-данных и HTML-шаблон

```
<html>
<!-- загрузить XML-утилиты для работы с островками данных и шаблонами -->
<head><script src="xml.js"></script></head>
<!-- Развернуть все шаблоны документа после загрузки -->
<body onload="XML.expandTemplates()">

<!-- Это островок XML-данных -->
<xml id="data" style="display:none"> <!-- скрытие с помощью CSS -->
  <contacts>
    <contact name="Able Baker"><email>able@example.com</email></contact>
    <contact name="Careful Dodger"><email>dodger@example.com</email></contact>
    <contact name="Eager Framer"><email>framer@example.com</email></contact>
  </contacts>
</xml>

<!-- Это обычные HTML-элементы -->
<table>
<tr><th>Имя</th><th>Адрес</th></tr>
<!-- Это шаблон. Данные берутся из островка с id="data". -->
<!-- Шаблон разворачивается и копируется для каждого тега <contact> -->
<tr datasource="#data" foreach="//contact">
```

```
<!-- В этот элемент вставляется значение атрибута "name" тега <contact> -->
<td data="@name"></td>
<!-- Здесь вставляется содержимое <email> - потомка узла <contact> -->
<td data="email"></td>
</tr> <!-- конец шаблона -->
</table>
</body>
</html>
```

Наиболее важной частью примера 21.12 является обработчик события `onload`**, который вызывает функцию** `XML.expandTemplates()`**. Реализация этой функции демонстрируется в примере 21.13. Программный код, который практически не зависит от платформы, основан на модели DOMLevel 1 и вспомогательных функциях** `XML.getNode()` **и** `XML.getNodes()`**, реализованных в примере 21.10 и предназначенных для работы с XPath-выражениями.**

Пример 21.13. Разворачивание HTML-шаблонов

```
/*
 * Разворачивает любые шаблоны, вложенные в элемент e. Если в каком-либо
 * из шаблонов используются XPath-выражения с пространствами имен, во втором
 * аргументе необходимо передать отображение префиксов пространств имен
 * на соответствующие им URL-адреса, как в случае с XML.XPathExpression()
 *
 * Если элемент e не указан, используется document.body. Обычно эта функция
 * вызывается без аргументов из обработчика события onload. В этом случае
 * она автоматически разворачивает все шаблоны.
 */
XML.expandTemplates = function(e, namespaces) {
    // Немножко подправить аргументы.
    if (!e) e = document.body;
    else if (typeof e == "string") e = document.getElementById(e);
    if (!namespaces) namespaces = null; // Значение undefined не работает

    // HTML-элемент является шаблоном, если имеет атрибут "datasource".
    // Рекурсивный поиск и разворачивание всех шаблонов.
    // Обратите внимание: шаблоны внутри шаблонов недопустимы.
    if (e.getAttribute("datasource")) {
        // Если это шаблон - развернуть его.
        XML.expandTemplate(e, namespaces);
    }
    else {
        // Иначе рекурсивно обойти все дочерние узлы. Прежде чем развернуть
        // шаблон, создается статическая копия потомка, чтобы развернутый
        // шаблон не мешал итерации.
        var kids = []; // Создать копию дочернего элемента
        for(var i = 0; i < e.childNodes.length; i++) {
            var c = e.childNodes[i];
            if (c.nodeType == 1) kids.push(e.childNodes[i]);
        }

        // Обойти все дочерние элементы
        for(var i = 0; i < kids.length; i++)
            XML.expandTemplates(kids[i], namespaces);
    }
```

```
};

/**
 * Разворачивает один указанный шаблон. Если XPath-выражение в шаблоне использует
 * имена пространств, вторым аргументом следует передать отображение
 * префиксов пространств имен на соответствующие им URL-адреса.
 */
XML.expandTemplate = function(template, namespaces) {
    if (typeof template=="string") template=document.getElementById(template);
    if (!namespaces) namespaces = null;        // Undefined does not work

    // Для начала определить, откуда брать данные для шаблона
    var datasource = template.getAttribute("datasource");

    // Если значение атрибута datasource начинается с '#', следовательно,
    // это имя островка XML-данных. Иначе это URL-адрес внешнего XML-файла.
    var datadoc;
    if (datasource.charAt(0) == '#')          // Получить островок данных
        datadoc = XML.getDataIsland(datasource.substring(1));
    else                                      // Или загрузить внешний документ
        datadoc = XML.load(datasource);

    // Теперь нужно определить, какие узлы в datasource будут служить
    // источниками данных. Если в шаблоне имеется атрибут foreach,
    // использовать его значение как XPath-выражение для получения списка узлов.
    // В противном случае задействовать все дочерние элементы элемента document.
    var datanodes;
    var foreach = template.getAttribute("foreach");
    if (foreach) datanodes = XML.getNodes(datadoc, foreach, namespaces);
    else {
        // Если атрибут "foreach" не задан, использовать дочерние
        // элементы элемента documentElement
        datanodes = [];
        for(var c=datadoc.documentElement.firstChild;c!=null;c=c.nextSibling)
            if (c.nodeType == 1) datanodes.push(c);
    }

    // Удалить элемент шаблона из его родительского элемента,
    // но запомнить родителя, а также nextSibling шаблона.
    var container = template.parentNode;
    var insertionPoint = template.nextSibling;
    template = container.removeChild(template);

    // Для каждого элемента массива datanodes обратно в контейнер вставляется
    // копия шаблона, но перед этим выполняется разворачивание всех дочерних
    // элементов копии, в которых присутствует атрибут "data".
    for(var i = 0; i < datanodes.length; i++) {
        var copy = template.cloneNode(true);          // Копировать шаблон
        expand(copy, datanodes[i], namespaces);       // Развернуть копию
        container.insertBefore(copy, insertionPoint); // Вставить копию
    }

    // Эта вложенная функция отыскивает все дочерние элементы для элемента e,
    // в котором определен атрибут data. Этот атрибут интерпретируется как
    // XPath-выражение и вычисляется в контексте datanode. Извлекает текст
    // из результата выполнения XPath-выражения и вставляет его как содержимое
```

```
    // разворачиваемого HTML-узла. Все остальное содержимое удаляется.
    function expand(e, datanode, namespaces) {
        for(var c = e.firstChild; c != null; c = c.nextSibling) {
            if (c.nodeType != 1) continue; // Только элементы
                var dataexpr = c.getAttribute("data");
                if (dataexpr) {
                    // Выполнить XPath-выражение в контексте datanode.
                    var n = XML.getNode(datanode, dataexpr, namespaces);
                    // Удалить все содержимое элемента
                    c.innerHTML = "";
                    // И вставить текст, полученный в результате
                    // выполнения XPath-выражения
                    c.appendChild(document.createTextNode(getText(n)));
                }

                // Если элемент не был развернут, обойти его рекурсивно.
                else expand(c, datanode, namespaces);
            }
        }

        // Эта вложенная функция извлекает текст DOM-узла,
        // выполняя рекурсию, если это необходимо.
        function getText(n) {
            switch(n.nodeType) {
            case 1: /* элемент */
                var s = "";
                for(var c = n.firstChild; c != null; c = c.nextSibling)
                    s += getText(c);
                return s;
            case 2: /* атрибут */
            case 3: /* текст */
            case 4: /* cdata */
                return n.nodeValue;
            default:
                return "";
            }
    }
};
```

21.7. XML и веб-службы

Веб-службы – это одна из важнейших областей использования XML, а SOAP – это популярный протокол для работы с веб-службами, который целиком основан на формате XML. В этом разделе я покажу, как с помощью объекта `XMLHttpRequest` и XPath-запросов выполнять SOAP-запросы к веб-службе.

JavaScript-код из примера 21.14 конструирует XML-документ, представляющий SOAP-запрос, и использует объект `XMLHttpRequest` для передачи запроса веб-службе. (Веб-служба возвращает обменный курс валют двух стран.) Затем с помощью XPath-запроса из тела SOAP-ответа, полученного от сервера, извлекается результат.

Прежде чем перейти к рассмотрению программного кода, необходимо сделать несколько замечаний. Во-первых, описание протокола SOAP далеко выходит за

рамки темы этой главы, поэтому в примере демонстрируются простые SOAP-запрос и SOAP-ответ без описания самого протокола и формата XML. Во-вторых, в примере не используются файлы на языке определения веб-служб (Web Services Definition Language, WSDL) для поиска сведений о веб-службе. Адрес сервера, метод и имена параметров жестко «зашиты» в программный код примера.

Третье замечание самое существенное. Использование веб-служб из клиентского JavaScript-кода строго ограничивается политикой общего происхождения (см. раздел 13.8.2). Напомню, что политика общего происхождения запрещает клиентскому сценарию соединяться и получать данные с сервера, не являющегося источником документа с этим сценарием. Это означает, что обычно JavaScript-сценарий, обращающийся к веб-службе, может быть полезен, только если документ, содержащий этот сценарий, хранится на том же сервере, что и сама веб-служба. Разработчики веб-служб могут использовать JavaScript для предоставления упрощенного HTML-интерфейса к своим веб-службам, но политика общего происхождения препятствует широкому применению клиентского JavaScript-кода для объединения на единственной веб-странице результатов вызова различных веб-служб из разных концов Интернета.

Чтобы запустить пример 21.14 в IE, необходимо ослабить действие политики общего происхождения. Для этого выберите команду Сервис→Свойства обозревателя, в появившемся диалоговом окне перейдите на вкладку Безопасность, щелчком выделите значок Интернет и щелкните на кнопке Другой. В следующем диалоговом окне прокрутите список параметров безопасности и найдите группу переключателей Доступ к источникам данных за пределами домена. Обычно в этой группе (так и должно быть) установлен переключатель Отключить. Для проверки нашего примера установите переключатель Предлагать.

Чтобы иметь возможность проверить пример 21.14 в Firefox, в программный код примера включен вызов специфичного для Firefox метода `enablePrivilege()`. Этот метод запросит у пользователя разрешение на выдачу сценарию расширенных привилегий, чтобы преодолеть ограничения политики общего происхождения. Данный прием будет работать в случае запуска примера из локальной файловой системы со спецификатором `file:` в URL-адресе, но не будет работать, если загрузить пример с веб-сервера (если только в сценарии не будет цифровой подписи, описание которой далеко выходит за рамки темы этой книги).

Теперь, когда все необходимые замечания сделаны, можно перейти к изучению программного кода.

Пример 21.14. Запрос веб-службы с использованием протокола SOAP

```
/**
 * Данная функция возвращает обменный курс валют двух стран.
 * Значение обменного курса определяется путем обращения по протоколу SOAP
 * к веб-службе, размещенной на сервере XMethods (http://www.xmethods.net).
 * Служба предназначена исключительно для демонстрационных целей.
 * Она гарантирует свою доступность или точность возвращаемых данных.
 * Пожалуйста, не перегружайте сервер XMethod слишком частыми запусками этого примера.
 * Подробности см. по адресу: http://www.xmethods.net/v2/demoguidelines.html
 */
function getExchangeRate(country1, country2) {
    // В Firefox необходимо запросить у пользователя разрешение
```

```
// на получение необходимых привилегий. Специальные привилегии нужны по той
// простой причине, что выполняется обращение к веб-серверу, не являющемуся
// источником документа с этим сценарием. Привилегия UniversalXPConnect
// позволяет отправлять запросы серверу с помощью объекта XMLHttpRequest,
// а привилегия UniversalBrowserRead – просматривать ответ сервера.
// В IE вместо этого пользователь должен установить переключатель
// "Предлагать" в группе "Доступ к источникам данных за пределами домена"
// диалогового окна Сервис->Свойства обозревателя->Безопасность->Другой.
if (typeof netscape != "undefined") {
    netscape.security.PrivilegeManager.
    enablePrivilege("UniversalXPConnect UniversalBrowserRead");
}

// Создать функцию XMLHttpRequest для запуска SOAP-запроса.
// Данная вспомогательная функция определена в последней главе.
var request = HTTP.newRequest();

// Запрос будет послан методом POST в синхронном режиме
request.open("POST", "http://services.xmethods.net/soap", false);

// Установить некоторые заголовки: тело запроса POST содержит XML
request.setRequestHeader("Content-Type", "text/xml");

// Этот заголовок является обязательным для протокола SOAP
request.setRequestHeader("SOAPAction", '""');

// Отправить сформированный SOAP-запрос серверу
request.send(
    '<?xml version="1.0" encoding="UTF-8"?>' +
    '<soap:Envelope' +
    ' xmlns:ex="urn:xmethods-CurrencyExchange"' +
    ' xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"' +
    ' xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"' +
    ' xmlns:xs="http://www.w3.org/2001/XMLSchema"' +
    ' xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">' +
    ' <soap:Body ' +
    ' soap:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">'+
    ' <ex:getRate>' +
    ' <country1 xsi:type="xs:string">' + country1 + '</country1>' +
    ' <country2 xsi:type="xs:string">' + country2 + '</country2>' +
    ' </ex:getRate>' +
    ' </soap:Body>' +
    '</soap:Envelope>'
);

// Если был получен HTTP-код ошибки, возбудить исключение
if (request.status != 200) throw request.statusText;

// Этот XPath-запрос извлекает элемент <getRateResponse> из документа
var query = "/s:Envelope/s:Body/ex:getRateResponse";

// Этот объект определяет пространства имен, используемые в запросе
var namespaceMapping = {
    s: "http://schemas.xmlsoap.org/soap/envelope/", // пространство имен SOAP
    ex: "urn:xmethods-CurrencyExchange" // пространство имен, определяемое службой
};
```

```
    // Извлечь элемент <getRateResponse> из документа ответа
    var responseNode=XML.getNode(request.responseXML, query,
                                 namespaceMapping);

    // Фактический результат находится в текстовом узле внутри узла <Result>
    // внутри <getRateReponse>
    return responseNode.firstChild.firstChild.nodeValue;
}
```

21.8. E4X: EcmaScript для XML

Расширение EcmaScript для XML, больше известное как E4X, – это стандартное расширение[1] языка JavaScript, которое определяет ряд дополнительных средств для работы с XML-документами. К моменту написания этих строк расширение E4X еще не получило широкого распространения. Броузер Firefox 1.5 поддерживает его, также оно доступно в Rhino версии 1.6 – интерпретаторе JavaScript, реализованном на языке Java. Компания Microsoft не планирует поддержку E4X в IE 7 и не совсем понятно, появится ли такая поддержка в других броузерах, и если появится, то когда.

Несмотря на то, что расширение E4X является официальным стандартом, оно еще не получило достаточно широкого распространения, чтобы описывать его в этой книге полностью. Однако несмотря на ограниченную распространенность, уникальные возможности E4X, несомненно, заслуживают упоминания. Этот раздел представляет собой обзор расширения E4X в примерах. Возможно, в будущих изданиях книги это описание будет расширено.

Самое поразительное в E4X – то, что синтаксис XML становится частью языка JavaScript, благодаря чему появляется возможность включать XML-литералы непосредственно в JavaScript-код:

```
// Создать XML-объект
var pt =
    <periodictable>
        <element id="1"><name>Водород</name></element>
        <element id="2"><name>Гелий</name></element>
        <element id="3"><name>Литий</name></element>
    </periodictable>;

// Добавить новый элемент в таблицу
pt.element += <element id="4"><name>Бериллий</name></element>;
```

В синтаксисе литералов расширения E4X фигурные скобки используются в качестве экранирующих символов, что позволяет размещать JavaScript-выражения прямо внутри XML-данных. Например, вот другой способ создания XML-элемента, продемонстрированного в предыдущем примере:

```
pt = <periodictable></periodictable>;          // Изначально таблица пуста
var elements = ["Водород", "Гелий", "Литий"]; // Добавить элементы
// Создать XML-теги, используя содержимое массива
```

---

[1] Расширение E4X описывается стандартом ECMA-357. Официальные спецификации можно найти по адресу *http://www.ecmainternational.org/publications/standards/Ecma-357.htm*.

```
for(var n = 0; n < elements.length; n++) {
    pt.element += <element id={n+1}><name>{elements[n]}</name></element>;
}
```

В дополнение к синтаксису литералов существует возможность работать со строками в формате XML. Следующий фрагмент добавляет еще один элемент к периодической таблице:

```
pt.element += new XML('<element id="5"><name>Бор</name></element>');
```

Для работы с фрагментами XML-текста вместо метода `XML()` следует использовать метод `XMLList()`:

```
pt.element += new XMLList('<element id="6"><name>Carbon</name></element>' +
                          '<element id="7"><name>Nitrogen</name></element>');
```

После того как XML-документ будет определен, обращаться к нему можно с помощью интуитивно понятного синтаксиса E4X:

```
var elements = pt.element;   // Получить список всех тегов <element>
var names = pt.element.name; // Список всех тегов <name>
var n = names[0];            // "Водород": содержимое нулевого тега <name>.
```

Расширение E4X добавляет также новый синтаксис для работы с XML-объектами. Две точки (`..`) – это оператор доступа к узлу-потомку; его можно использовать вместо обычного оператора доступа к члену (`.`):

```
// Это еще один способ получить список всех тегов <name>
var names2 = pt..name;
```

E4X позволяет использовать даже оператор группировки:

```
// Получить список всех потомков всех тегов <element>.
// Это еще один способ получить список всех тегов <name>.
var names3 = pt.element.*;
```

В E4X имена атрибутов отличаются от имен тегов благодаря символу `@` (этот синтаксис заимствован из XPath). Например, запросить значение атрибута можно следующим образом:

```
// Получить атомный номер гелия
var atomicNumber = pt.element[1].@id;
```

Оператор группировки для имен атрибутов объединяет оба знака (`@*`):

```
// Список всех атрибутов всех тегов <element>
var atomicNums = pt.element.@*;
```

Расширение включает даже удивительно мощный и краткий синтаксис фильтрации списков с произвольным условием:

```
// Получить список всех элементов и отфильтровать его так, чтобы
// он включал только те из них, которые имеют значение атрибута id < 3
var lightElements = pt.element.(@id < 3);

// Получить список всех тегов <element> и отфильтровать его так, чтобы
// он включал только те из них, имена которых начинаются с символа "Б".
// Затем получить список тегов <name> из оставшихся после фильтрации тегов <element>.
var bElementNames = pt.element.(name.charAt(0) == 'Б').name;
```

В E4X определяется новый оператор цикла для обхода списка XML-тегов и их атрибутов. Цикл `for/each/in` напоминает цикл `for/in`, только вместо обхода свойств объекта выполняется обход значений свойств объекта:

```
// Вывести имена всех элементов периодической таблицы
// (Предполагается, что была определена функция print().)
for each (var e in pt.element) {
    print(e.name);
}

// Вывести атомные числа элементов
for each (var n in pt.element.@*) print(n);
```

В броузерах, поддерживающих E4X, данный цикл `for/each/in` может с успехом использоваться для обхода массивов.

E4X-выражения допускается указывать слева от оператора присваивания. Это позволяет изменять существующие и добавлять новые теги и атрибуты:

```
// Добавить новый атрибут в тег <element> для Водорода
// и новый дочерний элемент, чтобы он выглядел так:
//
// <element id="1" symbol="H">
//   <name>Водород</name>
//   <weight>1.00794</weight>
// </element>
//
pt.element[0].@symbol = "H";
pt.element[0].weight = 1.00794;
```

С помощью стандартного оператора `delete` столь же просто удалять теги и атрибуты:

```
delete pt.element[0].@symbol; // Удалить атрибут
delete pt..weight;            // Удалить все теги <weight>
```

Расширение E4X спроектировано так, чтобы выполнять наиболее распространенные операции с XML-документами с использованием синтаксиса этого языка. Кроме того, E4X определяет методы XML-объектов. Вот пример вызова метода `insertChildBefore()`:

```
pt.insertChildBefore(pt.element[1],
                     <element id="1"><name>Дейтерий</name></element>);
```

Обратите внимание: объекты, созданные и управляемые E4X-выражениями, являются XML-объектами. Это не DOM-объекты `Node` и `Element`, и с ними нельзя работать, используя DOM API. Стандарт E4X определяет необязательный XML-метод `domNode()`, который возвращает DOM-объект `Node`, эквивалентный XML-объекту, но в Firefox 1.5 этот метод не реализован. Аналогично стандарт E4X утверждает, что DOM-объект `Node` может передаваться конструктору `XML()` для получения E4X-эквивалента DOM-дерева. Эта возможность также не реализована в Firefox 1.5, что ограничивает область применения E4X в клиентских JavaScript-сценариях.

Расширение E4X полностью поддерживает пространства имен и включает языковые конструкции и API для работы с пространствами имен XML. Однако для простоты представленные примеры не используют этот синтаксис.

22

Работа с графикой на стороне клиента

В этой главе рассказывается о том, как работать с графикой из JavaScript-сценариев. Она начинается с описания традиционных приемов создания визуальных эффектов, таких как смена изображений (когда одно статическое изображение сменяется другим при наведении указателя мыши). Затем рассказывается о том, как создавать собственные графические изображения. Комбинирование JavaScript-кода и CSS-стилей позволяет рисовать вертикальные и горизонтальные линии и прямоугольники, чего обычно вполне достаточно для создания как простеньких рисунков, так и сложных гистограмм.

Далее мы перейдем к рассмотрению технологий векторной графики, которые предоставляют намного более широкие возможности по созданию графических изображений на стороне клиента. Способность воспроизводить на стороне клиента сложные графические изображения имеет важное значение по нескольким причинам:

- Объем программного кода, создающего изображение на стороне клиента, обычно много меньше, чем объем самого изображения, что позволяет сберечь существенную долю полосы пропускания.
- Динамическое воспроизведение графических изображений потребляет существенные ресурсы центрального процессора. Передав эту задачу клиенту (у которого обычно всегда имеется некоторый резерв мощности процессора), можно существенно снизить нагрузку на сервер и немного сэкономить на стоимости аппаратных средств для него.
- Воспроизведение графики на стороне клиента прекрасно согласуется с положениями архитектуры Ajax, в которой серверы призваны поставлять данные, а клиенты представлять эти данные.

В эту главу включено описание пяти технологий создания векторной графики, которые могут использоваться в JavaScript-сценариях на стороне клиента:

- Масштабируемая векторная графика (Scalable Vector Graphics, SVG) – это W3C-стандарт XML-подобного языка создания графических изображений. В чистом виде SVG поддерживается в Firefox 1.5, а в других броузерах мас-

штабируемая векторная графика поддерживается модулями расширения. Поскольку графические изображения в формате SVG – это XML-документы, они могут динамически создаваться в JavaScript-сценариях.

- Векторный язык разметки (Vector Markup Language, VML) – это альтернатива SVG от компании Microsoft. Данная технология мало известна, хотя и доступна в Internet Explorer начиная с версии 5.5. Как и в случае SVG, графические изображения в формате VML – это XML-документы, потому они тоже могут строиться динамически на стороне клиента.
- HTML-тег `<canvas>` непосредственно предоставляет прикладной интерфейс (API) для рисования из JavaScript-сценариев. Впервые этот тег появился в броузере Safari 1.3, а затем перекочевал в Firefox 1.5 и Opera 9.
- Flash-плеер доступен в виде модулей расширения для подавляющего большинства основных веб-броузеров. Впервые прикладной интерфейс для рисования появился в Flash-плеере версии 6, а в версии 8 этот интерфейс был максимально приспособлен для использования из клиентских JavaScript-сценариев.
- Наконец, язык программирования Java поддерживает весьма мощный прикладной интерфейс создания изображений и доступен во многих веб-броузерах в виде модулей расширения компании Sun Microsystems. Как описывается в главах 12 и 23, JavaScript-сценарии могут вызывать методы Java-апплетов, а в броузерах на базе Mozilla – вызывать Java-методы даже в отсутствие апплетов. Такая степень взаимодействия с Java позволяет JavaScript-сценарию использовать на стороне клиента мощный прикладной Java-интерфейс создания графических изображений.

Однако прежде чем погружаться в эти сложные технологии создания изображений, рассмотрим сначала самые основы.

22.1. Работа с готовыми изображениями

Готовые изображения могут включаться в HTML-страницу с помощью тега `<img>`. Подобно любому HTML-элементу, тег `<img>` является частью модели DOM и потому может управляться, как и любой другой элемент документа. В данном разделе описаны наиболее общие приемы.

22.1.1. Изображения и модель DOM Level 0

Изображения были одним из первых управляемых HTML-элементов, и модель DOM Level 0 позволяет обращаться к ним посредством массива `images[]` объекта `Document`. Каждый элемент данного массива – это объект `Image`, представляющий свой тег `<img>` в документе. Полное описание объекта `Image` вы найдете в четвертой части книги. Обращаться к объектам `Image` можно также с использованием методов модели DOM Level 1, таких как `getElementById()` и `getElementsByTagName()` (см. глава 15).

Объекты `Image` содержатся в массиве `document.images[]` в том порядке, в котором они встречаются в документе. Однако иногда удобнее получать доступ к изображениям по именам. Если в теге `<img>` определен атрибут `name`, доступ к изображению может быть получен по значению этого атрибута. Рассмотрим следующий пример тега `<img>`:

```
<img name="nextpage" src="nextpage.gif">
```

Предположим, что в документе нет другого тега <img> с тем же значением атрибута name, тогда доступ к соответствующему объекту Image может быть получен любым из следующих двух способов:

```
document.images.nextpage
document.images["nextpage"]
```

Если в документе нет никаких других тегов с тем же значением атрибута name, тогда данный объект Image может быть доступен даже как свойство объекта document:

```
document.nextpage
```

22.1.2. Традиционный прием смены изображений

Главная особенность объекта Image заключается в том, что его свойство src доступно и для чтения, и для записи. Прочитав значение этого свойства, можно получить URL-адрес, с которого было загружено изображение. Еще важнее то, что можно установить свойство src и тем самым заставить броузер загрузить и отобразить новое изображение на том же месте.

Возможность динамической замены одного изображения другим в HTML-документе открывает доступ к любым специальным эффектам, начиная от анимации и заканчивая цифровыми часами, которые сами обновляются в режиме реального времени. На практике чаще всего этот прием смены изображений реализуется, когда указатель мыши наводится на изображение. (Чтобы избежать неприятных визуальных эффектов, новое изображение должно иметь те же размеры, что и предыдущее.) Когда изображение размещается внутри тега гиперссылки, эффект смены изображений является мощным побудительным мотивом, приглашающим пользователя щелкнуть на изображении.[1] Следующий фрагмент HTML-кода выводит изображение в теге <a> и с помощью обработчиков событий onmouseover и onmouseout создает эффект смены изображений:

```
<a href="help.html"
   onmouseover="document.helpimage.src='images/help_rollover.gif';"
   onmouseout="document.helpimage.src='images/help.gif';">
<img name="helpimage" src="images/help.gif" border="0">
</a>
```

Обратите внимание: в этом фрагменте тег <img> имеет атрибут name, что облегчает обращение к соответствующему объекту Image из обработчиков событий тега <a>. Установка атрибута border препятствует появлению синей рамки гиперссылки вокруг изображения. Все необходимое выполняется в обработчиках событий тега <a>: они меняют выводимое изображение, просто записывая в свойство src URL-адрес требуемого изображения. Чтобы эффект наблюдался и в старых броузерах,

---

[1] Обсуждение эффекта смены изображений не будет полным, если не упомянуть, что этот эффект может быть реализован с помощью CSS-псевдокласса :hover, изменяющего различные фоновые CSS-изображения в элементах, на которые наводится указатель мыши. К сожалению, реализация смены изображений на базе CSS сопряжена с трудностями, обусловленными несовместимостью броузеров. На практике псевдокласс :hover чаще используется для создания эффектов в текстовых, а не графических гиперссылках.

в которых обработчики этих событий поддерживаются только в отдельных тегах, таких как <a>, обработчики событий были помещены в тег <a>. Практически в любом современном броузере обработчики событий могут включаться непосредственно в тег <img>, что упрощает поиск объекта Image. В этом случае обработчик события мог бы сослаться на объект Image с помощью ключевого слова this:

```
<img src="images/help.gif"
     onmouseover="this.src='images/help_rollover.gif'"
     onmouseout="this.src='images/help.gif'">
```

Эффект смены изображений обычно означает возможность щелкнуть на изображении, поэтому такого рода теги <img> должны заключаться в тег <a> или предусматривать обработчик события onclick.

22.1.3. Невидимые изображения и кэширование

Чтобы радовать глаз, эффект смены изображений и родственные ему эффекты должны иметь минимальное время отклика. Это означает, что необходим некоторый способ, гарантирующий предварительную загрузку всех необходимых изображений в кэш броузера. Чтобы принудительно поместить изображение в кэш, нужно сначала создать объект Image с помощью конструктора Imange(). Затем, записав в свойство src требуемый URL-адрес, загрузить изображение. Этот объект не добавляется в документ, поэтому хотя изображение будет невидимо, броузер загрузит его и поместит в свой кэш. Позднее, когда тот же URL-адрес будет использоваться для изменения изображения, находящегося на экране, изображение быстро загрузится из кэша броузера.

Фрагмент кода, воспроизводящий эффект смены изображений, который был продемонстрирован в предыдущем разделе, не выполняет предварительную загрузку изображений, поэтому пользователь может заметить задержку при смене изображений, когда первый раз наведет указатель мыши на изображение. Чтобы исправить ситуацию, необходимо немного изменить код:

```
<script>(new Image()).src = "images/help_rollover.gif";</script>
<img src="images/help.gif"
     onmouseover="this.src='images/help_rollover.gif'"
     onmouseout="this.src='images/help.gif'">
```

22.1.4. Ненавязчивая смена изображений

Только что продемонстрированный фрагмент содержит один тег <script> и два атрибута обработчиков событий с JavaScript-кодом, чтобы реализовать единственный эффект смены изображений. Это прекрасный пример *навязчивого* JavaScript-кода. Хотя примеры смешения представления (HTML-разметка) с поведением (JavaScript-код) встречаются не так уж редко, лучше избегать такой практики, если такая возможность есть. Особенно в случаях, когда JavaScript-код затрудняет понимание HTML-кода. В примере 22.1 приводится функция, которая добавляет эффект смены изображений к указанному элементу <img>.

Пример 22.1. Добавление эффекта смены изображений

```
/**
 * Добавляет эффект смены изображений к заданному тегу <img>, вставляя обработчики
 * событий, которые будут изменять URL-адрес изображения при наведении указателя мыши.
```

```
 *
 * Если аргумент img содержит строку, выполняется поиск элемента
 * по значению атрибута id или name.
 *
 * Это метод устанавливает свойства обработчиков событий onmouseover
 * и onmouseout в указанный элемент, перекрывая и отключая любые обработчики,
 * определенные в этих свойствах ранее.
 */
function addRollover(img, rolloverURL) {
    if (typeof img == "string") { // Если img - это строка,
        var id = img;             // значит, это id, а не объект Image
        img = null;               // и потому у нас еще нет объекта.

        // Прежде всего, необходимо отыскать изображение по атрибуту id
        if (document.getElementById) img = document.getElementById(id);
        else if (document.all) img = document.all[id];

        // Если по атрибуту id отыскать не удалось, попробовать отыскать
        // по атрибуту name.
        if (!img) img = document.images[id];

        // Если не удалось найти изображение, ничего не делать и тихо выйти
        if (!img) return;
    }

    // Если элемент найден, но это не тег <img>, также ничего больше не делать
    if (img.tagName.toLowerCase() != "img") return;

    // Запомнить первоначальный URL-адрес изображения
    var baseURL = img.src;

    // Загрузить сменное изображение в кэш броузера
    (new Image()).src = rolloverURL;

    img.onmouseover = function() { img.src = rolloverURL; }
    img.onmouseout = function() { img.src = baseURL; }
}
```

Функция `addRollover()`, объявленная в примере 22.1, «не совсем» ненавязчивая, потому что для ее использования по-прежнему требуется включать в HTML-код сценарий, который вызывал бы эту функцию. Для достижения поставленной цели – сделать реализацию эффекта смены изображений по-настоящему ненавязчивой – необходимо каким-то образом без JavaScript-кода указать, какие изображения должны меняться, и задать URL-адреса сменных изображений. Самый простой способ – включить в теги `<img>` ложные атрибуты. Например, изображение с эффектом смены можно обозначить так:

```
<img src="normalImage.gif" rollover="rolloverImage.gif">
```

Используя такое соглашение по оформлению изображений, можно без труда отыскать все изображения, которые должны меняться, и настраивать реализацию эффекта с помощью функции `initRollovers()` – ее определение приводится в примере 22.2.

Пример 22.2. Добавление эффектов смены изображений ненавязчивым образом

```
/**
 * Находит все теги <img> в документе, которые имеют атрибут "rollover".
```

```
 * Значение этого атрибута используется как URL-адрес сменного изображения,
 * выводимого, когда указатель мыши наводится на изображение;
 * устанавливает соответствующие обработчики событий, с помощью
 * которых воспроизводится эффект смены изображений.
 */
function initRollovers() {
    var images = document.getElementsByTagName("img");
    for(var i = 0; i < images.length; i++) {
        var image = images[i];
        var rolloverURL = image.getAttribute("rollover");
        if (rolloverURL) addRollover(image, rolloverURL);
    }
}
```

Все, что осталось сделать, – это гарантировать запуск метода `initRollovers()` после загрузки документа. Следующий программный код должен работать в современных броузерах:

```
if (window.addEventListener)
    window.addEventListener("load", initRollovers, false);
else if (window.attachEvent)
    window.attachEvent("onload", initRollovers);
```

Более подробное обсуждение обработчика события `onload` вы найдете в главе 17.

Обратите внимание: если объединить функции `addRollover()` и `initRollovers()` в одном файле с программным кодом, выполняющим регистрацию обработчика события, получится полностью ненавязчивое решение по реализации эффекта смены изображений. Все, что необходимо для получения эффекта смены изображений, –просто подключить получившийся файл с программным кодом в теге `<script src=>` и вставить атрибут `rollover` в необходимые теги `<img>`.

Если необходимо соблюсти строгое соответствие HTML-файлов стандартам языка разметки, а потому нельзя использовать нестандартный атрибут `rollover` в тегах `<img>`, перейдите на XHTML и задействуйте для нового атрибута пространство имен XML. В примере 22.3 демонстрируется версия функции `initRollovers()`, различающая пространства имен. Однако следует отметить, что эта версия функции не работает в Internet Explorer 6, потому что этот броузер не воспринимает DOM-методы, которые поддерживают пространств имен.

Пример 22.3. Инициализация эффекта смены изображений средствами XHTML с использованием пространств имен

```
/**
 * Находит все теги <img> в документе, которые имеют атрибут "ro:src".
 * Значение этого атрибута используется как URL-адрес сменного изображения, выводимого,
 * когда указатель мыши наводится на изображение, и устанавливает соответствующие
 * обработчики событий, с помощью которых достигается эффект смены изображений.
 * Префикс ro: пространства имен должен отображаться на URI-адрес
 * "http://www.davidflanagan.com/rollover"
 */
function initRollovers() {
    var images = document.getElementsByTagName("img");
    for(var i = 0; i < images.length; i++) {
        var image = images[i];
```

```
            var rolloverURL = image.getAttributeNS(initRollovers.xmlns, "src");
            if (rolloverURL) addRollover(image, rolloverURL);
        }
    }

    // Это вымышленный URI-адрес для нашего пространства имен "ro:"
    initRollovers.xmlns = "http://www.davidflanagan.com/rollover";
```

22.1.5. Анимация изображений

Еще один довод в пользу манипулирования свойством `src` тега `<img>` – это способность к анимации; когда смена изображений происходит достаточно часто, создается иллюзия плавного движения. Типичное применение этой методики – отображение серии погодных карт, иллюстрирующих существовавший или прогнозируемый процесс развития штормовой ситуации в часовых интервалах за двухдневный период.

В примере 22.4 приводится определение класса `ImageLoop`, с помощью которого создаются подобного рода эффекты. Он демонстрирует те же приемы работы со свойством `src` и предварительной загрузки изображений, которые были показаны в примере 22.1. Здесь также добавлен обработчик события `onload` объекта `Image`, который определяет момент окончания загрузки изображения (или, в данном случае, – серии изображений). Программный код, реализующий анимацию, управляется методом `Window.setInterval()`, который сам по себе чрезвычайно прост: он наращивает номер кадра и записывает в свойство `src` указанного тега `<img>` URL-адрес изображения для следующего кадра.

Вот пример HTML-файла, в котором используется класс `ImageLoop`:

```
<head>
<script src="ImageLoop.js"></script>
<script>
var animation =
    new ImageLoop("loop", 5,["images/0.gif", "images/1.gif", "images/2.gif",
                             "images/3.gif", "images/4.gif", "images/5.gif",
                             "images/6.gif", "images/7.gif", "images/8.gif"]);
</script>
</head>
<body>
<img id="loop" src="images/loading.gif">
<button onclick="animation.start()">Start</button>
<button onclick="animation.stop()">Stop</button>
</body>
```

Программный код примера 22.4 получился несколько сложнее, чем можно было бы ожидать, потому что и обработчик события `Image.onload` и функция таймера `Window.setInterval()` вызывают функции как функции, а не как методы. По этой причине в конструкторе `ImageLoop()` потребовалось определить вложенные функции, «знающие», как взаимодействовать с вновь созданным объектом `ImageLoop`.

Пример 22.4. Анимация

```
/**
 * ImageLoop.js: Класс ImageLoop для создания эффекта анимации
 *
 * Аргументы конструктора:
```

```
 *   imageId:   идентификатор тега <img>, в котором воспроизводится анимация
 *   fps:       количество кадров в секунду
 *   frameURLs: массив URL-адресов, по одному на каждый кадр в анимации
 *
 * Общедоступные методы:
 *   start(): начинает анимацию (но ждет, пока загрузятся все кадры)
 *   stop():  останавливает анимацию
 *
 * Общедоступные свойства:
 * loaded: true - если все кадры были загружены, иначе - false
 */
function ImageLoop(imageId, fps, frameURLs) {
    // Запомнить id элемента. Не искать его сейчас, потому что конструктор
    // может быть вызван еще до того, как документ будет полностью загружен.
    this.imageId = imageId;
    // Рассчитать время задержки между кадрами
    this.frameInterval = 1000/fps;
    // Создать массив, где будут храниться объекты Image для каждого кадра
    this.frames = new Array(frameURLs.length);

    this.image = null;        // Элемент <img>, найденный по атрибуту id
    this.loaded = false;      // Еще не все изображения загружены
    this.loadedFrames = 0;    // Количество загруженных кадров
    this.startOnLoad = false; // Начать воспроизведение по окончании загрузки?
    this.frameNumber = -1;    // Текущий отображаемый кадр
    this.timer = null;        // Возвращаемое значение функции setInterval()

    // Инициализировать массив frames[] и загрузить изображения
    for(var i = 0; i < frameURLs.length; i++) {
        this.frames[i] = new Image( );             // Создать объект Image
        // Зарегистрировать обработчик события, чтобы узнать,
        // когда будет загружено изображение
        this.frames[i].onload = countLoadedFrames; // Определяется позже
        this.frames[i].src = frameURLs[i];         // Загрузить изображение
    }

    // Эта вложенная функция - обработчик события, который подсчитывает
    // количество загруженных кадров. Когда все изображения будут загружены,
    // устанавливает флаг и в случае необходимости начинает анимацию.
    var loop = this;
    function countLoadedFrames() {
        loop.loadedFrames++;
        if (loop.loadedFrames == loop.frames.length) {
            loop.loaded = true;
            if (loop.startOnLoad) loop.start();
        }
    }

    // Далее определяется функция, которая отображает следующий кадр анимации.
    // Эта функция не может быть обычным методом, т. к. setInterval() может
    // вызывать только функции, а не методы.
    // Поэтому здесь создается замыкание, включающее ссылку на объект ImageLoop
    this._displayNextFrame = function() {
        // Сначала нарастить номер кадра. Оператор деления по модулю (%)
        // выполняет переход от последнего кадра к первому
```

```
            loop.frameNumber = (loop.frameNumber+1)%loop.frames.length;
            // Записать в свойство src URL-адрес нового кадра
            loop.image.src = loop.frames[loop.frameNumber].src;
        };
    }

    /**
     * Этот метод начинает анимацию ImageLoop. Если загрузка кадров еще
     * не закончилась, он просто взводит флаг, в результате чего анимация
     * начинается автоматически по окончании загрузки
     */
    ImageLoop.prototype.start = function() {
        if (this.timer != null) return; // Анимация уже начата
        // Если загрузка еще не закончилась, установить флаг запуска
        if (!this.loaded) this.startOnLoad = true;
        else {
            // Если элемент <img> еще не был найден по id, сделать это сейчас
            if (!this.image) this.image = document.getElementById(this.imageId);
            // Сразу же отобразить первый кадр
            this._displayNextFrame();
            // И взвести таймер для воспроизведения последующих кадров
            this.timer = setInterval(this._displayNextFrame, this.frameInterval);
        }
    };

    /** Останавливает анимацию ImageLoop */
    ImageLoop.prototype.stop = function() {
        if (this.timer) clearInterval(this.timer);
        this.timer = null;
    };
```

22.1.6. Прочие свойства изображений

В дополнение к обработчику события `onload`, продемонстрированному в примере 22.4, объект `Image` поддерживает еще два обработчика. Обработчик события `onerror` вызывается в случае ошибки в процессе загрузки изображения, например, когда URL-адрес ссылается на поврежденный файл изображения. Обработчик события `onabort` вызывается, когда пользователь отменяет загрузку изображения (например, щелкнув на кнопке Остановить в броузере) до того, как она завершится. Для любых изображений вызывается один (и только один) из этих обработчиков.

Каждый объект `Image` обладает также свойством `complete`. В этом свойстве находится значение `false`, пока изображение не загружено; оно изменяется на `true`, когда изображение полностью загружается или когда броузер останавливается при попытке загрузить изображение. Другими словами, свойство `complete` получает значение `true` только после того, как будет вызван один из трех обработчиков событий.

Другие свойства объекта `Image` являются просто отражениями атрибутов тега `<img>`. В современных броузерах эти свойства доступны и для чтения, и для записи, а потому могут использоваться JavaScript-сценариями для динамического изменения размеров изображения, заставляя броузер растягивать или сжимать картнку.

22.2. Графика и CSS

Каскадные таблицы стилей описывались в главе 16, где вы узнали, как с помощью CSS-стилей воспроизводить DHTML-эффекты. С помощью стилей можно также рисовать несложные графические элементы: свойство `background-color` позволяет создать прямоугольник со сплошной цветной заливкой, а свойство `border` — контур прямоугольника. Кроме того, такие свойства, как `border-left` и `border-top`, предоставляют возможность нарисовать лишь одну сторону прямоугольника, что в результате дает вертикальные и горизонтальные линии. В броузерах, поддерживающих стили, эти линии можно рисовать даже пунктиром или штрихами!

Это не так много, но в комбинации со средствами абсолютного позиционирования из этих простых прямоугольников и линий можно строить диаграммы, как показано на рис. 22.1 и 22.2. В следующих подразделах рассказывается, как были созданы эти рисунки.

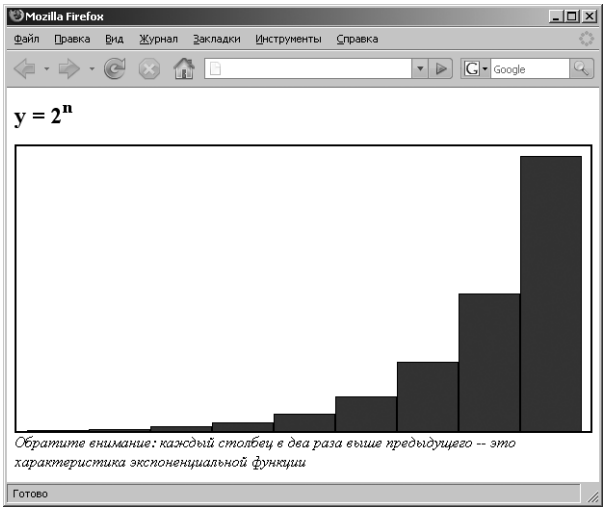

Рис. 22.1. *Гистограмма, нарисованная средствами CSS*

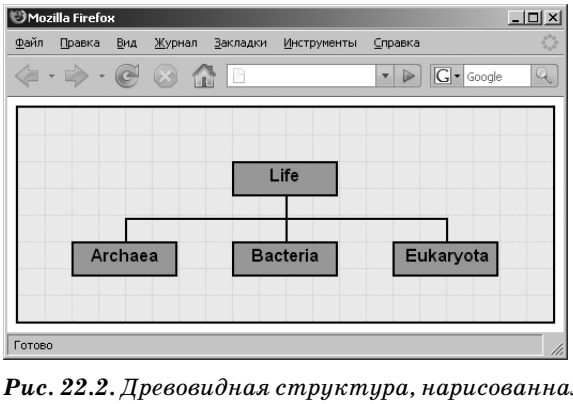

Рис. 22.2. *Древовидная структура, нарисованная средствами CSS*

22.2.1. Создание гистограмм средствами CSS

Гистограмма, представленная на рис. 22.1, была создана с помощью следующего HTML-файла:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
                      "http://www.w3.org/TR/html4/loose.dtd">
<!-- Без объявления DOCTYPE в IE рисунок будет выглядеть неправильно -->
<html>
<head>
<script src="BarChart.js"></script> <!-- Подключить библиотеку -->
<script>
function drawChart() {
    var chart = makeBarChart([1,2,4,8,16,32,64,128,256], 600, 300);
    var container = document.getElementById("chartContainer");
    container.appendChild(chart);
}
</script>
</head>
<body onload="drawChart( )">
<h2>y = 2<sup>n</sup></h2><!-- Заголовок гистограммы -->
<div id="chartContainer"><!-- Здесь рисуется гистограмма --></div>
<!-- Подпись под гистограммой -->
<i>Обратите внимание: каждый столбец в два раза выше предыдущего --
это характеристика экспоненциальной функции</i>
</body>
</html>
```

Совершенно очевидно, что все самое интересное сосредоточено в функции `makeBarChar()` **из файла *BarChart.js*, содержимое которого приводится в примере 22.5.**

Пример 22.5. Рисование гистограмм средствами CSS

```
/**
 * BarChart.js:
 * В этом файле содержится определение функции makeBarChart(), которая
 * создает гистограмму для вывода содержимого массива data[].
 * Общий размер гистограммы определяется необязательными аргументами
 * width и height, которые учитывают пространство, необходимое для рамок
 * гистограммы и внутренних отступов. Необязательный аргумент barcolor
 * определяет цвет столбиков. Функция возвращает созданный ею элемент
 * <div>, таким образом, вызывающий сценарий может манипулировать
 * этим элементом, например, изменять величину отступов. Вызывающий
 * сценарий должен вставить в документ полученный от функции элемент,
 * чтобы сделать его видимым.
 **/
function makeBarChart(data, width, height, barcolor) {
    // Предусмотреть значения по умолчанию для необязательных аргументов
    if (!width) width = 500;
    if (!height) height = 350;
    if (!barcolor) barcolor = "blue";

    // Аргументы width и height определяют общий размер гистограммы.
    // Чтобы получить размер создаваемого элемента, необходимо вычесть
    // из этих значений толщину рамок и величину отступов.
```

```
    width -= 24;  // Вычесть 10px отступа и 2px толщины рамки слева и справа
    height -= 14; // Вычесть 10px отступа сверху и 2px толщины рамки сверху и снизу

    // Создать элемент для размещения гистограммы. Обратите внимание:
    // гистограмма позиционируется в относительных координатах, т. е.
    // в ней могут располагаться дочерние элементы с абсолютным
    // позиционированием, и отображаться при этом в нормальном потоке
    // вывода элементов документа.
    var chart = document.createElement("div");
    chart.style.position = "relative";      // Относительное позиционирование
    chart.style.width = width + "px";       // Ширина гистограммы
    chart.style.height = height + "px";     // Высота гистограммы
    chart.style.border = "solid black 2px"; // Определить рамку
    chart.style.paddingLeft = "10px";       // Добавить отступ слева
    chart.style.paddingRight = "10px";      // Справа
    chart.style.paddingTop = "10px";        // Сверху
    chart.style.paddingBottom = "0px";      // Но не снизу
    chart.style.backgroundColor = "white";  // Фон гистограммы - белый

    // Рассчитать ширину каждого столбика
    var barwidth = Math.floor(width/data.length);
    // Отыскать наибольшее число в массиве data[]. Обратите внимание
    // на грамотное использование функции Function.apply().
    var maxdata = Math.max.apply(this, data);
    // Масштабирующий множитель: scale*data[i] дает высоту столбика
    var scale = height/maxdata;

    // Обойти в цикле массив с данными и создать столбики для всех элементов
    for(var i = 0; i < data.length; i++) {
        var bar = document.createElement("div"); // Создать столбик
        var barheight = data[i] * scale;         // Рассчитать высоту
        bar.style.position = "absolute";         // Уст. размер и положение
        bar.style.left = (barwidth*i+1+10)+"px"; // Добавить рамку столбика
                                                 // и отступ
        bar.style.top = height-barheight+10+"px";// Добавить отступ
                                                 // гистограммы
        bar.style.width = (barwidth-2) + "px";   // -2 - рамка столбика
        bar.style.height = (barheight-1) + "px"; // -1 - рамка сверху
        bar.style.border = "solid black 1px";    // Стиль рамки столбика
        bar.style.backgroundColor = barcolor;    // Цвет столбика
        bar.style.fontSize = "0px";              // Учесть особенность IE
        chart.appendChild(bar);                  // Добавить столбик
                                                 // в гистограмму
    }
    // В заключение вернуть элемент с гистограммой
    return chart;
}
```

Программный код примера 22.5 достаточно прямолинеен и в нем не сложно разобраться. Здесь демонстрируются приемы создания новых элементов `<div>` и добавления их в документ – об этих приемах рассказывалось в главе 15. Кроме того, здесь использованы приемы установки свойств CSS-стилей в создаваемых элементах, о которых рассказывалось в главе 16. В документе нет текстового содержимого, гистограмма представляет собой просто набор прямоугольников,

для каждого из которых аккуратно вычисляются размеры и координаты внутри другого прямоугольника. Видимыми прямоугольники делают CSS-атрибуты `border` и `background-color`. Один из важнейших фрагментов программного кода – код задания стиля `position:relative` без установки стилей `top` и `left` для самой гистограммы. Это позволяет гистограмме оставаться в нормальном потоке вывода документа, но иметь дочерние элементы с абсолютным позиционированием относительно левого верхнего угла гистограммы. Если бы для гистограммы не был задан стиль относительного (или абсолютного) позиционирования, ни один из столбиков не удалось бы корректно вывести.

В примере 22.5 можно найти несложные арифметические вычисления высоты столбиков гистограммы в пикселах на основе значений отображаемых данных. Программный код, который вычисляет координаты и размеры столбиков, также учитывает размеры рамок и отступов.

22.2.2. Класс CSSDrawing

Программный код, представленный в примере 22.5, призван решить единственную задачу – нарисовать гистограмму. Однако средствами CSS можно рисовать и более сложные диаграммы, например деревья, как показано на рис. 22.2, при условии, что они будут состоять из прямоугольников, а также горизонтальных и вертикальных линий.

В примере 22.6 приводится определение класса `CSSDrawing`, предоставляющего простой прикладной интерфейс для рисования прямоугольников и линий, а в примере 22.7 – код, который использует класс `CSSDrawing` для воссоздания диаграммы, представленной на рис. 22.2.

Пример 22.6. Класс CSSDrawing

```
/**
 * Данная функция-конструктор создает элемент div, в котором средствами CSS
 * может быть нарисована фигура. С помощью методов экземпляра можно рисовать
 * линии и прямоугольники и вставлять полученные фигуры в документ.
 *
 * При вызове конструктора можно использовать две различные сигнатуры:
 *
 *   new CSSDrawing(x, y, width, height, classname, id)
 *
 * В данном случае элемент <div> создается со стилем position:absolute
 * в указанных координатах и с заданными размерами.
 *
 * Конструктор может также вызываться только с аргументами width и height:
 *
 *   new CSSDrawing(width, height, classname, id)
 *
 * В этом случае элемент <div> создается с заданными шириной и высотой
 * и со стилем position:relative (это необходимо, чтобы дочерние элементы,
 * изображающие линии и прямоугольники, могли иметь абсолютное
 * позиционирование).
 *
 * В обоих случаях аргументы classname и id являются необязательными.
 * Если они определены, их значения используются в качестве
```

```
 * значений атрибутов class и id созданного элемента <div> и могут
 * использоваться, чтобы связать CSS-стили с фигурой.
 */
function CSSDrawing(/* переменное число аргументов */) {
    // Создать и запомнить элемент <div>, предназначенный для рисования
    var d = this.div = document.createElement("div");
    var next;

    // Выяснить количество числовых аргументов - четыре или два,
    // это размеры и координаты элемента div соответственно
    if (arguments.length >= 4 && typeof arguments[3] == "number") {
        d.style.position = "absolute";
        d.style.left = arguments[0] + "px";
        d.style.top = arguments[1] + "px";
        d.style.width = arguments[2] + "px";
        d.style.height = arguments[3] + "px";
        next = 4;
    }
    else {
        d.style.position = "relative"; // Это очень важно
        d.style.width = arguments[0] + "px";
        d.style.height = arguments[1] + "px";
        next = 2;
    }

    // Установить атрибуты class и id, если они были заданы.
    if (arguments[next]) d.className = arguments[next];
    if (arguments[next+1]) d.id = arguments[next+1];
}

/**
 * Добавляет к рисунку прямоугольник.
 *
 * x, y, w, h:    определяют координаты и размеры прямоугольника.
 * content:       строка текста или HTML-кода, который выводится в прямоугольнике
 * classname, id: необязательные значения атрибутов class и id для прямоугольника.
 *                Могут использоваться для связи прямоугольника со стилями,
 *                что позволяет определить цвет, рамку и пр.
 * Возвращаемое значение: Элемент <div>, изображающий прямоугольник
 */
CSSDrawing.prototype.box = function(x, y, w, h, content, classname, id) {
    var d = document.createElement("div");
    if (classname) d.className = classname;
    if (id) d.id = id;
    d.style.position = "absolute";
    d.style.left = x + "px";
    d.style.top = y + "px";
    d.style.width = w + "px";
    d.style.height = h + "px";
    d.innerHTML = content;
    this.div.appendChild(d);
    return d;
};

/**
```

```
 * Добавляет к рисунку горизонтальную линию.
 *
 * x, y, width:   определяют координаты начальной точки и толщину линии
 * classname, id: необязательные значения атрибутов class и id. По крайней
 *                мере, один должен присутствовать, чтобы определить стиль
 *                рамки, который будет использоваться для определения
 *                стиля линии, цвета и толщины.
 * Возвращаемое значение: Элемент <div>, изображающий линию
 */
CSSDrawing.prototype.horizontal = function(x, y, width, classname, id) {
    var d = document.createElement("div");
    if (classname) d.className = classname;
    if (id) d.id = id;
    d.style.position = "absolute";
    d.style.left = x + "px";
    d.style.top = y + "px";
    d.style.width = width + "px";
    d.style.height = 1 + "px";
    d.style.borderLeftWidth = d.style.borderRightWidth =
        d.style.borderBottomWidth = "0px";
    this.div.appendChild(d);
    return d;
};

/**
 * Добавляет к рисунку вертикальную линию.
 * Подробности в описании метода horizontal().
 */
CSSDrawing.prototype.vertical = function(x, y, height, classname, id) {
    var d = document.createElement("div");
    if (classname) d.className = classname;
    if (id) d.id = id;
    d.style.position = "absolute";
    d.style.left = x + "px";
    d.style.top = y + "px";
    d.style.width = 1 + "px";
    d.style.height = height + "px";
    d.style.borderRightWidth = d.style.borderBottomWidth =
        d.style.borderTopWidth = "0px";
    this.div.appendChild(d);
    return d;
};

/** Вставляет рисунок в документ как дочерний элемент указанного контейнера */
CSSDrawing.prototype.insert = function(container) {
    if (typeof container == "string")
        container = document.getElementById(container);
    container.appendChild(this.div);
}

/** Вставляет рисунок в документ, замещая указанный элемент */
CSSDrawing.prototype.replace = function(elt) {
    if (typeof elt == "string") elt = document.getElementById(elt);
    elt.parentNode.replaceChild(this.div, elt);
}
```

Конструктор `CSSDrawing()` создает новый объект `CSSDrawing`, который представляет собой всего лишь обертку элемента `<div>`. Методы экземпляра `box()`, `vertical()` и `horizontal()` рисуют средствами CSS прямоугольники, вертикальные линии и горизонтальные линии соответственно. Каждый метод позволяет определить координаты и размеры прямоугольника или линии, а также значения атрибутов `class` и `id` создаваемого элемента прямоугольника или линии. Атрибуты `class` или `id` могут использоваться для связи рисуемых элементов со стилями CSS, определяющими цвет, толщину линий и тому подобное. Чтобы сделать объект `CSSDrawing` видимым, недостаточно создать его. Нужно еще вставить его в документ с помощью метода `insert()` или `replace()`.

В примере 22.7 демонстрируется порядок использования класса `CSSDrawing`. Обе части примера – и JavaScript-код в методе `drawFigure()`, и таблица стилей CSS – играют важную роль в создании рисунка. В программном коде определяются координаты и размеры прямоугольников и линий, а в таблице стилей – цвет и толщина линий. Обратите внимание, насколько тесно связаны между собой сценарий JavaScript и таблицы стилей CSS: программный код метода `drawFigure()` должен учитывать толщину рамки и ширину отступов, заданных в таблице стилей. Это можно отнести к недостаткам прикладного интерфейса рисования, определяемого классом `CSSDrawing`.

Пример 22.7. Рисование диаграммы с помощью класса CSSDrawing

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
                      "http://www.w3.org/TR/html4/loose.dtd">
<!-- Без этого объявления DOCTYPE рисунок в IE получится неправильным -->
<html>
<head>
<script src="CSSDrawing.js"></script> <!-- Подключить определение класса -->
<style>
/* Стили для прямоугольника самого рисунка */
.figure { border: solid black 2px; background-color: #eee;}
/* Стили для линий сетки */
.grid { border: dotted black 1px; opacity: .1; }
/* Стили для прямоугольников на рисунке */
.boxstyle {
    border: solid black 2px;
    background: #aaa;
    padding: 2px 10px 2px 10px;
    font: bold 12pt sans-serif;
    text-align: center;
}

/* Стили для линий, соединяющих прямоугольники */
.boldline { border: solid black 2px; }
</style>
<script>
// Рисует сетку в заданном прямоугольнике с расстояниями между линиями dx,dy
function drawGrid(drawing, x, y, w, h, dx, dy) {
    for(var x0 = x; x0 < x +w; x0 += dx)
        drawing.vertical(x0, y, h, "grid");
    for(var y0 = y; y0 < y + h; y0 += dy)
        drawing.horizontal(x, y0, w, "grid");
```

```
}
function drawFigure( ) {
    // Создать новый рисунок
    var figure = new CSSDrawing(500, 200, "figure");

    // Вставить в рисунок сетку
    drawGrid(figure, 0, 0, 500, 200, 25, 25);

    // Нарисовать четыре прямоугольника
    figure.box(200, 50, 75, 25, "Life", "boxstyle"); // верхний прямоугольник
    figure.box(50, 125, 75, 25, "Archaea", "boxstyle"); // линейка из 3
    figure.box(200, 125, 75, 25, "Bacteria", "boxstyle"); // ..прямоугольников
    figure.box(350, 125, 75, 25, "Eukaryota", "boxstyle"); // ..ниже

    // Это линия, опускающаяся вниз от центра нижней границы прямоугольника
    // "Life". Начальная координата y этой линии: 50+25+2+2+2+2, или
    // y + height + top border + top padding + bottom padding + bottom border
    // Обратите внимание: для подобных вычислений необходимо знать как программный
    // код, так и таблицы стилей. Такой подход нельзя признать идеальным.
    figure.vertical(250, 83, 20, "boldline");

    figure.horizontal(100, 103, 300, "boldline"); // горизонтальная линия
    figure.vertical(100, 103, 22, "boldline");    // соединение с "archaea"
    figure.vertical(250, 103, 22, "boldline");    // соединение с "bacteria"
    figure.vertical(400, 103, 22, "boldline");    // соединение с "eukaryota"

    // Вставить рисунок в документ, заменив элемент-заполнитель
    figure.replace("placeholder");
}
</script>
</head>
<body onload="drawFigure()">
<div id="placeholder"></div>
</body>
</html>
```

22.3. SVG – масштабируемая векторная графика

Масштабируемая векторная графика (SVG) – это грамматика языка XML для описания графических изображений. Слово «векторная» в названии указывает на фундаментальное отличие от таких форматов растровой графики, как GIF, JPEG и PNG, где изображение задается матрицей пикселов. Формат SVG представляет собой точное, не зависящее от разрешения (отсюда слово «масштабируемая») описание шагов, которые необходимо выполнить, чтобы нарисовать требуемый рисунок. Вот пример простого SVG-изображения в текстовом формате:

```
<!-- Начало рисунка и объявление пространства имен -->
<svg xmlns="http://www.w3.org/2000/svg"
     viewBox="0 0 1000 1000"> <!-- Система координат рисунка -->
<defs>                        <!-- Настройка некоторых определений -->
  <linearGradient id="fade">  <!-- цветовой градиент с именем "fade" -->
    <stop offset="0%" stop-color="#008"/> <!-- Начинаем с темно-голубого -->
    <stop offset="100%" stop-color="#ccf"/><!-- Заканчиваем светло-голубым -->
  </linearGradient>
</defs>
```

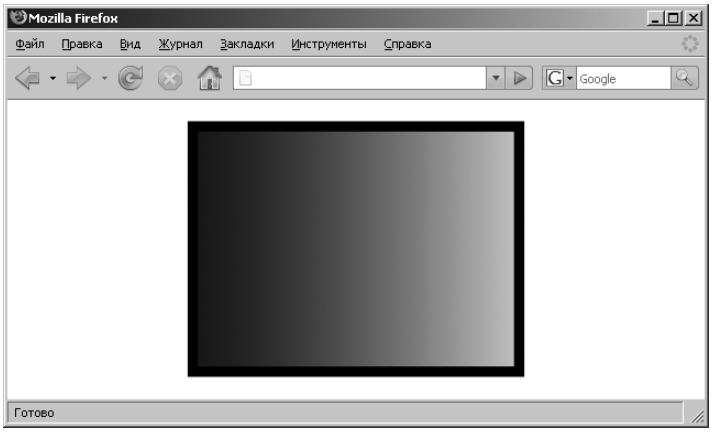

Рис. 22.3. *Простое изображение в формате SVG*

```
<!--
  Нарисовать прямоугольник с тонкой черной рамкой и заполнить его градиентом
-->
<rect x="100" y="200" width="800" height="600"
    stroke="black" stroke-width="25" fill="url(#fade)"/>
</svg>
```

На рис. 22.3 показано графическое представление этого SVG-файла.

SVG – это довольно обширная грамматика умеренной сложности. Помимо простых примитивов рисования она позволяет воспроизводить произвольные кривые, текст и анимацию. Рисунки в формате SVG могут даже содержать JavaScript-сценарии и таблицы CSS-стилей, что позволяет наделить их информацией о поведении и представлении. В этом разделе показано, как с помощью клиентского JavaScript-кода (встроенного в HTML-, а не в SVG-документ) можно динамически создавать графические изображения средствами SVG. Приводимые здесь примеры SVG-изображений позволяют лишь отчасти оценить возможности формата SVG. Полное описание этого формата доступно в виде общирной, но вполне понятной спецификации, которая поддерживается консорциумом W3C и находится по адресу *http://www.w3.org/TR/SVG/*. Обратите внимание: эта спецификация включает в себя полное описание объектной модели документа (DOM) для SVG-документов. В то же время в данном разделе рассматриваются приемы манипулирования SVG-графикой с помощью стандартной модели XML DOM, а модель SVG DOM вообще не упоминается.

К моменту написания этих строк из ведущих веб-броузеров лишь Firefox 1.5 имел встроенную поддержку формата SVG. Чтобы в этом броузере просматривать SVG-графику, достаточно просто ввести URL-адрес требуемого изображения. SVG-графику очень удобно встраивать прямо в XHTML-файлы, как это показано в следующем примере:

```
<?xml version="1.0"?>
<!--
    Объявить пространство имен HTML по умолчанию, а SVG - с префиксом "svg:"
-->
```

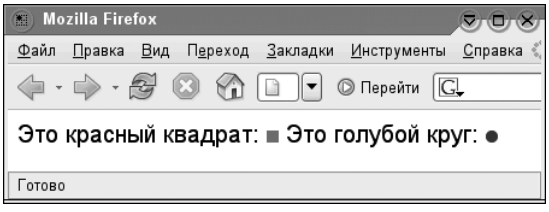

Рис. 22.4. *SVG-графика в XHTML-документе*

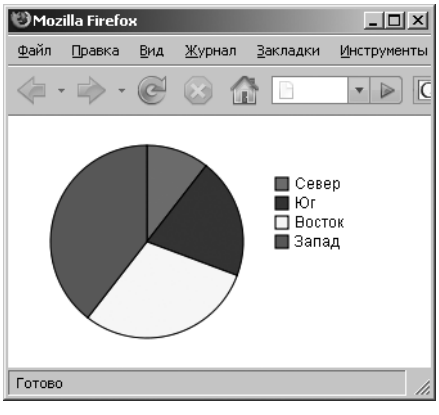

Рис. 22.5. *Круговая диаграмма в формате SVG, построенная JavaScript-сценарием*

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:svg="http://www.w3.org/2000/svg">
<body>
Это красный квадрат: <!-- HTML-текст -->
<svg:svg width="20" height="20"> <!-- SVG-изображение -->
<svg:rect x="0" y="0" width="20" height="10" fill="red"/></svg:svg>
Это голубой круг:
<svg:svg width="20" height="20">
<svg:circle cx="10" cy="10" r="10" fill="blue"/></svg:svg>
</body>
</html>
```

На рис. 22.4 показано, как Firefox 1.5 отображает этот XHTML-документ.

Изображения в формате SVG могут также встраиваться в HTML-документы внутри тега `<object>`, что дает возможность отображать их с помощью модулей расширения. Компания Adobe свободно распространяет (без открытых исходных текстов) модуль просмотра SVG-графики для работы в составе наиболее распространенных броузеров и операционных систем. Найти его можно, следуя по ссылкам, начиная с адреса *http://www.adobe.com/svg*.

Так как формат SVG – это грамматика языка XML, рисование SVG-изображений заключается лишь в использовании модели DOM для создания соответствующих XML-элементов. В примере 22.8 приводится код функции `pieChart()`, которая создает SVG-элементы для воспроизведения круговой диаграммы, подобной показанной на рис. 22.5. (Другие технологии создания векторной графики, описы-

ваемые в этой главе, также могут использоваться для создания аналогичных круговых диаграмм.)

Пример 22.8. Рисование круговой диаграммы средствами JavaScript и SVG

```
/**
 * Рисует круговую диаграмму внутри элемента <svg>.
 * Аргументы:
 *   canvas: SVG-элемент (или id этого элемента) для рисования
 *   data: массив значений для диаграммы, по одному для каждого сектора
 *   cx, cy, r: координаты центра и радиус круга
 *   colors: массив HTML-строк цветов, по одному на каждый сектор
 *   labels: массив меток легенды, по одной на каждый сектор
 *   lx, ly: координаты верхнего левого угла легенды диаграммы
 */
function pieChart(canvas, data, cx, cy, r, colors, labels, lx, ly) {
    // Отыскать "холст", если он задан значением id
    if (typeof canvas == "string") canvas = document.getElementById(canvas);

    // Сложить вместе все значения, чтобы получить общую сумму всей диаграммы
    var total = 0;
    for(var i = 0; i < data.length; i++) total += data[i];

    // Определить величину каждого сектора. Углы измеряются в радианах.
    var angles = []
    for(var i = 0; i < data.length; i++) angles[i] = data[i]/total*Math.PI*2;

    // Цикл по всем секторам диаграммы.
    startangle = 0;
    for(var i = 0; i < data.length; i++) {
        // Точка, где заканчивается сектор
        var endangle = startangle + angles[i];

        // Вычислить координаты точек пересечения образующих
        // сектор радиусов с окружностью.
        // В соответствии с выбранными формулами углу 0 радиан соответствует
        // точка в самой верхней части окружности, а положительные значения
        // откладываются от нее по часовой стрелке.
        var x1 = cx + r * Math.sin(startangle);
        var y1 = cy - r * Math.cos(startangle);
        var x2 = cx + r * Math.sin(endangle);
        var y2 = cy - r * Math.cos(endangle);

        // Это флаг для углов, больших половины окружности
        var big = 0;
        if (endangle - startangle > Math.PI) big = 1;

        // Мы описываем сектор с помощью элемента <svg:path>
        // Примечательно, что он создается вызовом createElementNS()
        var path = document.createElementNS(SVG.ns, "path");

        // Эта строка хранит информацию о пути пера, рисующего сектор
        var d = "M " + cx + "," + cy + // Начало в центре окружности
            " L " + x1 + "," + y1 +    // Нарисовать линию к точке (x1,y1)
            " A " + r + "," + r +      // Нарисовать дугу с радиусом r
            " 0 " + big + " 1 " +      // Информация о дуге...
            x2 + "," + y2 +            // Дуга заканчивается в точке (x2,y2)
```

```
            " Z";                       // Закончить рисование в точке (cx,cy)

        // Это XML-элемент, поэтому значения всех атрибутов должны
        // устанавливаться с помощью setAttribute(). Здесь нельзя
        // использовать свойства JavaScript
        path.setAttribute("d", d);              // Установить этот путь
        path.setAttribute("fill", colors[i]);   // Установить цвет сектора
        path.setAttribute("stroke", "black");   // Рамка сектора - черная
        path.setAttribute("stroke-width", "2"); // 2 единицы толщиной
        canvas.appendChild(path);               // Вставить сектор в "холст"

        // Следующий сектор начинается в точке, где закончился предыдущий
        startangle = endangle;

        // Нарисовать маленький квадрат для идентификации сектора в легенде
        var icon = document.createElementNS(SVG.ns, "rect");
        icon.setAttribute("x", lx);           // Координаты квадрата
        icon.setAttribute("y", ly + 30*i);
        icon.setAttribute("width", 20);       // Размер квадрата
        icon.setAttribute("height", 20);
        icon.setAttribute("fill", colors[i]); // Тем же цветом, что и сектор
        icon.setAttribute("stroke", "black"); // Такая же рамка
        icon.setAttribute("stroke-width", "2");
        canvas.appendChild(icon);             // Добавить в "холст"

        // Добавить метку правее квадрата
        var label = document.createElementNS(SVG.ns, "text");
        label.setAttribute("x", lx + 30);     // Координаты текста
        label.setAttribute("y", ly + 30*i + 18);
        // Стиль текста можно было определить посредством таблицы CSS-стилей
        label.setAttribute("font-family", "sans-serif");
        label.setAttribute("font-size", "16");
        // Добавить текстовый DOM-узел в элемент <svg:text>
        label.appendChild(document.createTextNode(labels[i]));
        canvas.appendChild(label);            // Добавить текст в "холст"
    }
};
```

Программный код из примера 22.8 достаточно прост. Здесь выполняются некоторые математические расчеты для преобразования исходных данных в углы секторов круговой диаграммы. Однако основную часть примера составляет программный код, создающий SVG-элементы и выполняющий настройку атрибутов этих элементов. Обратите внимание: поскольку формат SVG задействует пространства имен, вместо метода `createElement()` используется метод `createElementNS()`. Константа `SVG.ns` с именем пространства имен определена в примере 22.9.

Самая малопонятная часть этого примера – код, выполняющий рисование сектора диаграммы. Для отображения каждого сектора используется тег `<svg:path>`. Этот SVG-элемент описывает рисование произвольных фигур, состоящих из линий и кривых. Описание фигуры вставляется в тег `<svg:path>` в виде значения атрибута `d`. Основу описания составляет компактная грамматика символьных кодов и чисел, определяющих координаты, углы и прочие значения. Например, символ M означает «move to» (переместиться в точку), и вслед за ним должны следовать координаты X и Y точки. Символ L означает «line to» (рисовать линию до точки); он рисует линию от текущей точки до точки с координатами, которые

следуют далее. Кроме того, в этом примере используется символьный код A, который рисует дугу (arc). Вслед за этим символом следуют семь чисел, описывающих дугу. Точное описание нас здесь не интересует, но вы можете найти его в спецификации, по адресу *http://www.w3.org/TR/SVG/*.

В примере 22.8 использована константа `SVG.ns`, которая описывает пространство имен SVG. Эта константа и ряд вспомогательных функций определены в виде отдельного файла *SVG.js*, содержимое которого приводится в примере 22.9.

Пример 22.9. Вспомогательный программный SVG-код

```
// Создать пространство имен для вспомогательных функций
var SVG = {};

// Эти URL-адреса определяют пространства имен, связанные с SVG
SVG.ns = "http://www.w3.org/2000/svg";
SVG.xlinkns = "http://www.w3.org/1999/xlink";

// Создает и возвращает пустой элемент <svg>.
// Обратите внимание: элемент не добавляется в документ.
// Кроме того, можно определить размеры изображения в пикселах,
// а также его внутреннюю систему координат.
SVG.makeCanvas = function(id, pixelWidth, pixelHeight, userWidth, userHeight) {
    var svg = document.createElementNS(SVG.ns, "svg:svg");
    svg.setAttribute("id", id);
    // Размер "холста" в пикселах
    svg.setAttribute("width", pixelWidth);
    svg.setAttribute("height", pixelHeight);
    // Установить координаты, которые будут использоваться при рисовании
    svg.setAttribute("viewBox", "0 0 " + userWidth + " " + userHeight);
    // Определить пространство имен XLink, которое используется SVG
    svg.setAttributeNS("http://www.w3.org/2000/xmlns/", "xmlns:xlink",
                       SVG.xlinkns);
    return svg;
};

// Сериализовать элемент "холста" в строку и использовать эту строку
// в спецификаторе data: URL-адреса для отображения тега <object>.
// Это позволит SVG работать в броузерах, которые поддерживают URL-адреса
// со спецификатором data: и в которых установлен модуль SVG.
SVG.makeDataURL = function(canvas) {
    // Мы не будем беспокоиться о вопросах сериализации в IE, поскольку
    // этот броузер не поддерживает URL-адреса со спецификатором data:
    var text = (new XMLSerializer()).serializeToString(canvas);
    var encodedText = encodeURIComponent(text);
    return "data:image/svg+xml," + encodedText;
};

// Создает тег <object> для вывода SVG-рисунка с помощью
// URL-адреса со спецификатором data:
SVG.makeObjectTag = function(canvas, width, height) {
    var object = document.createElement("object"); // Создать тег <object>
    object.width = width;                          // Установить размеры
    object.height = height;
    object.data = SVG.makeDataURL(canvas);         // SVG-изображение как URL-адрес
                                                   // со спецификатором data:
```

```
    object.type = "image/svg+xml"                    // MIME-тип для SVG
    return object;
}
```

Наиболее важной в этом примере является функция `SVG.makeCanvas()`. Она с помощью DOM-методов создает элемент `<svg>`, который затем используется в качестве «холста» для рисования SVG-графики. Функция `makeCanvas()` позволяет определить размеры выводимого SVG-изображения (в пикселах), а также размеры внутренней системы координат (или «пользовательское пространство»), которая потребуется в процессе рисования. (Например, когда пользовательское пространство с размерами 1000×1000 отображается в квадрате 250×250, каждому элементу пользовательского пространства соответствует одна четвертая пиксела.) Функция `createCanvas()` создает и возвращает тег `<svg>`, но не вставляет его в документ. Сделать это должен вызывающий программный код.

Другие две вспомогательные функции из примера 22.9 используются для вывода SVG-графики с помощью модулей расширения в броузерах. Функция `SVG.makeDataURL()` сериализует XML-текст тега `<svg>` и преобразует его в URL-адрес со спецификатором `data:`. Функция `SVG.makeObjectTag()` идет еще дальше — она создает тег `<object>` для встраивания SVG-графики, а затем вызывает функцию `SVG.makeDataURL()`, возвращаемое значение которой записывается в атрибут `data` этого тега. Подобно функции `SVG.makeCanvas()`, метод `SVG.makeObjectTag()` возвращает тег `<object>`, и точно так же не вставляет его в документ.

Чтобы метод `SVG.makeObjectTag()` мог работать, броузер должен поддерживать URL-адреса со спецификатором `data:` (как Firefox 1.0) и DOM-методы, распознающие пространства имен (как `document.createElementNS()`), а также иметь установленный модуль расширения для просмотра SVG-графики. Обратите внимание: эти методы не будут работать в IE, потому что IE не поддерживает ни URL-адреса со спецификатором `data:`, ни метод `createElementNS()`. Для создания SVG-изображения в IE вместо обращений к DOM-методам можно использовать методы манипулирования строками, чтобы собрать SVG-документ. После этого графика может быть преобразована в URL-адрес со спецификатором `javascript:` вместо `data:`.

Я закончу этот раздел содержимым HTML-файла, который объединяет функцию `pieChart()` из примера 22.8 и вспомогательные SVG-методы из примера 22.9. Следующий фрагмент создает «холст» SVG, рисует на нем диаграмму и затем дважды вставляет «холст» в документ — один раз непосредственно и второй раз в виде тега `<object>`:

```
<script src="SVG.js"></script> <!-- Вспомогательные методы -->
<script src="svgpiechart.js"></script> <!-- Методы рисования диаграммы -->
<script>
function init() {
    // Создать тег <svg> для рисования с разрешением 600x400 и выводов в 300x200 пикселов
    var canvas = SVG.makeCanvas("canvas", 300, 200, 600, 400);
    pieChart(canvas, [12, 23, 34, 45], 200, 200, 150, // Холст, данные, размер
                     ["red", "blue", "yellow", "green"], // Цвет секторов
                     ["Север","Юг","Восток","Запад"], 400, 100); // Легенда

    // Добавить рисунок прямо в документ:
    document.body.appendChild(canvas);
```

```
    // Встроить в тег <object>
    var object = SVG.makeObjectTag(canvas, 300, 200);
    document.body.appendChild(object);
}
// Запустить эту функцию, когда документ будет загружен полностью
window.onload = init;
</script>
```

22.4. VML – векторный язык разметки

Формат VML – это ответ Microsoft на появление SVG. Подобно SVG, VML также является грамматикой языка XML, предназначенной для описания графических изображений. VML во многом напоминает SVG. Несмотря на то, что возможности формата VML не столь широки, как SVG, он предлагает полный набор примитивов рисования и имеет встроенную поддержку в IE начиная с версии 5.5. Компания Microsoft (и некоторые ее партнеры) передали формат VML консорциуму W3C для рассмотрения в качестве стандарта, но их усилия пока ни к чему не привели. Лучшее из существующих описание VML, представленное компанией Microsoft, доступно на веб-сайте W3C по адресу *http://www.w3.org/TR/NOTE-VML*. Обратите внимание: несмотря на то, что этот документ размещен на сайте W3C, формат VML еще не стандартизован, а его реализация является собственностью компании Microsoft.

Хотя VML – это очень мощная технология, ей так и не удалось завоевать популярность. Из-за не слишком широкого распространения[1] она не подвергалась тщательному документированию. Веб-сайты Microsoft обычно указывают на спецификации, переданные консорциуму W3C, как на авторитетный источник сведений. К сожалению, т. к. этот документ – лишь проект, он никогда не подвергался тщательному изучению с целью стандартизации и потому в отдельных местах страдает неполнотой, в других – неточностями. При работе с VML вам, скорее всего, придется идти путем проб и ошибок, экспериментируя с реализацией в IE в процессе создания требуемых изображений. Это предупреждение можно считать незначительным, если рассматривать VML как мощный механизм создания векторной графики на стороне клиента, тем более учитывая, что данный механизм встроен в веб-броузер, который по-прежнему доминирует на рынке.

VML – это диалект XML, отличающийся от HTML, но IE не обладает полноценной поддержкой XHTML-документов, а его реализация модели DOM не поддерживает функции, различающие пространства имен, такие как `document.createElementNS()`. Работа с тегами в пространстве имен VML в IE обеспечивается с помощью HTML-атрибутов «поведения» (еще одно расширение, характерное для IE). Все HTML-файлы, содержащие VML-документы, в первую очередь должны объявить пространство имен, например так:

```
<html xmlns:v="urn:schemas-microsoft-com:vml">
```

Это же пространство имен может быть объявлено иначе, специфичным только для IE (нестандартным) способом:

---

[1] Насколько мне известно, Google Maps (*http://local.google.com*) – единственный высокопрофессиональный сайт, в котором используется технология VML.

```
<xml:namespace ns="urn:schemas-microsoft-com:vml" prefix="v"/>
```

Затем с помощью следующего нестандартного CSS-объявления необходимо указать, как должны обрабатываться теги из этого пространства имен:

```
<style>v\:* { behavior: url(#default#VML); }</style>
```

После того как все необходимые объявления сделаны, можно свободно перемешивать VML- и HTML-код, как в следующем фрагменте, который создает нечто похожее на SVG-изображение, представленное на рис. 22.4:

```
<html xmlns:v="urn:schemas-microsoft-com:vml">
<head><style>v\:* { behavior: url(#default#VML); }</style></head>
<body>
Это красный квадрат:<v:rect style="width:10px;height:10px;" fillcolor="red"/>
Это голубой круг:<v:oval style="width:10px;height:10px;" fillcolor="blue"/>
</body>
</html>
```

Однако вернемся к теме этой главы, в которой основной упор делается на использовании JavaScript-сценариев для динамического создания графических изображений на стороне клиента. В примере 22.10 демонстрируется, как построить круговую диаграмму средствами VML. Пример очень напоминает код построения круговой диаграммы средствами SVG, в нем просто SVG-примитивы рисования заменены VML-примитивами. Для простоты в этом примере объединены функции `makeVMLCanvas()`, `pieChart()` и программный код, который вызывает эти функции для вывода диаграммы. Результат работы это сценария здесь не показан, поскольку он ничем не отличается от диаграммы, полученной средствами SVG и представленной на рис. 22.5.

Пример 22.10. Рисование круговой диаграммы средствами JavaScript и VML

```
<!--
  HTML-документ, использующий VML, должен объявить это пространство имен
-->
<html xmlns:v="urn:schemas-microsoft-com:vml">
<head>
<!--
  Так связывается VML-поведение с пространством имен VML
-->
<style>v\:* { behavior: url(#default#VML); }</style>
<script>
/*
 * Создает и возвращает VML-элемент <v:group>, в котором будет размещен рисунок.
 * Обратите внимание: возвращаемый элемент не добавляется в документ.
 */
function makeVMLCanvas(id, pixelWidth, pixelHeight) {
    var vml = document.createElement("v:group");
    vml.setAttribute("id", id);
    vml.style.width = pixelWidth + "px";
    vml.style.height = pixelHeight + "px";
    vml.setAttribute("coordsize", pixelWidth + " " + pixelHeight);

    // Для начала нарисовать белый прямоугольник с черной рамкой.
    var rect = document.createElement("v:rect");
```

```
    rect.style.width = pixelWidth + "px";
    rect.style.height = pixelHeight + "px";
    vml.appendChild(rect);

    return vml;
}

/* Рисует диаграмму на "холсте" VML */
function pieChart(canvas, data, cx, cy, r, colors, labels, lx, ly) {
    // Отыскать элемент canvas, если он задан значением атрибута id
    if (typeof canvas == "string") canvas = document.getElementById(canvas);

    // Получить сумму значений всех данных
    var total = 0;
    for(var i = 0; i < data.length; i++) total += data[i];

    // Рассчитать размер каждого сектора (в градусах)
    var angles = []
    for(var i = 0; i < data.length; i++) angles[i] = data[i]/total*360;

    // Цикл по всем секторам.
    // Углы в VML измеряются в градусах/65535, отсчет начинается от крайней правой
    // точки окружности (3 часа) и продолжается против часовой стрелки
    startangle = 90; // Начало в 12часов.
    for(var i = 0; i < data.length; i++) {
        // Подправить углы так, чтобы отсчет секторов начинался от крайней
        // верхней точки окружности и продолжался по часовой стрелке.
        var sa = Math.round(startangle * 65535);
        var a = -Math.round(angles[i] * 65536);

        // Создать VML-элемент shape
        var wedge = document.createElement("v:shape");
        // В VML путь пера при рисовании фигуры описывается похожим
        // на SVG образом
        var path = "M " + cx + " " + cy +    // Перейти в точку (cx,cy)
            " AE " + cx + " " + cy + " " +   // Дуга с центром в (cx,cy)
            r + " " + r + " " +              // Горизонтальный и вертикальный радиусы
            sa + " " + a +                   // Начальный угол и общий угол
            " X E";                          // Закончить линию в центре окружн.
        wedge.setAttribute("path", path); // Установить фигуру сектора
        wedge.setAttribute("fillcolor", colors[i]); // Установить цвет
        wedge.setAttribute("strokeweight", "2px");  // Рамка

        // Позиционировать сектор с помощью CSS-стилей. Координаты точек
        // пути интерпретируются относительно размеров, поэтому каждой
        // фигуре мы задаем размеры всего "холста".
        wedge.style.position = "absolute";
        wedge.style.width = canvas.style.width;
        wedge.style.height = canvas.style.height;

        // Добавить фигуру в элемент canvas
        canvas.appendChild(wedge);

        // Следующий сектор начинается там, где закончился предыдущий
        startangle -= angles[i];

        // Создать VML-элемент <rect> для легенды
        var icon = document.createElement("v:rect");
```

```
        icon.style.left = lx + "px"; // CSS-позиционирование
        icon.style.top = (ly+i*30) + "px";
        icon.style.width = "20px";   // CSS-размеры
        icon.style.height = "20px";
        icon.setAttribute("fillcolor", colors[i]); // Цвет сектора
        icon.setAttribute("stroke", "black");      // Цвет рамки
        icon.setAttribute("strokeweight", "2");    // Толщина рамки
        canvas.appendChild(icon);                  // Добавить в "холст"

        // VML обладает широкими возможностями для работы с текстом,
        // но большая часть текста - это просто HTML-код, который непосредственно
        // добавляется в VML-рисунок с использованием координат рисунка
        var label = document.createElement("div"); // <div> для текста
        label.appendChild(document.createTextNode(labels[i])); // Текст
        label.style.position = "absolute"; // CSS-позиционирование
        label.style.left = (lx + 30) + "px";
        label.style.top = (ly + 30*i + 5) + "px";
        label.style.fontFamily = "sans-serif"; // Стили текста
        label.style.fontSize = "16px";
        canvas.appendChild(label); // Добавить текст в рисунок
    }
}

function init() {
    var canvas = makeVMLCanvas("canvas", 600, 400);
    document.body.appendChild(canvas);
    pieChart(canvas, [12, 23, 34, 45], 200, 200, 150,
             ["red", "blue", "yellow", "green"],
             ["Север", "Юг", "Восток", "Запад"],
             400, 100);
}
</script>
</head>
<body onload="init()">
</body>
</html>
```

22.5. Создание графики с помощью тега <canvas>

Следующая остановка в нашем путешествии по технологиям создания векторной графики на стороне клиента – тег <canvas>. Этот нестандартный HTML-тег специально предназначен для создания векторной графики на стороне клиента. Сам он не имеет визуального представления, но предоставляет JavaScript-сценариям интерфейс для создания рисунков внутри тега <canvas>. Впервые тег <canvas> был введен компанией Apple в веб-броузере Safari 1.3. (Причина такого радикального расширения HTML кроется в том, что HTML-средства визуализации Safari использовались также в компоненте Dashboard (инструментальная панель) рабочего стола Mac OS X, и компании Apple требовался механизм управления графикой в Dashboard.)

Броузеры Firefox 1.5 и Opera 9 последовали за Safari – оба также поддерживают тег <canvas>. Существует даже возможность использовать тег <canvas> в IE совместно со свободно распространяемым JavaScript-кодом (изначально разработан-

ным в Google), который обеспечивает работу тега <canvas> поверх VML (*http://excanvas.sourceforge.net*). Неофициальный консорциум производителей веб-броузеров продолжает прикладывать усилия по стандартизации тега <canvas>, предварительные спецификации можно найти по адресу *http://www.whatwg.org/specs/web-apps/current-work*.

Существенное отличие между тегом <canvas> и технологиями SVG и VML заключается в том, что тег <canvas> предоставляет прикладной интерфейс Canvas на базе JavaScript, предназначенный для создания изображений, в то время как SVG и VML описывают изображение в виде XML-документов. Функционально эти два подхода эквивалентны: любой из них может моделироваться с использованием другого. Однако внешне они совершенно отличаются, и каждый из них имеет свои сильные и слабые стороны. Например, из SVG-рисунков легко можно удалять элементы. Чтобы удалить элемент из аналогичного рисунка, созданного в теге <canvas>, обычно требуется полностью ликвидировать рисунок, а затем создать его заново. Поскольку прикладной интерфейс Canvas основан на синтаксисе JavaScript, а рисунки, созданные с его помощью, получаются более компактными (чем SVG- и VML-рисунки), я решил описать его в этой книге. Подробные сведения вы найдете в соответствующих разделах четвертой части книги.

Большая часть прикладного интерфейса Canvas определена не в элементе <canvas>, а в объекте «контекста рисования», получить который можно методом getContext() элемента, играющего роль «холста».[1] Этот сценарий рисует маленький красный квадрат и голубой круг, что является типичным для рисования в теге <canvas>.

```
<head>
<script>
window.onload = function() { // Создает рисунок после загрузки документа
    var canvas = document.getElementById("square"); // Получить элемент холста
    var context = canvas.getContext("2d");          // Получить 2D-контекст
    context.fillStyle = "#ff0000";                  // Цвет заливки - красный
    context.fillRect(0,0,10,10);                    // Залить квадрат

    canvas = document.getElementById("circle");     // Новый элемент холста
    context = canvas.getContext("2d");              // Получить его контекст
    context.fillStyle = "#0000ff";                  // Цвет заливки - голубой
    context.beginPath();                            // Начать рисование
    // Добавить в рисунок полную окружность с радиусом 5 и с центром в точке (5,5)
    context.arc(5, 5, 5, 0, 2*Math.PI, true);
    context.fill( );                                // Залить фигуру
}
</script>
</head>
<body>
Это красный квадрат: <canvas id="square" width=10 height=10></canvas>.
```

[1] Этот метод требует единственный аргумент – строку "2d" – и возвращает контекст рисования, который реализует прикладной интерфейс для создания двухмерных изображений. В будущем если тег <canvas> будет расширен для создания трехмерных изображений, этот метод, по всей видимости, будет получать в виде аргумента строку "3d".

```
Это голубой круг: <canvas id="circle" width=10 height=10></canvas>.
</body>
```

В предыдущих разделах вы видели, что при использовании SVG и VML сложные фигуры описываются как «путь» пера, состоящий из линий и кривых, которые могут быть нарисованы или залиты цветом. В прикладном интерфейсе `Canvas` также используется нотация пути перемещения пера, только путь описывается не как строка символов и чисел, а как последовательность вызовов методов, таких как `beginPath()` и `arc()` в данном примере. После того как описание пути завершается, вызывается другой метод, такой как `fill()`, который обрабатывет этот путь. Порядок обработки определяется различными свойствами объекта контекста рисунка, такими как `fillStyle`.

Одна из причин, объясняющих такую компактность прикладного интерфейса `Canvas`, заключается в том, что он никак не поддерживает работу с текстом. Чтобы вставить текст в графическое изображение тега `<canvas>`, вам придется вручную вставлять текст в рисунок в виде растрового изображения либо накладывать текст в формате HTML поверх тега `<canvas>`, используя возможности позиционирования CSS.

В примере 22.11 демонстрируется, как нарисовать круговую диаграмму в теге `<canvas>`. Большая часть этого кода покажется знакомой по примерам использования SVG и VML. Новый программный код в этом примере – это методы прикладного интерфейса `Canvas`, описание которых вы найдете в четвертой части книги.

Пример 22.11. Рисование круговой диаграммы в теге <canvas>

```
<html>
<head>
<script>
// Создает и возвращает новый тег <canvas> с заданным id и размерами.
// Обратите внимание: этот метод не добавляет тег <canvas> в документ
function makeCanvas(id, width, height) {
    var canvas = document.createElement("canvas");
    canvas.id = id;
    canvas.width = width;
    canvas.height = height;
    return canvas;
}

/**
 * Рисует круговую диаграмму в указанном теге <canvas>, который передается
 * либо в виде ссылки на элемент, либо в виде id.
 * Аргумент data - это массив чисел: каждое число представляет
 * отдельный сектор в диаграмме.
 * Центр диаграммы определяется значениями cx и cy, а радиус - r.
 * Цвета секторов - это HTML-строки цветов в массиве colors[].
 * Легенда размещается в координатах (lx,ly), метки легенды - в массиве labels[].
 */
function pieChart(canvas, data, cx, cy, r, colors, labels, lx, ly) {
    // Получить объект canvas по id
    if (typeof canvas == "string") canvas = document.getElementById(canvas);

    // Рисование производится с использованием объекта контекста
    var g = canvas.getContext("2d");
```

```
    // Все линии будут иметь черный цвет и толщину 2 пиксела
    g.lineWidth = 2;
    g.strokeStyle = "black";

    // Сумма всех значений
    var total = 0;
    for(var i = 0; i < data.length; i++) total += data[i];

    // Рассчитать угловые размеры каждого сектора (в радианах)
    var angles = []
    for(var i = 0; i < data.length; i++) angles[i] = data[i]/total*Math.PI*2;

    // Цикл по всем секторам диаграммы
    startangle = -Math.PI/2; // Начать с крайней верхней, а не с крайней
                             // правой точки окружности
    for(var i = 0; i < data.length; i++) {
        // Это угол конца сектора
        var endangle = startangle + angles[i];

        // Нарисовать сектор
        g.beginPath();           // Начать новую фигуру
        g.moveTo(cx,cy);         // Переместиться в центр
        // Нарисовать линию до точки startangle и дугу до точки endangle
        g.arc(cx,cy,r,startangle, endangle, false);
        g.closePath();          // Вернуться в центр фигуры и закончить рисование фигуры
        g.fillStyle = colors[i]; // Определить цвет заливки
        g.fill();                // Залить сектор
        g.stroke();              // Рамка сектора (штриховая)

        // Следующий сектор начинается там, где заканчивается предыдущий.
        startangle = endangle;

        // Нарисовать прямоугольник в легенде
        g.fillRect(lx, ly+30*i, 20, 20);
        g.strokeRect(lx, ly+30*i, 20, 20);

        // И вставить метку правее прямоугольника. Прикладной интерфейс Canvas
        // не поддерживает работу с текстом, поэтому здесь просто добавляется
        // обычный HTML-текст. Чтобы разместить текст правее прямоугольника
        // поверх элемента canvas, используются возможности позиционирования CSS.
        // Это было бы понятнее, если бы сам тег <canvas> позиционировался абсолютно
        var label = document.createElement("div");
        label.style.position = "absolute";
        label.style.left = (canvas.offsetLeft + lx+30)+"px";
        label.style.top = (canvas.offsetTop+ly+30*i-4) + "px";
        label.style.fontFamily = "sans-serif";
        label.style.fontSize = "16px";
        label.appendChild(document.createTextNode(labels[i]));
        document.body.appendChild(label);
    }
}

function init() {
    // Создать элемент canvas
    var canvas = makeCanvas("canvas", 600, 400);

    // Добавить в документ
    document.body.appendChild(canvas);
```

```
    // И нарисовать в нем круговую диаграмму
    pieChart("canvas", [12, 23, 34, 45], 200, 200, 150,
             ["red", "blue", "yellow", "green"],
             ["Север", "Юг", "Восток", "Запад"],
             400, 100);
}
</script>
</head>
<body onload="init()"></body>
</html>
```

22.6. Создание графики средствами Flash

Каждая из обсуждавшихся до сих пор в этой главе технологий создания векторной графики имеет ограниченный круг применения: тег `<canvas>` доступен только в броузерах Safari 1.3, Firefox 1.5 и Opera 9; технология VML может (и всегда будет) применяться только в IE, а встроенной поддержкой SVG обладает только броузер Firefox 1.5. Конечно, существуют модули поддержки SVG и для других броузеров, но эти модули пока не получили широкого распространения.

Один из мощнейших модулей векторной графики, который у многих (практически у всех) *установлен*, – это Flash-плеер компании Adobe (прежде – Macromedia). Flash-плеер имеет собственный язык сценариев, который называется ActionScript (фактически – диалект JavaScript). Начиная с версии 6 Flash-плеер предоставляет простой, но мощный прикладной интерфейс создания изображений в виде ActionScript-кода. Кроме того, Flash версий 6 и 7 предоставляет ограниченные средства взаимодействия между клиентским JavaScript-кодом и ActionScript-кодом, что дает возможность из JavaScript-сценариев посылать подключаемому Flash-модулю команды рисования, которые исполняются интерпретатором ActionScript.

Материал этой главы ориентирован на Flash 8, к моменту написания этих строк данная версия была самой новой. Flash 8 включает прикладной интерфейс `ExternalInterface`, который существенно упрощает экспорт ActionScript-методов, так что они могут совершенно прозрачно вызываться из JavaScript-сценариев. В главе 23 продемонстрировано, как вызывать методы рисования Flash 6 и 7.

Для рисования средствами Flash необходим файл с расширением *.swf*, который сам по себе не выполняет рисование, но экспортирует в JavaScript-сценарий прикладной интерфейс для работы с графикой.[1] Мы начнем рассмотрение с ActionScript-файла, содержимое которого приводится в примере 22.12.

Пример 22.12. Canvas.as

```
import flash.external.ExternalInterface;

class Canvas {
    // Свободно распространяемый компилятор mtasc автоматически вставит
    // вызов метода main() в скомпилированный SWF-файл. Если для создания
```

[1] Программный код примера 22.13, создающий круговую диаграмму, использует этот прикладной интерфейс рисования, но я не буду описывать его здесь. Всю необходимую документацию можно найти на веб-сайте компании Adobe.

```
    // файла Canvas.swf вы используете Flash IDE, вам потребуется вызвать
    // метод Canvas.main() из первого кадра ролика.
    static function main( ) { var canvas = new Canvas( ); }

    // Этот конструктор содержит код инициализации нашего Flash-класса Canvas
    function Canvas() {
        // Определить поведение холста при изменении размеров
        Stage.scaleMode = "noScale";
        Stage.align = "TL";

        // Импортировать функции рисования Flash API
        ExternalInterface.addCallback("beginFill", _root, _root.beginFill);
        ExternalInterface.addCallback("beginGradientFill", _root,
                                      _root.beginGradientFill);
        ExternalInterface.addCallback("clear", _root, _root.clear);
        ExternalInterface.addCallback("curveTo", _root, _root.curveTo);
        ExternalInterface.addCallback("endFill", _root, _root.endFill);
        ExternalInterface.addCallback("lineTo", _root, _root.lineTo);
        ExternalInterface.addCallback("lineStyle", _root, _root.lineStyle);
        ExternalInterface.addCallback("moveTo", _root, _root.moveTo);

        // А также экспортировать функцию addText(), представленную далее
        ExternalInterface.addCallback("addText", null, addText);
    }

    static function addText(text, x, y, w, h, depth, font, size) {
        // Создать объект TextField для визуализации текста
        // в заданных координатах
        var tf = _root.createTextField("tf", depth, x, y, w, h);
        // Представить выводимый текст
        tf.text = text;
        // Установить параметры шрифта текста
        var format = new TextFormat();
        format.font = font;
        format.size = size;
        tf.setTextFormat(format);
    }
}
```

Программный код файла *Canvas.as*, представленный в примере 22.12, должен быть скомпилирован в файл *Canvas.swf*, прежде чем его можно будет использовать в Flash-плеере. Подробное описание того, как это делается, выходит за рамки темы данной книги, но вы можете воспользоваться коммерческой версией Flash IDE компании Adobe или свободно распространяемым компилятором ActionScript.[1]

К сожалению, Flash предоставляет лишь низкоуровневый прикладной интерфейс. В частности, `curveTo()` – это единственная функция, рисующая кривые (точнее, кривые Безье второго порядка). Все окружности, эллипсы и кривые Бе-

[1] Я пользуюсь свободно распространяемым компилятором mtasc (*http://www.mtasc.org*) и компилировал файл командой `mtasc -swf Canvas.swf -main -version 8 -header 500:500:1 Canvas.as`. После компиляции получившийся файл имел размер всего 578 байт – много меньше, чем большинство растровых изображений.

зье третьего порядка приходится аппроксимировать простейшими кривыми второго порядка. Этот низкоуровневый прикладной интерфейс отлично подходит для создания Flash-роликов в скомпилированном формате SWF: все вычисления, необходимые для создания более сложных кривых, можно выполнить на этапе компиляции, и Flash-плееру достаточно уметь рисовать простейшие кривые. Высокоуровневый прикладной интерфейс можно выстроить поверх примитивов, предоставляемых Flash-плеером, и его вполне можно реализовать средствами ActionScript или JavaScript (пример 22.13 написан на языке JavaScript).

Пример 22.13 начинается с вспомогательной функции, выполняющей внедрение файла *Canvas.swf* в HTML-документ. В разных броузерах эта операция выполняется по-разному, а функция `insertCanvas()` скрывает эти различия. Вслед за ней идет функция `wedge()`, использующая прикладной интерфейс Flash для рисования сектора круговой диаграммы. Вслед за ней идет функция `pieChart()`, вызывающая функцию `wedge()` для рисования отдельного сектора. Заканчивается пример определением обработчика события `onload`, который вставляет Flash-холст в документ и создает на нем рисунок.

Пример 22.13. Рисование круговой диаграммы средствами JavaScript и Flash

```
<html>
<head>
<script>
// Встраивает Flash-холст заданного размера в виде единственного
// потомка указанного контейнерного элемента. Для переносимости функция
// использует тег <embed> в Netscape-совместимых броузерах и тег <object> - в остальных
// Идея взята из FlashObject, автор Джеф Стернс (Geoff Stearns).
// http://blog.deconcept.com/flashobject/
function insertCanvas(containerid, canvasid, width, height) {
    var container = document.getElementById(containerid);
    if (navigator.plugins && navigator.mimeTypes&&navigator.mimeTypes.length){
        container.innerHTML =
            "<embed src='Canvas.swf' type='application/x-shockwave-flash' " +
            "width='" + width +
            "' height='" + height +
            "' bgcolor='#ffffff' " +
            "id='" + canvasid +
            "' name='" + canvasid +
            "'>";
    }
    else {
        container.innerHTML =
            "<object classid='clsid:D27CDB6E-AE6D-11cf-96B8-444553540000' "+
            "width='" + width +
            "' height='" + height +
            "' id='"+ canvasid + "'>" +
            " <param name='movie' value='Canvas.swf'>" +
            " <param name='bgcolor' value='#ffffff'>" +
            "</object>";
    }
}

// Прикладной интерфейс Flash еще более низкоуровневый, чем другие, в нем
// имеется возможность создания лишь простейших кривых Безье.
```

```
// Данный метод рисует сектор, используя этот интерфейс.
// Обратите внимание: углы должны задаваться в радианах.
function wedge(canvas, cx, cy, r, startangle, endangle, color) {
    // Вычислить начальную точку сектора
    var x1 = cx + r*Math.sin(startangle);
    var y1 = cy - r*Math.cos(startangle);

    canvas.beginFill(color, 100); // Заливать указанным непрозрачным цветом
    canvas.moveTo(cx, cy);        // Перейти в центр окружности
    canvas.lineTo(x1, y1);        // Нарисовать линию до границы окружности

    // Разбить дугу на части меньше 45 градусов и рисовать каждую часть
    // отдельным вызовом вложенного метода arc() method
    while(startangle < endangle) {
        var theta;
        if (endangle-startangle > Math.PI/4) theta = startangle+Math.PI/4;
        else theta = endangle;
        arc(canvas,cx,cy,r,startangle,theta);
        startangle += Math.PI/4;
    }

    canvas.lineTo(cx, cy); // Завершить рисование линией к центру
    canvas.endFill();      // Залить сектор

    // Данная вложенная функция рисует часть окружности с помощью кривых Безье.
    // Разность endangle - startangle не должна превышать 45 градусов.
    // Текущей должна быть позиция в точке startangle.
    // Можете принять реализацию функции на веру, если вам сложно
    // понять математику, лежащую в ее основе.
    function arc(canvas, cx, cy, r, startangle, endangle) {
        // Вычислить конечную точку кривой
        var x2 = cx + r*Math.sin(endangle);
        var y2 = cy - r*Math.cos(endangle);

        var theta = (endangle - startangle)/2;
        // Это расстояние от центра до контрольной точки
        var l = r/Math.cos(theta);
        // Угол между центром дуги и контрольной точкой:
        var alpha = (startangle + endangle)/2;

        // Вычислить контрольную точку для кривой
        var controlX = cx + l * Math.sin(alpha);
        var controlY = cy - l * Math.cos(alpha);

        // Обратиться к Flash API, чтобы нарисовать дугу как кривую Безье.
        canvas.curveTo(controlX, controlY, x2, y2);
    }
}

/**
 * Рисует круговую диаграмму на Flash-холсте, заданном в виде ссылки
 * на элемент или в виде значения атрибута id.
 * data - массив чисел: каждое число соответствует сектору диаграммы.
 * Центр диаграммы находится в точке с координатами (cx, cy),
 * радиус задается аргументом r.
 * Аргумент colors - это Flash-значения цветов в виде массива colors[].
 * Легенда размещается, начиная с координат (lx,ly),
```

```
 * с метками из массива labels[].
 */
function pieChart(canvas, data, cx, cy, r, colors, labels, lx, ly) {
    // Получить элемент холста по id
    if (typeof canvas == "string")
        canvas = document.getElementById(canvas);

    // Все линии будут черного цвета, непрозрачные, толщиной 2 пиксела.
    canvas.lineStyle(2, 0x000000, 100);

    // Найти сумму всех значений данных
    var total = 0;
    for(var i = 0; i < data.length; i++) total += data[i];

    // И рассчитать угловые размеры (в радианах) для каждого сектора.
    var angles = []
    for(var i = 0; i < data.length; i++) angles[i] = data[i]/total*Math.PI*2;

    // Цикл по всем секторам диаграммы
    startangle = 0;
    for(var i = 0; i < data.length; i++) {
        // Это угол, где сектор заканчивается
        var endangle = startangle + angles[i];

        // Нарисовать сектор: эта функция была определена ранее
        wedge(canvas, cx, cy, r, startangle, endangle, colors[i]);

        // Следующий сектор начинается там, где закончился предыдущий.
        startangle = endangle;

        // Нарисовать прямоугольник в легенде
        canvas.beginFill(colors[i], 100);
        canvas.moveTo(lx, ly+30*i);
        canvas.lineTo(lx+20, ly+30*i);
        canvas.lineTo(lx+20, ly+30*i+20);
        canvas.lineTo(lx, ly+30*i+20);
        canvas.lineTo(lx, ly+30*i);
        canvas.endFill();

        // Добавить текст рядом с прямоугольником
        canvas.addText(labels[i], lx+30, ly+i*30, 100, 20, // Текст и его координаты
                       i, // Каждое текстовое поле должно иметь другую глубину
                       "Helvetica", 16);                  // Шрифт
    }
}

// Когда документ загрузится, вставить Flash-холст и создать на нем рисунок.
// Обратите внимание: Flash-значения цветов - это целые числа, а не строки
window.onload = function() {
    insertCanvas("placeholder", "canvas", 600, 400);
    pieChart("canvas", [12, 23, 34, 45], 200, 200, 150,
             [0xff0000, 0x0000ff, 0xffff00, 0x00ff00],
             ["Север", "Юг", "Восток", "Запад"],
            400, 100);
}
</script>
</head>
```

```
<body>
<div id="placeholder"></div>
</body>
</html>
```

22.7. Создание графики с помощью Java

Подключаемый Java-модуль производства компании Sun Microsystems не так широко распространен, как Flash-плеер, но становится все более и более популярным, поэтому многие производители компьютеров даже заранее устанавливают его на свои компьютеры. Java2D API – это мощный прикладной интерфейс создания векторной графики, появившийся в Java начиная с версии 1.2. Он является более высокоуровневым, чем Flash API, и обладает более богатыми возможностями (например, поддерживает работу с текстом), чем прикладной интерфейс тега <canvas>. По своим характеристикам Java2D больше напоминает SVG. В этом разделе демонстрируются два интересных способа использования прикладного интерфейса Java2D из клиентских JavaScript-сценариев.

22.7.1. Построение круговой диаграммы средствами Java

При использовании Java можно выбрать тот же самый подход, что и в случае с Flash-модулем: создать апплет с именем «Canvas», не имеющий собственного поведения и существующий лишь для экспорта прикладного интерфейса Java2D. Затем этот апплет можно будет вызывать из клиентских JavaScript-сценариев. (Подробнее о работе с Java-апплетами из JavaScript-сценариев рассказывается в главе 23.) Как может выглядеть такой апплет, показано в примере 22.14. Обратите внимание: этот апплет экспортирует лишь малую толику методов прикладного интерфейса Java2D. Интерфейс рисования Flash-модуля состоит всего из восьми методов, поэтому не составляет никакого труда экспортировать их все. Интерфейс Java2D содержит значительно больше методов. Чисто технически было бы совсем несложно сделать доступными все методы, но тогда апплет получился бы слишком объемным, чтобы приводить здесь его код. Программный код примера 22.14 демонстрирует базовый подход и обеспечивает достаточно богатый прикладной интерфейс создания круговой диаграммы, представленной на рис. 22.5.

Пример 22.14. Java-апплет Canvas.java, предназначенный для создания графических изображений на стороне клиента

```
import java.applet.*;
import java.awt.*;
import java.awt.geom.*;
import java.awt.image.*;

/**
 * Этот простой апплет сам ничего не делает: он просто экспортирует API
 * для использования в JavaScript-сценариях на стороне клиента.
 */
public class Canvas extends Applet {
    BufferedImage image; // Рисование будет производиться на невидимом изображении
    Graphics2D g;        // С использованием этого графического контекста
```

```
    // Этот метод вызывается броузером для инициализации апплета
    public void init() {
        // Определить размеры апплета и создать невидимое изображение
        // с этими размерами.
        int w = getWidth();
        int h = getHeight();
        image = new BufferedImage(w, h, BufferedImage.TYPE_INT_RGB);
        // Получить графический контекст для рисования на этом изображении
        g = image.createGraphics();
        // Сначала установить белый цвет фона
        g.setPaint(Color.WHITE);
        g.fillRect(0, 0, w, h);
        // Включить сглаживание
        g.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
                           RenderingHints.VALUE_ANTIALIAS_ON);
    }

    // Этот метод автоматически вызывается броузером, когда возникает
    // необходимость перерисовать апплет. Он копирует невидимое
    // изображение на экран. JavaScript-код, выполняющий рисование с помощью
    // этого апплета, должен вызвать унаследованный метод repaint().
    public void paint(Graphics g) { g.drawImage(image, 0, 0, this); }

    // Эти методы устанавливают основные параметры рисования
    // Это лишь часть того, что поддерживает Java2D API
    public void setLineWidth(float w) { g.setStroke(new BasicStroke(w)); }
    public void setColor(int color) { g.setPaint(new Color(color)); }
    public void setFont(String fontfamily, int pointsize) {
        g.setFont(new Font(fontfamily, Font.PLAIN, pointsize));
    }

    // Далее следуют простые примитивы рисования
    public void fillRect(int x, int y, int w, int h) { g.fillRect(x,y,w,h); }
    public void drawRect(int x, int y, int w, int h) { g.drawRect(x,y,w,h); }
    public void drawString(String s, int x, int y) { g.drawString(s, x, y); }

    // Эти методы выполняют заливку и рисование произвольных фигур
    public void fill(Shape shape) { g.fill(shape); }
    public void draw(Shape shape) { g.draw(shape); }

    // Эти методы возвращают простейшие фигуры
    // Это лишь пример. Java2D API поддерживает множество других методов.
    public Shape createRectangle(double x, double y, double w, double h) {
        return new Rectangle2D.Double(x, y, w, h);
    }
    public Shape createEllipse(double x, double y, double w, double h) {
        return new Ellipse2D.Double(x, y, w, h);
    }
    public Shape createWedge(double x, double y, double w, double h,
                             double start, double extent) {
        return new Arc2D.Double(x, y, w, h, start, extent, Arc2D.PIE);
    }
}
```

Данный апплет компилируется с помощью компилятора *javac*, который создает файл с именем *Canvas.class*:

```
% javac Canvas.java
```

Затем скомпилированный апплет *Canvas.class* можно внедрить в HTML-файл и манипулировать им, например, таким образом:

```
<head>
<script>
window.onload = function() {
    var canvas = document.getElementById('square');
    canvas.setColor(0x0000ff); // Обратите внимание: цвет - целое число
    canvas.fillRect(0,0,10,10);
    canvas.repaint();
    canvas = document.getElementById('circle');
    canvas.setColor(0xff0000);
    canvas.fill(canvas.createEllipse(0,0,10,10));
    canvas.repaint();
};
</script>
</head>
<body>
Это голубой квадрат:
<applet id="square" code="Canvas.class" width=10 height=10></applet>
Это красный круг:
<applet id="circle" code="Canvas.class" width=10 height=10></applet>
</body>
```

Этот фрагмент программы основан на обработчике события `onload` – он не запустится, пока апплет не будет полностью загружен и готов к работе. В старых броузерах и в подключаемых Java-модулях до версии 5 обработчик события `onload` часто вызывался до инициализации апплета, что приводило к сбоям в работе подобного программного кода. Когда рисование производится в ответ на другие пользовательские события, а не на событие `onload`, то проблем, как правило, не возникает.

В примере 22.15 приводится JavaScript-код, создающий круговую диаграмму с помощью Java-апплета `Canvas`. В этом примере отсутствует функция `makeCanvas()`, которая определена в других примерах. Кроме того, из-за проблемы с обработчиком события `onload`, описанной ранее, данный пример рисует диаграмму только после щелчка на кнопке, а не автоматически, после загрузки документа.

Пример 22.15. Рисование круговой диаграммы средствами JavaScript и Java

```
<head>
<script>
// Рисует круговую диаграмму с использованием Java-апплета Canvas
function pieChart(canvas, data, cx, cy, r, colors, labels, lx, ly) {
    // Отыскать рисунок по имени, если это необходимо
    if (typeof canvas == "string") canvas = document.getElementById(canvas);

    // Все линии будут иметь толщину 2 единицы. Текст будет выводиться
    // шрифтом sans-serif, полужирным начертанием, размером 16 пунктов.
    canvas.setLineWidth(2);
```

```
        canvas.setFont("SansSerif", 16);

        // Найти сумму всех значений
        var total = 0;
        for(var i = 0; i < data.length; i++) total += data[i];

        // Вычислить угловые размеры секторов в градусах
        var angles = []
        for(var i = 0; i < data.length; i++) angles[i] = data[i]/total*360;

        startangle = 90; // Начало отсчета в крайней верхней точке
        // Цикл по всем секторам
        for(var i = 0; i < data.length; i++) {
            // Этот объект описывает один сектор диаграммы
            var arc = canvas.createWedge(cx-r, cy-r, r*2, r*2,
                                         startangle, -angles[i]);
            canvas.setColor(colors[i]);           // Установить цвет
            canvas.fill(arc);                     // Залить сектор
            canvas.setColor(0x000000);            // Переключиться на черный цвет
            canvas.draw(arc);                     // Нарисовать рамку

            startangle -= angles[i];              // Для следующего сектора

            // Нарисовать прямоугольник для легенды
            canvas.setColor(colors[i]);           // Цвет сектора
            canvas.fillRect(lx, ly+30*i, 20, 20); // Залить прямоугольник
            canvas.setColor(0x000000);            // Переключиться на черный цвет
            canvas.drawRect(lx, ly+30*i, 20, 20); // Рамка прямоугольника

            // Нарисовать метку правее прямоугольника
            // Шрифт был установлен ранее
            canvas.drawString(labels[i], lx+30, ly+30*i+18);
        }

        // Отобразить апплет
        canvas.repaint(); // Не забывайте вызывать этот метод
    }

    // Эта функция вызывается в результате щелчка на кнопке Нарисовать!
    function draw() {
        pieChart("canvas", [12, 23, 34, 45], 200, 200, 150,
                 [0xff0000, 0x0000ff, 0xffff00, 0x00ff00], // Цвета - это целые
                 ["Север", "Юг", "Восток", "Запад"],
                 400, 100);
    }
    </script>
    </head>
    <body>
    <applet id="canvas" code="Canvas.class" width=600 height=400></applet>
    <button onclick="draw()">Нарисовать!</button>
    </body>
```

22.7.2. Создание на стороне клиента маленьких диаграмм в тексте с помощью Java

В этом разделе с помощью Java2D API мы будем создавать графические изображения, но не выводить их внутри апплета, а визуализировать в качестве потока

байтов формата PNG и затем преобразовывать в URL-адреса со спецификатором `data:`. Таким способом сценарии могут создавать собственные внутристрочные графические изображения. Хотя все то же самое можно сделать с помощью апплета, обсуждаемый здесь подход основан на непосредственном использовании Java-кода, что становится возможным благодаря технологии LiveConnect (см. главу 12), доступной в броузере Firefox и родственных ему броузерах.

Описываемый здесь подход позволяет выводить *внутристрочные диаграммы* (*sparklines*), представляющие собой графические изображения некоторых данных, непосредственно в поток текста. Вот пример такой диаграммы: Server load: 

Термин «sparkline» был введен их автором Эдвардом Тафтом (Edward Tufte), который описывает внутристрочные диаграммы так: «Маленькие графические изображения с высоким разрешением, встроенные в контекст окружающих их слов, чисел, изображений. Внутристрочная диаграмма – это простой в создании, тесно связанный с данными график, размер которого сопоставим с размером слова». (Подробнее о создании внутристрочных диаграмм можно узнать в книге Эдварда Тафта «Beautiful Evidence» [Graphics Press].)

В примере 22.16 приводится код, используемый для создания показанной здесь внутристрочной диаграммы загруженности сервера. JavaScript-функция `makeSparkline()` использует технологию LiveConnect для прямого (без промежуточного апплета) взаимодействия с прикладным интерфейсом Java2D.

Пример 22.16. Создание внутристрочной диаграммы средствами JavaScript и Java

```
<head>
<script>
/**
 * data - это массив чисел, которые должны быть представлены в виде
 * линейной диаграммы
 * dx - число пикселов между точками данных
 * config - это объект с данными, которые наверняка не будут изменяться
 * от вызова к вызову:
 *   height: высота изображения в пикселах
 *   ymin, ymax: диапазон значений по оси Y в пользовательском пространстве
 *   backgroundColor: цвет фона в числовом виде.
 *   lineWidth: толщина линии
 *   lineColor: цвет линии в виде числовой HTML-спецификации #
 *   dotColor: Если определено, последняя точка на графике будет залита этим цветом
 *   bandColor: Если определено, между значениями bandMin и bandMax будет
 *              нарисована полоса данного цвета, которая отображает
 *              "нормальный" диапазон значений данных с целью выделить
 *              значения, выходящие за этот диапазон
 */
function makeSparkline(data, dx, config) {
    var width = data.length * dx + 1; // Общая ширина изображения
    var yscale = config.height/(config.ymax - config.ymin); // Для масштабирования
    // Преобразует значения данных в пикселы
    function x(i) { return i * dx; }

    // Преобразует координату Y из системы координат пользовательского
    // пространства в систему координат изображения
```

```
function y(y) { return config.height - (y - config.ymin)*yscale; }

// Преобразует цвет из HTML-представления в объект java.awt.Color
function color(c) {
    c = c.substring(1);     // Удалить ведущий символ #
    if (c.length == (3)) { // Преобразовать в 6-символьный формат
                           // по мере необходимости
        c = c.charAt(0) + c.charAt(0) + c.charAt(1) + c.charAt(1) +
            c.charAt(2) + c.charAt(2);
    }
    var red = parseInt(c.substring(0,2), 16);
    var green = parseInt(c.substring(2,4), 16);
    var blue = parseInt(c.substring(4,6), 16);
    return new java.awt.Color(red/255, green/255, blue/255);
}

// Создать невидимое изображение для диаграммы
var image = new java.awt.image.BufferedImage(width, config.height,
                                java.awt.image.BufferedImage.TYPE_INT_RGB);

// Получить объект Graphics, который позволит рисовать на изображении
var g = image.createGraphics( );

// Включить сглаживание. Это сделает линии более гладкими, но менее четкими
g.setRenderingHint(java.awt.RenderingHints.KEY_ANTIALIASING,
                   java.awt.RenderingHints.VALUE_ANTIALIAS_ON);

// Залить изображение цветом фона
g.setPaint(color(config.backgroundColor));
g.fillRect(0, 0, width, config.height);

// Если свойство bandColor определено, нарисовать полосу
if (config.bandColor) {
    g.setPaint(color(config.bandColor));
    g.fillRect(0, y(config.bandMax),
               width, y(config.bandMin)-y(config.bandMax));
}

// Нарисовать линию графика
var line = new java.awt.geom.GeneralPath( );
line.moveTo(x(0), y(data[0]));
for(var i = 1; i < data.length; i++) line.lineTo(x(i), y(data[i]));

// Сначала установить цвет линии и ее толщину, затем нарисовать
g.setPaint(color(config.lineColor));                      // Цвет линии
g.setStroke(new java.awt.BasicStroke(config.lineWidth)); // Толщина
g.draw(line);                                             // Нарисовать!

// Если свойство dotColor определено, нарисовать точку
if (config.dotColor) {
    g.setPaint(color(config.dotColor));
    var dot=new java.awt.geom.Ellipse2D$Double(x(data.length-1)-.75,
                                               y(data[data.length-1])-.75,
                                               1.5, 1.5)
    g.draw(dot);
}

// Записать изображение как массив байтов в формате PNG
```

```
    var stream = new java.io.ByteArrayOutputStream();
    Packages.javax.imageio.ImageIO.write(image, "png", stream);
    var imageData = stream.toByteArray( );

    // Преобразовать данные в строку URL-адреса
    var rawString = new java.lang.String(imageData, "iso8859-1");
    var encodedString = java.net.URLEncoder.encode(rawString, "iso8859-1");
    encodedString = encodedString.replaceAll("\\+", "%20");

    // И вернуть ее в виде URL-адреса со спецификатором data:
    return "data:image/png," + encodedString;
}

// Далее приводится пример использования функции makeSparkline()
window.onload = function() {
    // Создать тег img для размещения диаграммы
    var img = document.createElement("img");
    img.align = "center";
    img.hspace = 1;

    // Установить атрибут src в значение URL-адреса диаграммы со спецификатором data:
    img.src = makeSparkline([3, 4, 5, 6, 7, 8, 8, 9, 10, 10, 12,
                             16, 11, 10, 11, 10, 10, 10, 11, 12,
                             16, 11, 10, 11, 10, 10, 10, 11, 12,
                             14, 16, 18, 18, 19, 18, 17, 17, 16,
                             14, 16, 18, 18, 19, 18, 17, 17, 16],
                            2, { height: 20, ymin: 0, ymax: 20,
                                 backgroundColor: "#fff",
                                 lineWidth: 1, lineColor: "#000",
                                 dotColor: "#f00", bandColor: "#ddd",
                                 bandMin: 6, bandMax: 14
                               });

    // Отыскать элемент-заполнитель для диаграммы
    var placeholder = document.getElementById("placeholder");

    // И заменить его изображением.
    placeholder.parentNode.replaceChild(img, placeholder);
}
</script>
</head>
<body>
Server load: <span id="placeholder"></span><span style="color:#f00">16</span>
</body>
```

23

Сценарии с Java-апплетами и Flash-роликами

Модуль расширения, или подключаемый модуль (*plug-in*), – это программный модуль, который может быть «подключен» к веб-броузеру для расширения его функциональных возможностей. Два наиболее распространенных (и, что не случайно, наиболее мощных) модуля расширения – это Java-модуль компании Sun Microsystems и Flash-плеер компании Adobe (которая приобрела компанию Macromedia). Java-модуль позволяет броузерам запускать приложения, известные как *апплеты*, написанные на языке программирования Java. Система безопасности Java не позволяет апплетам, полученным из источников, не пользующихся доверием, работать с файлами в локальной файловой системе или выполнять другие действия, которые могут привести к изменению данных или нарушению конфиденциальности. Несмотря на ограничения, накладываемые на апплеты системой безопасности, в составе Java-модуля распространяется огромная библиотека предопределенных классов, которые могут использоваться апплетами. Эта библиотека включает пакеты для работы с графикой и разработки графического интерфейса пользователя, пакеты с мощными сетевыми возможностями, пакеты для синтаксического разбора XML-документов и манипулирования ими, пакеты, реализующие криптографические алгоритмы. К моменту написания этих строк предварительный выпуск Java 6 включал полный комплект пакетов для поддержки веб-служб.

Модуль Flash-плеер приобрел небывалую популярность и повсеместное распространение. Это «виртуальная» машина, выполняющая интерпретацию «роликов», которые могут включать в себя потоковое видео, но обычно содержат анимацию и богатый графический интерфейс. Flash-ролики могут включать ActionScript-сценарии. ActionScript – это разновидность языка JavaScript, дополненного конструкциями объектно-ориентированного программирования, такими как классы, статические методы и (необязательно) типы переменных. Программный код ActionScript-сценариев в Flash-роликах обладает доступом к мощной (хотя и не такой обширной, как в Java-модуле) библиотеке программного кода, обеспечивающей работу с графикой и сетями, а также манипулирование XML-документами.

По отношению к Java и Flash применять термин *модуль расширения* было бы не совсем корректно. Это не просто дополнения к броузеру – оба этих модуля расширения прекрасно подходят для нужд разработки приложений в их окружении и оба соответствуют требованиям пользователя в большей степени, чем веб-приложения на базе DHTML. Как только вы начинаете понимать, насколько богатые возможности привносят эти модули в броузер, тут же возникает вполне естественное желание использовать эти возможности в JavaScript-сценариях. К счастью, все это вполне возможно. JavaScript-сценарий может взаимодействовать и с Java-апплетами, и с Flash-роликами. Возможно также обратное: Java-апплеты и Flash-ролики способны вызывать JavaScript-функции. В данной главе рассказывается о том, как это делается. Однако следует предупредить, что интерфейсы между JavaScript, Java и ActionScript не отличаются удобством, и если вы задумаете реализовать серьезное взаимодействие своего кода с Java- и Flash-модулями, то наверняка столкнетесь с фактами несовместимости, ошибками и прочими неприятностями.

В начале этой главы описывается, как из клиентских JavaScript-сценариев взаимодействовать с Java-апплетами. (Вспомните пример 22.14, где Java-апплет использовался для создания графического изображения.) Затем рассказывается о том, как в Firefox и родственных ему броузерах организовать непосредственное взаимодействие JavaScript-сценария с подключаемым Java-модулем даже в отсутствие апплета. (Этот прием продемонстрирован в примере 22.16.)

После описания механизмов взаимодействия JavaScript-сценариев с Java мы перейдем к вопросам создания апплетов, способных обращаться к JavaScript-свойствам и вызывать JavaScript-методы, в том числе таких апплетов, которые используют Java-версию DOM API для взаимодействия с документом, выводимым в веб-броузере.

Языкам Java и JavaScript посвящена также глава 12, однако она существенно отличается от данной главы. В главе 12 рассказывается о том, как встроить интерпретатор JavaScript в Java-приложение и как с помощью этого интерпретатора запускать сценарии, чтобы обеспечить взаимодействие с Java-приложениями. Глава 12 никак не затрагивает темы клиентского JavaScript-кода и апплетов. В то же время в ней описывается технология LiveConnect, которая позволяет JavaScript-сценарию взаимодействовать с Java, и это описание оказалось бы вполне к месту и в данной главе. Однако следует отметить, что функциональные возможности, описанные в главе 12, относятся к «Rhino-версии LiveConnect» и не пригодны для клиентского JavaScript-кода и апплетов.

Раздел этой главы с описанием Java предполагает, что у вас есть, по меньшей мере, базовые навыки программирования на языке Java. Если вы не используете апплеты в своих веб-страницах, можете просто пропустить этот раздел.

После того как будет закрыта тема Java, я перейду к вопросам организации взаимодействий с Flash – мы поговорим о том, как из JavaScript-сценариев вызывать ActionScript-методы, определенные внутри Flash-ролика, и как из ActionScript-кода, включенного в ролик, вызывать JavaScript-методы. Эти темы обсуждаются дважды, первый раз в контексте всех последних версий Flash, второй только в контексте Flash версии 8 и выше.

Благодаря своим возможностям технология Flash уже упоминалась в предыдущих главах этой книги. В главе 22 Flash-плеер с помощью простого ActionScript-сценария обеспечивал динамическое создание графических изображений на стороне клиента (см. пример 22.12), а в главе 19 мы использовали возможности Flash-плеера для сохранения данных на стороне клиента (см. пример 19.4).

23.1. Работа с апплетами

Для взаимодействия с апплетом сначала нужно суметь обратиться к HTML-элементу, в котором этот апплет содержится. В главе 15 говорилось, что Java-апплеты, встроенные в веб-страницу, становятся элементами массива `Document.applets[]`. А если в апплете указан атрибут `name` или `id`, то к апплету можно обращаться напрямую как к свойству объекта `Document`. Например, к апплету, созданному с помощью тега `<applet name="chart">`, можно обратиться так: `document.chart`. А если у апплета установлен атрибут `id`, этот апплет можно отыскать с помощью метода `Document.getElementById()`.

После того как ссылка на апплет получена, общедоступные поля и методы этого апплета становятся доступными JavaScript-коду, как если бы они были свойствами и методами самого HTML-элемента `<applet>`. Рассмотрим в качестве примера апплет `Canvas`, который был определен в примере 22.14. Если апплет встроить в HTML-страницу со значением `"canvas"` атрибута `id`, то станет возможным использовать следующий фрагмент для вызова методов апплета:

```
var canvas = document.getElementById('canvas');
canvas.setColor(0x0000ff);
canvas.fillRect(0,0,10,10);
canvas.repaint( );
```

JavaScript может даже читать и устанавливать значения полей, являющихся массивами. Предположим, что апплет со значением атрибута `name="chart"` определяет два поля, объявленные следующим образом (Java-код):

```
public int numPoints;
public double[] points;
```

JavaScript-программа может использовать эти поля так:

```
for(var i = 0; i < document.chart.numPoints; i++)
    document.chart.points[i] = i*i;
```

Этот пример иллюстрирует сложный момент, относящийся к взаимодействию между Java и JavaScript, – *преобразование типов*. Java – это строго типизированный язык с большим количеством отдельных элементарных типов. Язык JavaScript слабо типизирован и имеет только один числовой тип. В предыдущем примере целое число (`integer`) языка Java преобразуется в JavaScript-число, а различные JavaScript-числа – в Java-значения типа `double`. Чтобы эти значения были нужным образом преобразованы, выполняется очень много закулисной работы. Тема преобразования типов данных при взаимодействии JavaScript и Java уже обсуждалась в главе 12, и возможно теперь у вас возникнет желание вернуться к ней. В третьей части книги вы найдете интересные сведения в разделах, посвященных классам `JavaObject`, `JavaArray`, `JavaClass` и `JavaPackage`. Обратите внимание: в главе 12 рассматривалась технология LiveConnect, родившаяся

в компании Netscape, и не все броузеры используют эту технологию. Например, IE имеет собственную технологию взаимодействия между JavaScript и Java на основе ActiveX. Тем не менее независимо от базовой технологии основные правила преобразования значений между Java и JavaScript остаются более или менее одинаковыми для всех броузеров.

Наконец, очень важно отметить, что Java-методы могут возвращать Java-объекты, и JavaScript может обращаться к общедоступным полям и вызывать общедоступные методы этих объектов, точно так же, как поля и методы апплета. Вернемся еще раз к апплету `Canvas` из примера 22.14. Он определяет метод, который может возвращать объекты `Shape`. Программный JavaScript-код может вызывать методы этих объектов `Shape` и передавать их другим методам апплета, которые ожидают получить в качестве аргумента объект `Shape`.

В примере 23.1 приводится типичный Java-апплет, который ничего не делает, но определяет полезный для использования в JavaScript-сценариях метод. Этот метод `getText()` читает URL-адрес (который должен ссылаться на сервер, откуда был получен апплет) и возвращает его содержимое в виде Java-строки. Вот пример апплета, обеспечивающего вывод HTML-файлом своего содержимого:

```
<!-- Вывод содержимого HTML-файла с помощью апплета -->
<body onload="alert(document.http.getText('GetText.html'));">
<applet name="http" code="GetTextApplet.class" width="1" height="1"></applet>
</body>
```

В примере 23.1 использованы базовые Java-классы, предназначенные для работы с сетью, для ввода-вывода и для манипулирования текстом, но здесь не делается ничего особенно сложного. В этом примере просто определяется удобный метод, который объявляется общедоступным, благодаря чему появляется возможность обращаться к нему из JavaScript-сценариев.

Пример 23.1. Апплет, пригодный для использования в сценариях

```
import java.applet.Applet;
import java.net.URL;
import java.io.*;

public class GetTextApplet extends Applet {
    public String getText(String url)
        throws java.net.MalformedURLException, java.io.IOException
    {
        URL resource = new URL(this.getDocumentBase(), url);
        InputStream is = resource.openStream();
        BufferedReader in = new BufferedReader(new InputStreamReader(is));
        StringBuilder text = new StringBuilder();
        String line;
        while((line = in.readLine()) != null) {
            text.append(line);
            text.append("\n");
        }
        in.close();
        return text.toString();
    }
}
```

23.2. Работа с подключаемым Java-модулем

Броузер Firefox и родственные ему броузеры могут взаимодействовать не только с апплетами, но и непосредственно с подключаемыми Java-модулями, что исключает потребность в апплете. Благодаря технологии LiveConnect JavaScript-сценарии, исполняющиеся в этих броузерах, могут создавать и использовать экземпляры Java-объектов даже в отсутствие апплета. Однако данный прием непереносим, и в таких броузерах, как Internet Explorer, он работать не будет.

В броузерах, поддерживающих возможность взаимодействия с подключаемым Java-модулем, объект `Packages` предоставляет доступ ко всем Java-пакетам, которые известны броузеру. Выражение `Packages.java.lang` ссылается на пакет *java.lang*, а выражение `Packages.java.lang.System` — на класс *java.lang.System*. Для удобства идентификатор `java` представляет собой сокращенный вариант записи `Packages.java` (подробности см. в третьей части книги в разделе, посвященном свойству `Packages`). Так, JavaScript-сценарий может вызвать статический метод класса *java.lang.System* следующим образом:

```
// Вызов статического метода System.getProperty()
var javaVersion = java.lang.System.getProperty("java.version");
```

Однако обращаться можно не только к статическим методам и свойствам объектов: технология LiveConnect позволяет создавать новые экземпляры Java-классов с помощью оператора `new` языка JavaScript. В примере 23.2 демонстрируется JavaScript-сценарий, который создает средствами Java новое окно и выводит в нем сообщение. Обратите внимание: программный код JavaScript-сценария очень напоминает Java-код. Ранее этот фрагмент демонстрировался в главе 12, но здесь он встроен в тег `<script>` внутри HTML-файла. На рис. 23.1 показано окно, созданное средствами Java при запуске этого сценария в броузере Firefox.

Пример 23.2. Взаимодействие с подключаемым Java-модулем

```
<script>
// Определить идентификатор для упрощенного обращения к иерархии
// пакета javax.*
var javax = Packages.javax;

// Создать некоторые Java-объекты
var frame = new javax.swing.JFrame("Hello World");
```

***Рис. 23.1.** Окно, созданное средствами Java из JavaScript-сценария*

```
var button = new javax.swing.JButton("Hello World");
var font = new java.awt.Font("SansSerif", java.awt.Font.BOLD, 24);

// Вызвать методы новых объектов
frame.add(button);
button.setFont(font);
frame.setSize(300, 200);
frame.setVisible(true);
</script>
```

Когда сценарий взаимодействует напрямую с подключаемым Java-модулем, он подвергается тем же ограничениям в области безопасности, что и апплеты, полученные из источников, не вызывающих доверия. JavaScript-сценарии, например, не могут использовать класс *java.io.File*, потому что это даст им возможность читать, писать и удалять файлы в файловой системе клиента.

23.3. Взаимодействие с JavaScript-сценариями из Java

Выяснив, как управлять Java-кодом из JavaScript-сценария, мы можем перейти к противоположной задаче: управлению JavaScript-кодом из Java. Любое взаимодействие Java с JavaScript-сценарием осуществляется через экземпляр класса `netscape.javascript.JSObject`. (Полное описание класса `JSObject` вы найдете в части IV книги.) Экземпляр этого класса представляет собой обертку для одного JavaScript-объекта. Класс определяет методы, позволяющие читать и изменять значения свойств и элементов массивов в JavaScript-объектах, а также вызывать методы объекта. Вот определение этого класса:

```
public final class JSObject extends Object {
    // Этот статический метод возвращает начальный объект JSObject
    // для окна броузера.
    public static JSObject getWindow(java.applet.Applet applet);

    // Эти методы экземпляра используются для манипулирования объектом
    public Object getMember(String name); // Чтение свойства объекта
    public Object getSlot(int index);     // Чтение элемента массива
    public void setMember(String name, Object value); // Запись в свойство
    public void setSlot(int index, Object value);     // Запись в элемент
    public void removeMember(String name);            // Удаление свойства
    public Object call(String name, Object args[]);   // Вызов метода
    public Object eval(String s);         // Вычисление строки
    public String toString();             // Преобразование в строку
    protected void finalize();
}
```

Класс `JSObject` не имеет конструктора. Первый объект `JSObject` Java-апплет получает с помощью статического метода `getWindow()`. Метод, которому передается ссылка на апплет, возвращает объект `JSObject`, представляющий окно броузера, содержащее апплет. Следовательно, любой апплет, взаимодействующий с JavaScript-кодом, включает строку, которая выглядит примерно так:

```
JSObject win = JSObject.getWindow(this); // "this" - это сам апплет
```

Получив объект JSObject, ссылающийся на объект Window, можно посредством методов экземпляра этого начального объекта получить доступ к другим объектам JSObject, представляющим другие JavaScript-объекты:

```
import netscape.javascript.JSObject; // Это объявление должно стоять в начале файла
...
// Получить начальный объект JSObject, представляющий объект Window
JSObject win = JSObject.getWindow(this); // window
// С помощью метода getMember() получить ссылку на объект JSObject,
// представляющий объект Document
JSObject doc = (JSObject)win.getMember("document"); // .document
// С помощью метода call() получить ссылку на объект JSObject,
// представляющий элемент документа
JSObject div = (JSObject)doc.call("getElementById", // .getElementById('test')
                                  new Object[] { "test" });
```

Здесь следует обратить внимание на два момента. Во-первых, методы getMember() и call() возвращают значение типа Object, которое, как правило, должно быть преобразовано в некоторое более конкретное значение, такое как JSObject. Во-вторых, когда с помощью метода call() вызывается JavaScript-метод, аргументы передаются в виде массива Java-значений Object. Этот массив должен указываться обязательно, даже если метод ожидает получить единственный аргумент.

Класс JSObject имеет еще один важный метод eval(). Этот Java-метод работает так же, как одноименная JavaScript-функция: он исполняет строку, содержащую JavaScript-код. Работать с методом eval() много проще, чем с другими методами класса JSObject. Например, вот как установить CSS-стили элемента документа методом eval():

```
JSObject win = JSObject.getWindow(this);
win.eval("document.getElementById('test').style.backgroundColor = 'gray';");
```

Чтобы сделать то же самое без использования метода eval(), придется написать такой фрагмент:

```
JSObject win = JSObject.getWindow(this);            // window
JSObject doc = (JSObject)win.getMember("document"); // .document
JSObject div = (JSObject)doc.call("getElementById", // .getElementById('test')
new Object[] { "test" });
JSObject style = (JSObject)div.getMember("style");  // .style
style.setMember("backgroundColor", "gray");         // .backgroundColor="gray"
```

23.3.1. Компиляция и распространение апплетов, использующих класс JSObject

Прежде чем распространять апплет, его необходимо скомпилировать, а затем встроить в HTML-файл. Если апплет использует класс JSObject, для выполнения обоих шагов требуются особые инструкции.

Чтобы скомпилировать апплет, взаимодействующий с JavaScript, необходимо сообщить компилятору, где найти определение класса netscape.javascript.JSObject. Ранее, когда броузеры распространялись с собственным интерпретаторами Java, ответить на этот вопрос было достаточно сложно. Однако ныне, когда все броузеры используют подключаемый Java-модуль компании Sun Microsystems,

все стало гораздо проще. Класс `JSObject` находится в файле *jre/lib/plugin.jar* дистрибутива Java. Таким образом, чтобы скомпилировать апплет, использующий объект `JSObject`, можно выполнить следующую команду, подставив в нее имя каталога, в который был установлен Java-пакет:

```
% javac -cp /usr/local/java/jre/lib/plugin.jar ScriptedApplet.java
```

На использование апплетов, взаимодействующих с JavaScript-сценариями, накладываются дополнительные ограничения в области безопасности, согласно которым апплет не может взаимодействовать с JavaScript-сценарием, если автор веб-страницы (который, возможно, не является создателем апплета) явно не даст разрешение на такое взаимодействие. Чтобы выдать такое право, необходимо в тег апплета `<applet>` (или `<object>`, или `<embed>`) включить атрибут `mayscript`. Например:

```
<applet code="ScriptingApplet.class" mayscript width="300" height="300">
</applet>
```

В случае отсутствия атрибута `mayscript` апплет не сможет использовать класс `JSObject`.

23.3.2. Преобразование типов данных между Java и JavaScript

При обращении к методам или установке значений полей класс `JSObject` должен выполнять преобразование типов данных Java-значений в JavaScript-значения, а при передаче возвращаемого значения или чтении полей выполнять обратное преобразование. Преобразование типов, выполняемое объектом `JSObject`, несколько отличается от описанного в главе 12 преобразования, выполняемого в рамках технологии LiveConnect. К сожалению, преобразования, выполняемые классом `JSObject`, в большей степени зависят от платформы, чем преобразования при манипулировании Java-кодом из JavaScript-сценария.

Когда Java-код читает JavaScript-значение, преобразование выполняется достаточно просто:

- JavaScript-числа преобразуются в тип `java.lang.Double`.
- JavaScript-строки преобразуются в тип `java.lang.String`.
- Логические JavaScript-значения преобразуются в тип `java.lang.Boolean`.
- JavaScript-значение `null` преобразуется в Java-значение `null`.
- Преобразование JavaScript-значения `undefined` определяется платформой: с установленным модулем Java 5 в Internet Explorer значение `undefined` преобразуется в значение `null`, а в Firefox – в строку `"undefined"`.

Когда Java-код устанавливает значения JavaScript-свойств или передает аргументы JavaScript-методам, преобразование должно было бы выполняться не менее просто, но, к сожалению, для разных платформ оно выполняется по-разному. В Firefox 1.0 с установленным модулем Java 5 Java-значения не преобразуются, и JavaScript-сценарий получает их в виде объектов `JavaObject` (с которыми он может взаимодействовать в рамках технологии LiveConnect).

В IE 6 с установленным модулем Java 5 преобразования выполняются более естественным образом:

- Java-числа и Java-символы преобразуются в JavaScript-числа.
- Java-объекты `String` преобразуются в JavaScript-строки.
- Логические Java-значения преобразуются в логические JavaScript-значения.
- Java-значение `null` преобразуется в JavaScript-значение `null`.
- Любые другие Java-значения преобразуются в Java-объекты `JavaObject`.

Из-за таких существенных отличий между Firefox и IE необходимо с особой осторожностью подходить в JavaScript-сценариях к операциям, требующим преобразования значений. Например, когда создается функция, которая будет вызываться апплетом, необходимо явно выполнять преобразование аргументов с помощью функций `Number()`, `String()` и `Boolean()`, прежде чем пользоваться этими аргументами.

Чтобы вообще избавиться от проблемы преобразования типов данных, можно порекомендовать пользоваться методом `JSObject.eval()` везде, где требуется организовать взаимодействие с JavaScript-сценарием.

23.3.3. Common DOM API

В Java версии 1.4 и выше подключаемый Java-модуль включает прикладной интерфейс Common DOM, являющийся Java-реализацией модели DOM Level 2 поверх класса `netscape.javascript.JSObject`. Этот прикладной интерфейс позволяет Java-апплетам взаимодействовать с документом, в который они встраиваются путем Java-привязки прикладного интерфейса модели DOM.

Сама идея просто захватывающая, но ее реализация оставляет желать лучшего. Одна из серьезных проблем заключается в том, что реализация (как в IE, так и в Firefox), похоже, неспособна создавать новые текстовые узлы или получать существующие текстовые узлы, что делает прикладной интерфейс Common DOM бесполезным для извлечения и изменения содержимого документа. Тем не менее некоторые DOM-операции поддерживаются, и в примере 23.3 показано, как апплет может использовать Common DOM API для установки CSS-стилей в HTML-элементе.

Относительно этого примера следует сделать несколько замечаний. Во-первых, прикладной интерфейс, предназначенный для манипулирования моделью DOM, выглядит несколько необычно. Весь программный код, выполняющий DOM-операции, помещается в тело метода `run()` объекта `DOMAction`. Затем объект `DOMAction` передается методу объекта `DOMService`. Когда вызывается метод `run()`, ему передается объект `DOMAccessor`, который может использоваться для доступа к объекту `Document`, представляющему корень иерархии DOM-объектов.

Во-вторых, Java-привязка прикладного интерфейса модели DOM более громоздкая и неуклюжая, чем JavaScript-привязка того же самого прикладного интерфейса. Наконец, следует отметить, что задача, решаемая кодом этого примера, может быть легко решена передачей строки JavaScript-кода методу `JSObject.eval()`!

Программный код примера 23.3 явно не использует класс `JSObject` и потому при его компиляции не требуется вставлять дополнительные пути к классам.

Пример 23.3. Апплет, использующий Common DOM API

```
import java.applet.Applet;          // Сам класс Applet
import com.sun.java.browser.dom.*; // Common DOM API
import org.w3c.dom.*;              // W3C core DOM API
import org.w3c.dom.css.*;          // W3C CSS DOM API

// Этот апплет ничего сам не делает. Он просто определяет метод для вызова
// JavaScript-кода. Этот метод использует прикладной интерфейс Common DOM
// для выполнения операций с документом, в который встроен этот апплет.
public class DOMApplet extends Applet {
    // Устанавливает в указанном элементе заданный цвет фона.
    public void setBackgroundColor(final String id, final String color)
        throws DOMUnsupportedException, DOMAccessException
    {
        // Сначала следует получить объект DOMService
        DOMService service = DOMService.getService(this);

        // Затем вызвать метод invokeAndWait() объекта DOMService
        // и передать ему объект DOMAction
        service.invokeAndWait(new DOMAction() {
            // Все DOM-операции размещаются в теле метода run()
            public Object run(DOMAccessor accessor) {
                // Для получения объекта Document используется DOMAccessor
                // Обратите внимание: в качестве аргумента передается объект апплета
                Document d = accessor.getDocument(DOMApplet.this);

                // Получить требуемый элемент
                Element e = d.getElementById(id);

                // Привести тип элемента к ElementCSSInlineStyle, чтобы можно
                // было вызвать его метод getStyle(). Затем возвращаемое
                // значение метода привести к типу CSS2Properties.
                CSS2Properties style =
                 (CSS2Properties) ((ElementCSSInlineStyle)e).getStyle( );

                // Наконец, можно установить значение свойства
                style.setBackgroundColor(color);

                // DOMAction может возвращать значение, но это не тот случай
                return null;
            }
        });
    }
}
```

23.4. Взаимодействие с Flash–роликами

Выяснив, как JavaScript-сценарии могут взаимодействовать с Java-кодом и наоборот, можно перейти к рассмотрению Flash-плеера и организации взаимодействий с Flash-роликами. В следующих подразделах описываются различные уровни взаимодействий с Flash. Во-первых, JavaScript-сценарий может управлять самим Flash-плеером: запускать и останавливать ролики, выполнять переход к определенному кадру и т. д. Более интересная тема – фактический вызов ActionScript-функций, определенных внутри Flash-ролика. В этом разделе продемонстрированы некоторые трюки, которые требовались для этого до выхода Flash 8.

Затем сценарии поменяются ролями, и вы увидите, как ActionScript-код может взаимодействовать с JavaScript-сценарием. Далее в подразделе 23.4.5 вашему вниманию будет представлен пример, состоящий из двух частей (JavaScript- и ActionScript-сценариев) и демонстрирующий двунаправленное взаимодействие между JavaScript и ActionScript. В разделе 23.5 показан тот же пример, но переписанный с учетом особенностей Flash 8.

Flash – это больше, чем просто ActionScript-код, большинство разработчиков Flash-содержимого пользуются коммерческой версией среды разработки Flash компании Adobe. Тем не менее все относящиеся к технологии Flash примеры из этой главы содержат только фрагменты ActionScript-кода, которые могут быть легко преобразованы в SWF-файлы (т. е. в Flash-ролики) с помощью свободно распространяемого компилятора ActionScript под названием *mtasc* (*http://www.mtasc.org*). Примеры роликов не содержат дополнительных аудио- и видеоданных и поэтому могут быть скомпилированы без использования дорогой среды разработки.

Данная глава не ставит перед собой цель обучить вас программированию на языке ActionScript или использованию прикладного интерфейса библиотек, доступных Flash-плееру. В этом вам помогут многочисленные ресурсы в Интернете, один из которых может оказаться наиболее полезным, – это словарь языка ActionScript, расположенный по адресу *http://www.adobe.com/ support/flash/action_scripts/actionscript_dictionary/*.

23.4.1. Встраивание и доступ к Flash-роликам

Прежде чем начать взаимодействовать с Flash-роликом, его необходимо встроить в HTML-страницу, чтобы JavaScript-сценарий мог получить ссылку на него. Однако сделать это не так просто, поскольку в разных броузерах это делается поразному: IE требует, чтобы ролик был включен в тег `<object>` с определенными атрибутами, другие броузеры – в тег `<object>` с другими атрибутами или в тег `<embed>`. Тег `<object>` – это стандартный HTML-тег, тем не менее в описываемых здесь приемах взаимодействия Flash с JavaScript-сценариями предполагается, что ролик встроен в тег `<embed>`.

Решение проблемы основано на специфичных для IE условных HTML-комментариях, с помощью которых тег `<object>` скрывается во всех броузерах, отличных от IE, а тег `<embed>` скрывается в IE. Вот пример встраивания Flash-ролика *mymovie.swf* под именем «movie»:

```
<!--[if IE]>
<object id="movie" type="application/x-shockwave-flash"
        width="300" height="300">
  <param name="movie" value="mymovie.swf">
</object>
<![endif]--><!--[if !IE]> <-->
<embed name="movie" type="application/x-shockwave-flash"
       src="mymovie.swf" width="300" height="300">
</embed>
<!--> <![endif]-->
```

В теге <object> устанавливается атрибут id, а в теге <embed> – атрибут name. Это общераспространенная практика, которая позволяет сослаться на встроенный элемент независимо от типа броузера с помощью следующего фрагмента:

```
// Получить ссылку на Flash-ролик из объекта Window в IE
// и из объекта Document в других броузерах
var flash = window.movie || document.movie; // Получить Flash-объект
```

Для нормального взаимодействия Flash-ролика с JavaScript-сценарием тег <embed> обязательно должен иметь атрибут name. Кроме того, чтобы обеспечить переносимость сценариев, нужно определить атрибут id и использовать метод getElementById() для поиска тегов <object> и <embed>.

23.4.2. Управление Flash–плеером

После того как Flash-ролик внедрен в HTML-страницу и JavaScript-сценарий получил ссылку на HTML-элемент, в который внедрен ролик, можно манипулировать Flash-плеером, просто вызывая методы этого элемента. Вот несколько примеров действий, которые можно выполнять с помощью этих методов:

```
var flash = window.movie || document.movie; // Получить ссылку на объект с роликом
if (flash.IsPlaying()) { // Если идет воспроизведение ролика,
    flash.StopPlay();    // остановить его
    flash.Rewind();      // и перейти в начало ролика
}
if (flash.PercentLoaded( ) == 100) // Если ролик полностью загружен,
    flash.Play();                  // запустить его.
flash.Zoom(50);                    // Масштаб 50 %
flash.Pan(25, 25, 1);              // Сместить вправо и вниз на 25 %
flash.Pan(100, 0, 0);              // Сместить на 100 пикселов вправо
```

Эти методы Flash-плеера описываются в соответствующем разделе части IV книги, а некоторые из них демонстрируются в примере 23.5.

23.4.3. Взаимодействие с Flash–роликами

Более интересная тема, чем управление Flash-плеером, – это вызов ActionScript-методов, определенных в самом ролике. В случае с Java-апплетами это делалось достаточно просто: требуемый метод вызывался просто как метод тега <applet>. Во Flash 8 вызов методов производится так же просто. Но если вы ориентируетесь на широкий круг пользователей, у многих из которых может быть установлена старая версия Flash-плеера, вам придется сделать несколько дополнительных шагов.

Один из основных методов управления Flash-плеером называется SetVariable(), другой – GetVariable(). Эти методы могут использоваться для записи и получения значений ActionScript-переменных. Хотя метода InvokeFunction() здесь нет, можно взять на вооружение ActionScript-расширение для JavaScript и с целью обращения к функциям использовать метод SetVariable(). В ActionScript каждый объект имеет метод watch(), который может устанавливать *контрольные точки* на манер отладчика: когда изменяется значение заданного свойства объекта, вызывается указанная функция. Рассмотрим следующий фрагмент ActionScript-кода:

```
/* ActionScript */
// Определить некоторые переменные для хранения аргументов
// функции и возвращаемого значения
// _root ссылается на начало временной шкалы ролика. Функции SetVariable()
// и GetVariable() могут читать и изменять значения свойств этого объекта.
_root.arg1 = 0;
_root.arg2 = 0;
_root.result = 0;
// Данная переменная определяется для вызова функции
_root.multiply = 0;
// Теперь с помощью Object.watch() вызвать функцию при изменении значения
// свойства "multiply". Обратите внимание: здесь выполняется явное
// преобразование типов аргументов
_root.watch("multiply", function() {
                _root.result = Number(_root.arg1) * Number(_root.arg2);
                // Возвращает значение, записанное в свойство.
                return 0;
            });
```

В качестве примера предположим, что Flash-плеер выполняет операцию умножения гораздо эффективнее, чем интерпретатор JavaScript. Тогда, вызвать ActionScript-код, чтобы умножить два числа, можно следующим образом:

```
/* JavaScript */
// Вызвать Flash-ролик для умножения двух чисел
function multiply(x, y) {
    var flash = window.movie || document.movie; // Получить Flash-объект
    flash.SetVariable("arg1", x);               // Установить первый аргумент
    flash.SetVariable("arg2", y);               // Установить второй аргумент
    flash.SetVariable("multiply", 1);           // Вызвать операцию умножения
    var result = flash.GetVariable("result");   // Получить результат
    return Number(result);                      // Преобразовать и вернуть его
}
```

23.4.4. Обращение к JavaScript–коду из Flash

Взаимодействие с JavaScript-сценарием из ActionScript выполняется с помощью функции `fscommand()`, которой передаются две строки:

```
fscommand("eval", "alert('Привет из Flash')");
```

Хотя аргументы функции `fscommand()` называются *command* (команда) и *args* (аргументы), они не являются представлениями команды и ее аргументов – это могут быть любые две строки.

Когда ActionScript-сценарий вызывает функцию `fscommand()`, обе строки передаются специальной JavaScript-функции, которая может выполнять любые действия в ответ на полученную команду *command*. Обратите внимание: значение, возвращаемое JavaScript-функцией, не передается обратно ActionScript-сценарию. Имя JavaScript-функции, вызываемой функцией `fscommand()`, зависит от значения атрибута `id` или `name` тега `<object>` или `<embed>`, в который встраивается Flash-ролик. Если предположить, что ролик получил имя «movie», тогда JavaScript-функция должна называться `movie_DoFSCommand`. Вот пример такой функции:

```
function movie_DoFSCommand(command, args) {
```

```
    if (command == "eval") eval(args);
}
```

Этот прием достаточно прост, но в Internet Explorer он не работает. По ряду причин, когда Flash-ролик встраивается в броузер IE, он не имеет возможности взаимодействовать с JavaScript-сценарием непосредственно – только на фирменном языке Microsoft под названием VBScript. VBScript, в свою очередь, может взаимодействовать с JavaScript. Таким образом, чтобы обеспечить корректную работу функции `fscommand()` в IE, необходимо включить в HTML-файл следующий фрагмент (где также предполагается, что ролик получил имя «movie»).

```
<script language="VBScript">
sub movie_FSCommand(byval command, byval args)
  call movie_DoFSCommand(command, args) ' Вызов JavaScript-функции
end sub
</script>
```

Не важно, понимаете ли вы смысл этого фрагмента, просто включите его в свой HTML-файл. Броузеры, которые не понимают язык VBScript, будут просто игнорировать этот тег `<script>` и все его содержимое.

23.4.5. Пример: из Flash в JavaScript и обратно

Теперь объединим все полученные сведения в одном примере, представленном двумя файлами: сценарием на языке ActionScript (пример 23.4) и HTML-файлом с JavaScript-сценарием (пример 23.5). После того как Flash-ролик загрузится, он с помощью функции `fscommand()` извещает об этом JavaScript-сценарий, который в ответ на это активизирует кнопку на форме. Если щелкнуть на этой кнопке, JavaScript-сценарий с помощью функции `SetVariable()` передаст Flash-ролику команду нарисовать прямоугольник. Кроме того, если щелкнуть мышью внутри области вывода Flash-ролика, с помощью функции `fscommand()` будет выведено сообщение с координатами указателя мыши. Оба примера прекрасно прокомментированы и не должны вызывать затруднений при их изучении.

Пример 23.4. ActionScript-сценарий, взаимодействующий с JavaScript-сценарием

```
/**
 * Box.as: ActionScript-сценарий для демонстрации взаимодействия
 *         между JavaScript и Flash
 *
 * Этот сценарий написан на языке ActionScript 2.0, который основан
 * на JavaScript, но имеет объектно-ориентированные расширения.
 * Сценарий определяет единственную статическую функцию main() в классе Box.
 *
 * С помощью свободно распространяемого компилятора ActionScript с названием
 * mtasc этот сценарий может быть скомпилирован следующей командой:
 *
 *  mtasc -header 300:300:1 -main -swf Box1.swf Box1.as
 *
 * Компилятор создает SWF-файл, который вызывает метод main() из первого кадра ролика.
 * Если вы пользуетесь Flash IDE, нужно вставить вызов метода Box.main() в первый кадр.
 */
```

```
class Box {
    static function main() {
        // Эта ActionScript-функция должна вызываться из JavaScript-сценария.
        // Она рисует прямоугольник и возвращает занимаемую им площадь.
        var drawBox = function(x,y,w,h) {
            _root.beginFill(0xaaaaaa, 100);
            _root.lineStyle(5, 0x000000, 100);
            _root.moveTo(x,y);
            _root.lineTo(x+w, y);
            _root.lineTo(x+w, y+h);
            _root.lineTo(x, y+h);
            _root.lineTo(x,y);
            _root.endFill();
            return w*h;
        }

        // Здесь выполняется настройка возможности вызова функции
        // из JavaScript для версий Flash ниже 8. Прежде всего, необходимо
        // определить свойства начала временной шкалы, в которых будут
        // храниться аргументы и возвращаемое значение.
        _root.arg1 = 0;
        _root.arg2 = 0;
        _root.arg3 = 0;
        _root.arg4 = 0;
        _root.result = 0;

        // Затем нужно объявить еще одно свойство с тем же именем,
        // что и функция.
        _root.drawBox = 0;

        // Далее с помощью метода Object.watch() следует установить
        // "контрольную точку" для отслеживания изменений значения этого
        // свойства. Когда в него произойдет запись, будет вызвана
        // указанная функция. Это означает, что JavaScript-сценарий
        // сможет инициировать вызов функции с помощью функции SetVariable.
        _root.watch("drawBox",   // Имя отслеживаемого свойства
                    function() { // Функция, которая вызывается при изменении
                        // Вызвать функцию drawBox(), преобразовать аргументы
                        // из строк в числа и сохранить возвращаемое значение.
                        _root.result = drawBox(Number(_root.arg1),
                                               Number(_root.arg2),
                                               Number(_root.arg3),
                                               Number(_root.arg4));

                        // Вернуть 0, чтобы значение отслеживаемого
                        // свойства не изменилось.
                        return 0;
                    });

        // Это ActionScript-обработчик события.
        // Он вызывает глобальную функцию fscommand(), которой
        // передаются координаты указателя мыши в момент щелчка.
        _root.onMouseDown = function() {
            fscommand("mousedown", _root._xmouse + "," + _root._ymouse);
        }
```

```
        // Здесь снова вызывается функция fscommand() и сообщает
        // JavaScript-сценарию, что Flash-ролик загружен и готов к работе.
        fscommand("loaded", "");
    }
}
```

Пример 23.5. Взаимодействие с Flash-роликом

```
<!--
    Это Flash-ролик, встроенный в веб-страницу.
    Следуя устоявшейся практике, в IE используется тег <object id="">,
    а в других броузерах - тег <embed name="">.
    Обратите внимание на условные комментарии, которые являются
    особенностью IE.
-->
<!--[if IE]>
<object id="movie" type="application/x-shockwave-flash"
        width="300" height="300">
  <param name="movie" value="Box1.swf">
</object>
<![endif]--><!--[if !IE]> <-->
<embed name="movie" type="application/x-shockwave-flash"
       src="Box1.swf" width="300" height="300">
</embed>
<!--> <![endif]-->

<!--
    В этой HTML-форме имеется кнопка, с помощью которой происходит
    манипулирование роликом или плеером.
    Обратите внимание: изначально кнопка Draw недоступна. Когда Flash-ролик
    загружается, он отправляет команду JavaScript-сценарию,
    а JavaScript-сценарий активизирует кнопку.
-->
<form name="f" onsubmit="return false;">
<button name="button" onclick="draw()" disabled>Draw</button>
<button onclick="zoom()">Zoom</button>
<button onclick="pan()">Pan</button>
</form>

<script>
// Эта функция демонстрирует порядок обращения к Flash-ролику
// универсальным способом.
function draw() {
    // Сначала нужно получить ссылку на Flash-объект. Поскольку в качестве
    // значения атрибутов id и name тегов <object> и <embed> использовалось
    // имя "movie", этот объект будет доступен как свойство с именем "movie".
    // В IE это свойство принадлежит объекту window, в других броузерах
    // - объекту document.
    var flash = window.movie || document.movie; // Получить ссылку на Flash-объект.

    // Прежде чем взаимодействовать с роликом, необходимо убедиться, что он
    // полностью загружен. Эта строка избыточна, поскольку кнопка, которая
    // вызывает метод, будет недоступна до тех пор, пока Flash не сообщит,
    // что ролик загружен
    if (flash.PercentLoaded() != 100) return;
```

```
    // Затем путем установки переменных функции "передаются" аргументы.
    flash.SetVariable("arg1", 10);
    flash.SetVariable("arg2", 10);
    flash.SetVariable("arg3", 50);
    flash.SetVariable("arg4", 50);

    // Теперь можно вызвать функцию, изменив значение еще одного свойства.
    flash.SetVariable("drawBox", 1);

    // В заключение запрашивается возвращаемое значение функции.
    return flash.GetVariable("result");
}

function zoom() {
    var flash = window.movie || document.movie; // Получить ссылку на Flash-объект.
    flash.Zoom(50);
}

function pan() {
    var flash = window.movie || document.movie; // Получить ссылку на Flash-объект.
    flash.Pan(-50, -50, 1);
}

// Эта функция вызывается, когда Flash обращается к функции fscommand().
// Строковые аргументы поставляются со стороны Flash.
// Эта функция должна иметь строго определенное имя, в противном случае
// она вызвана не будет. Имя функции начинается с "movie", потому что
// это значение атрибута id/name тега <object> или <embed>,
// использовавшегося ранее.
function movie_DoFSCommand(command, args) {
    if (command=="loaded") {
        // Когда Flash сообщит, что ролик загружен,
        // сложно активировать кнопку формы.
        document.f.button.disabled = false;
    }
    else if (command == "mousedown") {
        // Flash сообщит, когда пользователь щелкнет мышью.
        // Flash может передавать только строки. Мы можем анализировать их
        // по мере необходимости, чтобы получить данные, посылаемые Flash.
        var coords = args.split(",");
        alert("Mousedown: (" + coords[0] + ", " + coords[1] + ")");
    }

    // Несколько других интересных команд.
    else if (command == "debug") alert("Flash debug: " + args);
    else if (command == "eval") eval(args);
}
</script>

<script language="VBScript">
' Этот сценарий написан не на языке JavaScript, а на языке Visual Basic
' Scripting Edition компании Microsoft. Данный сценарий необходим, чтобы
' броузер Internet Explorer мог принимать извещения fscommand() от Flash.
' Он игнорируется всеми другими броузерами, которые не поддерживают VBScript.
' Имя этой подпрограммы должно быть в точности таким, как показано.
sub movie_FSCommand(byval command, byval args)
```

```
    call movie_DoFSCommand(command, args) ' простой вызов функции JavaScript
end sub
</script>
```

23.5. Сценарии во Flash 8

Во Flash 8 реализован класс с именем `ExternalInterface`, который существенным образом упрощает организацию взаимодействий JavaScript-сценариев и Flash. Класс `ExternalInterface` определяет статическую функцию `call()`, предназначенную для вызова именованных JavaScript-функций и получения возвращаемых значений. В этом классе так же определен статический метод `addCallback()`, выполняющий экспорт ActionScript-функций для их использования в JavaScript-сценариях. Описание класса `ExternalInterface` можно найти в четвертой части книги.

Чтобы продемонстрировать простоту взаимодействий с помощью класса `ExternalInterface`, преобразуем примеры 23.4 и 23.5. В примере 23.6 приводится видоизмененный ActionScript-сценарий, а в примере 23.7 – видоизмененный JavaScript-сценарий (определения тегов `<object>`, `<embed>` и `<form>` по сравнению с примером 23.5 не изменились и потому здесь они опущены).

В комментариях вы найдете все, что необходимо для понимания принципов использования класса `ExternalInterface`. Кроме того, метод `ExternalInterface.addCallback()` демонстрируется в примере 22.12.

Пример 23.6. ActionScript-сценарий, использующий класс ExternalInterface

```
/**
 * Box2.as: ActionScript-сценарий для демонстрации взаимодействия между
 * JavaScript и Flash с использованием класса ExternalInterface из Flash 8.
 *
 * Скомпилируйте этот сценарий с помощью следующей команды:
 *
 * mtasc -version 8 -header 300:300:1 -main -swf Box2.swf Box2.as
 *
 * Если вы пользуетесь Flash IDE, вставьте в первый кадр ролика вызов метода Box.main().
 */
import flash.external.ExternalInterface;
class Box {
    static function main() {
        // Экспорт ActionScript-функции с помощью класса ExternalInterface.
        // Это существенно упрощает вызов функции из JavaScript-сценария,
        // но поддерживается только в Flash 8 и более поздних версиях.
        // Первый аргумент addCallback - имя функции, под которым она будет
        // доступна в JavaScript-сценарии. Второй аргумент -
        // ActionScript-объект, в контексте которого будет вызвана функция,
        // значение этого аргумента станет значением ключевого слова 'this'.
        // В третьем аргументе передается ссылка на вызываемую функцию.
        ExternalInterface.addCallback("drawBox", null, function(x,y,w,h) {
            _root.beginFill(0xaaaaaa, 100);
            _root.lineStyle(5, 0x000000, 100);
            _root.moveTo(x,y);
            _root.lineTo(x+w, y);
```

```
            _root.lineTo(x+w, y+h);
            _root.lineTo(x, y+h);
            _root.lineTo(x,y);
            _root.endFill();
            return w*h;
        });

        // Это ActionScript-обработчик события.
        // Вызовом ExternalInterface.call() он сообщает
        // JavaScript-сценарию координаты мыши во время щелчка.
        _root.onMouseDown = function() {
            ExternalInterface.call("reportMouseClick",
                                   _root._xmouse, _root._ymouse);
        }

        // Сообщить JavaScript-сценарию, что ролик загружен полностью и готов к работе.
        ExternalInterface.call("flashReady");
    }
}
```

Пример 23.7. Упрощенный способ взаимодействия с Flash при помощи класса ExternalInterface

```
<script>
// Когда ActionScript-функция экспортируется функцией ExternalInterface.addCallback,
// к ней можно обращаться как к методу Flash-объекта.
function draw() {
    var flash = window.movie || document.movie; // Получить Flash-объект
    return flash.drawBox(100, 100, 100, 50);    // Вызвать функцию
}

// Эти функции будут вызываться из Flash с помощью ExternalInterface.call().
function flashReady() { document.f.button.disabled = false; }
function reportMouseClick(x, y) { alert("click: " + x + ", " + y); }
</script>
```

III

Справочник по базовому JavaScript

Эта часть книги представляет собой полный справочник по всем классам, свойствам, функциям и методам базового прикладного программного интерфейса JavaScript. На первых нескольких страницах рассказывается о том, как пользоваться справочником, поэтому им стоит уделить особое внимание.

В этой части описываются следующие классы и объекты:

`Arguments`	`Global`	`Number`
`Array`	`JavaArray`	`Object`
`Boolean`	`JavaClass`	`RegExp`
`Date`	`JavaObject`	`String`
`Error`	`JavaPackage`	
`Function`	`Math`	

Справочник по базовому JavaScript

Данная часть книги представляет собой справочник, в котором описаны классы, методы и свойства, составляющие основу JavaScript. Вводная часть и образец справочной статьи призваны помочь вам понять, как работать со справочником и извлечь из него максимум. Не поленитесь и внимательно прочитайте этот материал – вам будет легче искать и использовать нужную информацию!

Материал справочника организован в алфавитном порядке. Справочные статьи, посвященные методам и свойствам классов, отсортированы по их полным именам, включающим имена определяющих их классов. Так, чтобы найти метод `replace()` класса `String`, следует искать описание метода `String.replace()`, а не `replace`.

В базовом JavaScript определены некоторые глобальные функции и свойства, такие как `eval()` и `NaN`. Формально они представляют собой свойства глобального объекта. Однако у глобального объекта нет имени, поэтому в справочнике они перечислены по их неполным именам. Для удобства полный набор глобальных функций и свойств базового JavaScript объединен в специальную справочную статью «Global» (хотя объекта или класса с таким именем нет).

Найдя искомую справочную статью, вы без особого труда найдете и нужную информацию. Но вам будет легче работать со справочником, если вы разберетесь, как написаны и организованы справочные статьи. Далее, после заголовка «Пример справочной статьи», показана структура всех справочных статей и рассказано, какую информацию можно в этих статьях найти. Прежде чем искать что-то в справочнике, не пожалейте времени на ознакомление с этим примером.

Пример справочной статьи

Доступность

как читать справочные статьи по базовому JavaScript

наследует от

Заголовок и краткое описание

Каждая статья справочника начинается с приведенного выше блока заголовка, состоящего из четырех частей. Статьи отсортированы по заголовкам. Краткое описание под заголовком дает общее представление о предмете, описанном в данной статье, и позволяет быстро понять, интересует ли вас ее оставшаяся часть.

Доступность

Информация о доступности приведена в правом верхнем углу блока заголовка. В предыдущих изданиях книги эта информация сообщала о том, какие веб-броузеры поддерживали предмет описания. Ныне большинство броузеров поддерживают большую часть элементов, описываемых в книге, и потому в разделе, описывающем доступность, для большего удобства приводится информация о стандарте, представляющем формальную спецификацию описываемого элемента. Например, здесь вы

можете увидеть строку «ECMAScript v1» или «DOM уровня 2 HTML». Если предмет статьи признан устаревшим, это также будет отмечено.

Если описываемый элемент является ультрасовременной новинкой и еще не поддерживается большинством броузеров или относится к броузеру Internet Explorer, здесь иногда упоминаются названия броузеров и номера версий.

Если элемент не был стандартизован, как, например, объект `History`, но с ним способны работать броузеры, поддерживающие определенную версию JavaScript, то в этом разделе указывается также номер версии JavaScript. Например, объект `History` доступен в JavaScript 1.0.

Если статья, описывающая метод, не включает информации о доступности, это означает, что доступность метода совпадает с доступностью класса, в котором этот метод определен.

Наследует от

Если класс является наследником суперкласса или переопределяет метод суперкласса, эта информация отображается в правом нижнем углу блока заголовка.

В главе 9 говорилось, что JavaScript-классы могут наследовать свойства и методы от других классов. Например, класс `String` является наследником класса `Object`, а класс `HTMLDocument` – наследником класса `Document`, который в свою очередь является наследником класса `Node`. В статье с описанием класса `String` порядок наследования описывается так: «Object→String», а в статье с описанием `HTMLDocument`: «Node→Document→HTMLDocument». Когда вы увидите эту информацию, возможно, вам потребуется просмотреть разделы с описаниями перечисленных суперклассов.

Если метод имеет то же имя, что и метод суперкласса, он переопределяет метод суперкласса. В качестве примера загляните в статью с описанием метода `Array.toString()`.

Конструктор

Если статья справочника описывает класс и класс имеет конструктор, данный раздел описывает порядок использования метода-конструктора для создания экземпляров класса. Поскольку конструкторы являются разновидностью методов, раздел «Конструктор» во многом похож на раздел «Синтаксис» в справочной статье с описанием метода.

Синтаксис

В справочных статьях функций, методов и свойств есть раздел «Синтаксис», где демонстрируется порядок использования функции, метода или свойства в программе. В статьях справочника этой книги для различных методов применены два разных стиля описания синтаксиса. В статьях, описывающих элементы базового и клиентского JavaScript (таких как методы объекта `Window`), которые не связаны с моделью DOM, используется нетипизированный синтаксис. Например, синтаксис метода `Array.concat()`:

```
массив.concat(значение, ...)
```

Курсивом набран текст, который должен быть заменен чем-то другим. В данном случае *массив* должен быть заменен переменной, в которой содержится массив, или JavaScript-выражением, результатом выполнения которого является массив. А *значение* просто представляет собой произвольное значение, которое будет добавлено в массив. Многоточие (`...`) указывает, что данный метод может принимать любое число аргу-

ментов *значение*. Поскольку слово concat, а также открывающая и закрывающая скобки набраны не курсивом, они должны включаться в JavaScript-код точно так, как показано здесь.

Большинство методов, описываемых в четвертой части книги, стандартизованы консорциумом W3C и их спецификации включают информацию о типах аргументов методов и возвращаемых значений. В этом случае статьи включают информацию о типах в раздел с описанием синтаксиса. Например, синтаксис метода Document.getElementById() описывается следующим образом:

```
Element getElementById(String elementId);
```

Такое описание соответствует синтаксису языка Java, подчеркивая, что метод getElementById() возвращает объект Element и ожидает получить одну строку в виде аргумента с именем *elementId*. Поскольку это метод объекта Document и он неявно вызывается в контексте документа, то префикс document не включается в описание синтаксиса.

Аргументы

Если в справочной статье описывается функция, метод или класс с методом-конструктором, то вслед за разделами «Конструктор» и «Синтаксис» следует подраздел «Аргументы», в котором описаны аргументы метода, функции или конструктора. Если аргументы отсутствуют, этот подраздел опускается:

аргумент1

Здесь приводится описание перечисленных аргументов. Это, например, описание аргумента *аргумент1*.

аргумент2

Это описание аргумента *аргумент2*.

Возвращаемое значение

Если конструктор, функция или метод возвращает какое-либо значение, в данном подразделе приведено описание этого значения.

Исключения

Если конструктор, функция или метод может генерировать исключения, в этом подразделе перечислены типы возможных исключений и описаны обстоятельства, при которых они могут возникать.

Константы

Некоторые классы определяют набор констант, которые выступают в качестве значений свойств или аргументов методов. Например, интерфейс Node определяет важные константы, являющиеся допустимыми значениями свойства nodeType. Если интерфейс определяет константы, они перечисляются и описываются в этом разделе. Обратите внимание: константы – это статические свойства самого класса, а не экземпляра этого класса.

Свойства

Если справочная статья описывает класс, в разделе «Свойства» перечислены свойства, определенные в этом классе, и приведено краткое описание каждого из них. В части III книги для каждого свойства имеется также собственная полная справочная статья. В части IV большинство свойств описываются в этом разделе. Для наиболее важных или сложных свойств в части IV также приводится отдельная статья, и этот факт отмечается в данном разделе. В части IV книги в описания свойств DOM-классов

включена информация о типе. Свойства других классов и все свойства, перечисленные в части III, являются нетипизированными. Список свойств выглядит следующим образом:

`prop1`

Это краткое описание нетипизированного свойства `prop1`. В подзаголовке указывается только имя свойства, а в описание включается тип свойства, смысл или назначение независимо от того, доступно оно только для чтения или для чтения и записи.

`readonly integer prop2`

Это краткое описание типизированного свойства `prop2`. В подзаголовок, наряду с именем свойства, включается информация о его типе. Сам абзац с описанием рассказывает о назначении свойства.

Методы

Справочная статья о классе, определяющем методы, включает раздел «Методы». Он во многом похож на раздел «Свойства», за исключением того, что в нем перечислены не свойства, а методы. Для всех методов имеются также отдельные справочные статьи. В этом разделе перечислены лишь имена методов. Сведения о типах аргументов и возвращаемом значении вы найдете в статье с описанием самого метода.

Описание

Большинство справочных статей содержат раздел «Описание», являющийся основным описанием класса, метода, функции или свойства, которому посвящена статья. Это основная часть справочной статьи. Те, кто до сих пор ничего не знал о классе, методе или свойстве, могут сразу перейти к этому разделу, а затем вернуться и посмотреть предыдущие разделы, такие как «Аргументы», «Свойства» и «Методы». Тем же, кто знаком с классом, методом или свойством, возможно, читать этот раздел не потребуется, поскольку им достаточно лишь быстро найти определенную информацию о нем (например, из раздела «Аргументы» или «Свойства»).

В некоторых статьях этот раздел представляет собой лишь короткий абзац. В других он может занимать одну или несколько страниц. Для некоторых простых методов разделы «Аргументы» и «Возвращаемое значение» сами достаточно хорошо описывают метод, тогда раздел «Описание» опускается.

Пример

Некоторые справочные статьи включают пример, иллюстрирующий типичное использование. Большинство статей, однако, не содержат примеров. Вы найдете примеры в первой половине этой книги.

Ошибки

Если предмет справочной статьи работает не совсем правильно, то в данном разделе приводится описание ошибок. Однако заметим, что в данной книге не ставилась задача учесть все ошибки во всех версиях и реализациях JavaScript.

См. также

Многие справочные статьи заканчиваются перекрестными ссылками на близкие к теме справочные статьи, которые могут представлять интерес. Иногда справочные статьи ссылаются на одну из глав книги.

arguments[] ECMAScript v1

массив аргументов функции

Синтаксис

```
arguments
```

Описание

Массив `arguments[]` определен только внутри тела функции, где он ссылается на объект `Arguments` этой функции. Данный объект имеет нумерованные свойства и представляет собой массив, содержащий все переданные функции аргументы. Идентификатор `arguments` – это, по существу, локальная переменная, автоматически объявляемая и инициализируемая внутри каждой функции. Она ссылается на объект `Arguments` только внутри тела функции и не определена в глобальном коде.

См. также `Arguments`; глава 8

Arguments ECMAScript v1

аргументы и другие свойства функции **Object→Arguments**

Синтаксис

```
arguments
arguments[n]
```

Элементы

Объект `Arguments` определен только внутри тела функции. Хотя формально он не является массивом, у него есть нумерованные свойства, работающие как элементы массива, и свойство `length`, равное количеству элементов массива. Его элементами являются значения, переданные функции в качестве аргументов. Элемент 0 – это первый аргумент, элемент 1 – второй аргумент и т. д. Все значения, переданные в качестве аргументов, становятся элементами массива в объекте `Arguments` независимо от того, присвоены ли этим аргументам имена в объявлении функции.

Свойства

`callee`

Ссылка на исполняемую в данный момент функцию.

`length`

Количество аргументов, переданных функции, и количество элементов массива в объекте `Arguments`.

Описание

Когда вызывается функция, для нее создается объект `Arguments`, и локальная переменная `arguments` автоматически инициализируется ссылкой на объект `Arguments`. Основное назначение объекта `Arguments` – предоставить возможность определить, сколько аргументов передано функции, и обратиться к неименованным аргументам. В дополнение к элементам массива и свойству `length`, свойство `callee` позволяет неименованной функции ссылаться на саму себя.

Для большинства задач объект Arguments можно рассматривать как массив с дополнительным свойством callee. Однако он не является экземпляром объекта Array, а свойство Arguments.length не ведет себя особым образом, как свойство Array.length, и не может использоваться для изменения размера массива.

Объект Arguments имеет одну очень необычную особенность. Когда у функции есть именованные аргументы, элементы массива в объекте Arguments являются синонимами локальных переменных, содержащих аргументы функции. Объект Arguments и имена аргументов предоставляют два различных способа обращения к одной и той же переменной. Изменение значения аргумента с помощью имени аргумента изменяет значение, извлекаемое через объект Arguments, а изменение значения аргумента через объект Arguments изменяет значение, извлекаемое по имени аргумента.

См. также Function; глава 8

Arguments.callee ECMAScript v1

функция, исполняющаяся в данный момент

Синтаксис

```
arguments.callee
```

Описание

arguments.callee ссылается на функцию, работающую в данный момент. Данный синтаксис предоставляет неименованной функции возможность сослаться на себя. Это свойство определено только внутри тела функции.

Пример

```
// Неименованный функциональный литерал использует свойство callee
// для ссылки на себя, чтобы произвести рекурсивный вызов
var factorial = function(x) {
    if (x < 2) return 1;
    else return x * arguments.callee(x1);
}
var y = factorial(5); // Возвращает 120
```

Arguments.length ECMAScript v1

число аргументов, переданных функции

Синтаксис

```
arguments.length
```

Описание

Свойство length объекта Arguments возвращает количество аргументов, переданных текущей функции. Это свойство определено только внутри тела функции.

Обратите внимание: это свойство возвращает фактическое количество переданных аргументов, а не ожидаемое. О количестве аргументов в объявлении функции говорится в статье о свойстве Function.length. Кроме того, следует отметить, что это свойство не ведет себя особым образом, как свойство Array.length.

Пример

```
// Используем объект Arguments, чтобы проверить, верное ли количество аргументов было передано
function check(args) {
    var actual = args.length;          // Фактическое количество аргументов
    var expected = args.callee.length; // Ожидаемое количество аргументов
    if (actual != expected) {          // Если они не совпадают, генерируем исключение
        throw new Error("Неверное число аргументов: ожидается: " +
                        expected + "; фактически передано " + actual);
    }
}
// Функция, демонстрирующая использование функции, приведенной ранее
function f(x, y, z) {
    check(arguments); // Проверить правильность количества аргументов
    return x + y + z; // Теперь выполнить оставшуюся часть функции обычным образом
}
```

См. также `Array.length`, `Function.length`

Array

ECMAScript v1

встроенная поддержка массивов **Object→Array**

Конструктор

`new Array()`

`new Array(`*размер*`)`

`new Array(`*элемент0, элемент1, ..., элементn*`)`

Аргументы

размер

Желаемое количество элементов в массиве. Длина возвращаемого массива (`length`) равна аргументу *размер*.

элемент0, ... элементn

Список аргументов из двух и более произвольных значений. Когда конструктор `Array()` вызывается с этими аргументами, элементы только что созданного массива инициализируются указанными значениями, а свойство `length` становится равным количеству аргументов.

Возвращаемое значение

Вновь созданный и инициализированный массив. Когда конструктор `Array()` вызывается без аргументов, он возвращает пустой массив и значение свойства `length` равно 0. При вызове с одним числовым аргументом конструктор возвращает массив с указанным количеством неопределенных элементов. При вызове с любыми другими аргументами конструктор инициализирует массив значениями, указанными его аргументами. Когда конструктор `Array()` вызывается как функция (без оператора `new`), он ведет себя точно так же, как при вызове с оператором `new`.

Исключения

`RangeError`

Когда конструктору `Array()` передается один целый аргумент *размер*, генерируется исключение `RangeError`, если размер отрицателен или превышает $2^{32}-1$.

Синтаксис литерала

ECMAScript v3 определяет синтаксис литералов для массивов. Программист может создавать и инициализировать массив, заключая список выражений, перечисленных через запятые, в квадратные скобки. Значения этих выражений становятся элементами массива. Например:

```
var a = [1, true, 'abc'];
var b = [a[0], a[0]*2, f(x)];
```

Свойства

`length`

Целое, доступное для чтения и записи, задает количество элементов массива, или, если элементы массива расположены не непрерывно, число, на единицу большее индекса последнего элемента массива. Изменение этого свойства укорачивает или расширяет массив.

Методы

`concat()`

Присоединяет элементы к массиву.

`join()`

Преобразует все элемента массива в строки и выполняет их конкатенацию.

`pop()`

Удаляет элемент из конца массива.

`push()`

Помещает элемент в конец массива.

`reverse()`

Меняет порядок следования элементов массива на противоположный.

`shift()`

Сдвигает элементы к началу массива.

`slice()`

Возвращает подмассив массива.

`sort()`

Сортирует элементы массива.

`splice()`

Вставляет, удаляет или заменяет элементы массива.

`toLocaleString()`

Преобразует массив в локализованную строку.

`toString()`

Преобразует массив в строку.

`unshift()`

Вставляет элементы в начало массива.

Описание

Массивы – это базовое средство JavaScript, подробно описанное в главе 7.

См. также

Глава 7

Array.concat() ECMAScript v3

выполняет конкатенацию массивов

Синтаксис

массив.concat(*значение*, ...)

Аргументы

значение, ...

Любое количество значений, присоединяемых к *массиву*.

Возвращаемое значение

Новый массив, образуемый присоединением к массиву каждого из указанных аргументов.

Описание

Метод concat() создает и возвращает новый массив, являющийся результатом присоединения каждого из его аргументов к *массиву*. Этот метод не изменяет *массив*. Если какие-либо из аргументов concat() сами являются массивами, то присоединяются элементы этих массивов, а не сами массивы.

Пример

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

См. также Array.join(), Array.push(), Array.splice()

Array.join() ECMAScript v1

выполняет конкатенацию элементов массива в строку

Синтаксис

массив.join()

массив.join(*разделитель*)

Аргументы

разделитель

Необязательный символ или строка, выступающая в качестве разделителя элементов в результирующей строке. Если аргумент опущен, используется запятая.

Возвращаемое значение

Строка, получающаяся в результате преобразования каждого элемента *массива* в строку и объединения их с *разделителем* между элементами путем конкатенации.

Описание

Метод join() преобразует каждый из элементов массива в строку и затем выполняет конкатенацию этих строк, вставляя указанный *разделитель* между элементами. Метод возвращает результирующую строку.

Обратное преобразование (разбиение строки на элементы массива) можно выполнить с помощью метода `split()` объекта `String`. Подробности см. в справочной статье, посвященной методу `String.split()`.

Пример

```
a = new Array(1, 2, 3, "testing");
s = a.join("+"); // s - это строка "1+2+3+testing"
```

См. также `String.split()`

Array.length

ECMAScript v1

размер массива

Синтаксис

массив`.length`

Описание

Свойство `length` массива всегда на единицу больше индекса последнего элемента, определенного в массиве. Для традиционных «плотных» массивов, в которых определена непрерывная последовательность элементов и которые начинаются с элемента 0, свойство `length` указывает количество элементов в массиве.

Свойство `length` инициализируется в момент создания массива с помощью метода-конструктора `Array()`. Добавление новых элементов изменяет значение `length`, если в этом возникает необходимость:

```
a = new Array();                    // a.length равно 0
b = new Array(10);                  // b.length равно 10
c = new Array("one", "two", "three"); // c.length равно 3
c[3] = "four";                      // c.length изменяется на 4
c[10] = "blastoff";                 // c.length становится равным 11
```

Чтобы изменить размер массива, можно установить значение свойства `length`. Если новое значение `length` меньше предыдущего, массив обрезается, и элементы в его конце теряются. Если значение `length` увеличивается (новое значение больше старого), массив становится больше, а новые элементы, добавленные в конец массива, получают значение `undefined`.

Array.pop()

ECMAScript v3

удаляет и возвращает последний элемент массива

Синтаксис

массив`.pop()`

Возвращаемое значение

Последний элемент *массива*.

Описание

Метод `pop()` удаляет последний элемент массива, уменьшает длину массива на единицу и возвращает значение удаленного элемента. Если массив уже пуст, `pop()` его не изменяет и возвращает значение `undefined`.

Пример

Метод pop() и парный ему метод push() позволяют реализовать стек, работающий по принципу «первым вошел, последним вышел». Например:

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

См. также Array.push()

Array.push() ECMAScript v3

добавляет элементы массива

Синтаксис

массив.push(*значение*, ...)

Аргументы

значение, ...

Одно или более значений, которые должны быть добавлены в конец массива.

Возвращаемое значение

Новая длина массива после добавления в него указанных значений.

Описание

Метод push() добавляет свои аргументы в указанном порядке в конец *массива*. Он изменяет существующий *массив*, а не создает новый. push() и парный ему метод pop() используют массив для реализации стека, работающего по принципу «первым вошел, последним вышел». Пример – в статье Array.pop().

См. также Array.pop()

Array.reverse() ECMAScript v1

изменяет порядок следования элементов в массиве на противоположный

Синтаксис

массив.reverse()

Описание

Метод reverse() объекта Array меняет порядок следования элементов в массиве на противоположный. Он делает это «на месте», т. е. переупорядочивает элементы указанного *массива*, не создавая новый. Если есть несколько ссылок на *массив*, новый порядок следования элементов массива будет виден через все ссылки.

Пример

```
a = new Array(1, 2, 3); // a[0] == 1, a[2] == 3;
a.reverse();            // Теперь a[0] == 3, a[2] == 1;
```

Array.shift() ECMAScript v3

сдвигает элементы к началу массива

Синтаксис

массив.shift()

Возвращаемое значение

Бывший первый элемент массива.

Описание

Метод shift() удаляет и возвращает первый элемент *массива*, смещая все последующие элементы на одну позицию вниз для занятия освободившегося места в начале массива. Если массив пуст, shift() не делает ничего и возвращает значение undefined. Обратите внимание: shift() не создает новый массив, а непосредственно изменяет сам *массив*.

Метод shift() похож на Array.pop() за исключением того, что удаление элемента производится из начала массива, а не с конца. shift() часто используется в сочетании с unshift().

Пример

```
var a = [1, [2,3], 4]
a.shift(); // Возвращает 1; a = [[2,3], 4]
a.shift(); // Возвращает [2,3]; a = [4]
```

См. также Array.pop(), Array.unshift()

Array.slice() ECMAScript v3

возвращает фрагмент массива

Синтаксис

массив.slice(*начало, конец*)

Аргументы

начало

Индекс элемента массива, с которого начинается фрагмент. Отрицательное значение этого аргумента указывает позицию, измеряемую от конца массива. Другими словами, −1 обозначает последний элемент, −2 − второй элемент с конца и т. д.

конец

Индекс элемента массива, расположенного непосредственно после конца фрагмента. Если этот аргумент не указан, фрагмент включает все элементы массива от элемента, заданного аргументом *начало*, до конца массива. Если этот аргумент отрицателен, позиция элемента отсчитывается от конца массива.

Возвращаемое значение

Новый массив, содержащий элементы *массива* от элемента, заданного аргументом *начало*, до элемента, определяемого аргументом *конец*, но не включая его.

Описание

Метод slice() возвращает фрагмент, или подмассив, массива. Возвращаемый массив содержит элемент, заданный аргументом *начало*, и все последующие элементы до эле-

мента, заданного аргументом *конец*, но не включая его. Если аргумент *конец* не указан, возвращаемый массив содержит все элементы от элемента, заданного аргументом *начало*, до конца массива.

Обратите внимание: `slice()` не изменяет массив. Для удаления фрагмента массива следует использовать метод `Array.splice()`.

Пример

```
var a = [1,2,3,4,5];
a.slice(0,3);    // Возвращает [1,2,3]
a.slice(3);      // Возвращает [4,5]
a.slice(1,-1);   // Возвращает [2,3,4]
a.slice(-3,-2); // Возвращает [3]; в IE 4 работает с ошибкой, возвращая [1,2,3]
```

Ошибки

В Internet Explorer 4 *начало* не может быть отрицательным числом. В более поздних версиях IE эта ошибка исправлена.

См. также

`Array.splice()`

Array.sort() ECMAScript v1

сортирует элементы массива

Синтаксис

массив`.sort()`

массив`.sort(`*orderfunc*`)`

Аргументы

orderfunc

Необязательная функция, определяющая порядок сортировки.

Возвращаемое значение

Ссылка на массив. Обратите внимание: массив сортируется на месте, копия массива не делается.

Описание

Метод `sort()` сортирует элементы массива на месте без создания копии массива. Если `sort()` вызывается без аргументов, элементы массива располагаются в алфавитном порядке (точнее, в порядке, определяемом используемой в системе кодировкой символов). Если необходимо, элементы сначала преобразуются в строки, чтобы их можно было сравнивать.

Чтобы отсортировать элементы массива в каком-либо другом порядке, необходимо указать функцию сравнения, которая сравнивает два значения и возвращает число, обозначающее их относительный порядок. Функция сравнения должна принимать два аргумента, *a* и *b*, и возвращать одно из следующих значений:

- Отрицательное число, если в соответствии с выбранным критерием сортировки значение *a* «меньше» значения *b* и должно находиться в отсортированном массиве перед *b*.
- Ноль, если *a* и *b* в смысле сортировки эквивалентны.
- Положительное число, если значение *a* «больше» значения *b*.

Следует отметить, что неопределенные элементы при сортировке всегда оказываются в конце массива. Это происходит, даже если указана специальная функция сортировки: неопределенные значения никогда не передаются в заданную функцию *orderfunc*.

Пример

Следующий фрагмент показывает, как написать функцию сравнения, сортирующую массив чисел в числовом, а не в алфавитном порядке:

```
// Функция сортировки чисел в порядке возрастания
function numberorder(a, b) { return a - b; }
a = new Array(33, 4, 1111, 222);
a.sort();            // Алфавитная сортировка: 1111, 222, 33, 4
a.sort(numberorder); // Числовая сортировка: 4, 33, 222, 1111
```

Array.splice() ECMAScript v3

вставляет, удаляет или замещает элементы массива

Синтаксис

массив.splice(*начало, удаляемое_количество, значение, ...*)

Аргументы

начало

Элемент массива, с которого следует начать вставку или удаление.

удаляемое_количество

Количество элементов, которые должны быть удалены из *массива*, начиная с элемента, заданного аргументом *начало*, и включая этот элемент. Этот аргумент не является обязательным. Если он не указан, splice() удаляет все элементы от позиции, заданной аргументом *начало*, до конца массива.

значение, ...

Ноль или более значений, которые должны быть вставлены в массив, начиная с индекса, указанного в аргументе *начало*.

Возвращаемое значение

Массив, содержащий удаленные из массива элементы, если они есть.

Описание

Метод splice() удаляет указанное количество элементов массива, начиная с элемента, позиция которого определяется аргументом *начало*, включая его, и заменяет значениями, перечисленными в списке аргументов. Элементы массива, расположенные после вставляемых или удаляемых элементов, сдвигаются и образуют непрерывную последовательность с остальной частью массива. Однако следует заметить, что в отличие от метода с похожим именем, slice(), метод splice() непосредственно изменяет *массив*.

Пример

Работу splice() проще всего понять на примере:

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

```
a.splice(1,1);     // Возвращает [4]; a равно [1]
a.splice(1,0,2,3); // Возвращает []; a равно [1 2 3]
```

См. также Array.slice()

Array.toLocaleString() ECMAScript v1

преобразует массив в локализованную строку **переопределяет Object.toLocaleString()**

Синтаксис

массив.toLocaleString()

Возвращаемое значение

Локализованное строковое представление массива.

Исключения

TypeError

Если метод вызывается для объекта, не являющегося массивом.

Описание

Метод toLocaleString() массива возвращает локализованное строковое представление массива. Это делается путем вызова метода toLocaleString() для всех элементов массива и последующей конкатенации полученных строк с использованием символа-разделителя, определяемого региональными параметрами.

См. также Array.toString(), Object.toLocaleString()

Array.toString() ECMAScript v1

преобразует массив в строку **переопределяет Object.toString()**

Синтаксис

массив.toString()

Возвращаемое значение

Строковое представление *массива*.

Исключения

TypeError

Если метод вызывается для объекта, не являющегося массивом.

Описание

Метод toString() массива преобразует массив в строку и возвращает эту строку. Когда массив используется в строковом контексте, JavaScript автоматически преобразует его в строку путем вызова этого метода. Однако в некоторых случаях может потребоваться явный вызов toString().

toString() сначала преобразует в строку каждый элемент (вызывая их методы toString()). После преобразования все элементы выводятся в виде списка строк, разделенных запятыми. Это значение совпадает со значением, возвращаемым методом join() без аргументов.

См. также Array.toLocaleString(), Object.toString()

Array.unshift() ECMAScript v3

вставляет элементы в начало массива

Синтаксис

массив.unshift(*значение*, ...)

Аргументы

значение, ...

Одно и более значений, которые должны быть вставлены в начало *массива*.

Возвращаемое значение

Новая длина массива.

Описание

Метод unshift() вставляет свои аргументы в начало массива, сдвигая существующие элементы к верхним индексам для освобождения места. Первый аргумент shift() становится новым нулевым элементом массива, второй аргумент – новым первым элементом и т. д. Обратите внимание: unshift() не создает новый массив, а изменяет существующий.

Пример

unshift() часто используется совместно с shift(). Например:

```
var a = [];           // a:[]
a.unshift(1);         // a:[1]          Возвращает: 1
a.unshift(22);        // a:[22,1]       Возвращает: 2
a.shift();            // a:[1]          Возвращает: 22
a.unshift(33,[4,5]); // a:[33,[4,5],1] Возвращает: 3
```

См. также

Array.shift()

Boolean ECMAScript v1

поддержка логических значений **Object→Boolean**

Конструктор

new Boolean(*значение*) // Функция-конструктор

Boolean(*значение*) // Функция преобразования

Аргументы

значение

Значение, которое должно быть сохранено в объекте Boolean или преобразовано в логическое значение.

Возвращаемое значение

При вызове в качестве конструктора (с оператором new) Boolean() преобразует аргумент в логическое значение и возвращает объект Boolean, содержащий это значение. При вызове в качестве функции (без оператора new) Boolean() просто преобразует свой аргумент в элементарное логическое значение и возвращает это значение.

Значения 0, NaN, null, пустая строка "" и значение undefined преобразуются в false. Все остальные элементарные значения, за исключением false (но включая строку "false"), а также все объекты и массивы преобразуются в true.

Методы

toString()

Возвращает "true" или "false" в зависимости от логического значения, представляемого объектом Boolean.

valueOf()

Возвращает элементарное логическое значение, содержащееся в объекте Boolean.

Описание

Логические значения – это базовый тип данных JavaScript. Объект Boolean представляет собой «обертку» вокруг логического значения. Объектный тип Boolean в основном существует для предоставления метода toString(), который преобразует логические значения в строки. Когда метод toString() вызывается для преобразования логического значения в строку (а он часто вызывается JavaScript неявно), JavaScript преобразует логическое значение во временный объект Boolean, для которого может быть вызван метод toString().

См. также

Object

Boolean.toString() ECMAScript v1

преобразует логическое значение в строку **переопределяет Object.toString()**

Синтаксис

b.toString()

Возвращаемое значение

Строка "true" или "false" в зависимости от того, чем является *b*: элементарным логическим значением или объектом Boolean.

Исключения

TypeError

Если метод вызывается для объекта, не являющегося объектом Boolean.

Boolean.valueOf() ECMAScript v1

логическое значение объекта Boolean **переопределяет Object.valueOf()**

Синтаксис

b.valueOf()

Возвращаемое значение

Элементарное логическое значение, которое содержится в *b*, являющимся объектом Boolean.

Исключения

TypeError

Если метод вызывается для объекта, не являющегося Boolean.

Date

ECMAScript v1

работа с датами и временем **Object→Date**

Конструктор

```
new Date()
new Date(миллисекунды)
new Date(строка_даты)
new Date(год, месяц, день, часы, минуты, секунды, мс)
```

Конструктор `Date()` без аргументов создает объект `Date` со значением, равным текущим дате и времени. Если конструктору передается единственный числовой аргумент, он используется как внутреннее числовое представление даты в миллисекундах, аналогичное значению, возвращаемому методом `getTime()`. Когда передается один строковый аргумент, он рассматривается как строковое представление даты в формате, принимаемом методом `Date.parse()`. Кроме того, конструктору можно передать от двух до семи числовых аргументов, задающих индивидуальные поля даты и времени. Все аргументы, кроме первых двух – полей года и месяца, – могут отсутствовать. Обратите внимание: эти поля даты и времени задаются на основе локального времени, а не времени UTC (Universal Coordinated Time – универсальное скоординированное время), аналогичного GMT (Greenwich Mean Time – среднее время по Гринвичу). В качестве альтернативы может использоваться статический метод `Date.UTC()`.

`Date()` может также вызываться как функция (без оператора `new`). При таком вызове `Date()` игнорирует любые переданные аргументы и возвращает текущие дату и время.

Аргументы

миллисекунды

Количество миллисекунд между нужной датой и полночью 1 января 1970 года (UTC). Например, передав в качестве аргумента число 5000, мы создадим дату, обозначающую пять секунд после полуночи 1 января 1970 года.

строка_даты

Единственный аргумент, задающий дату и (необязательно) время в виде строки. Строка должна иметь формат, понятный для `Date.parse()`.

год

Год в виде четырех цифр. Например, 2001 для 2001 года. Для совместимости с более ранними реализациями JavaScript к аргументу добавляется 1900, если значение аргумента находится между 0 и 99.

месяц

Месяц, заданный в виде целого от 0 (январь) до 11 (декабрь).

день

День месяца, заданный в виде целого от 1 до 31. Обратите внимание, что наименьшее из значений этого аргумента равно 1, а остальных аргументов – 0. Необязательный аргумент.

часы

Часы, заданные в виде целого от 0 (полночь) до 23 (11 часов вечера). Необязательный аргумент.

минуты

Минуты в часах, указанные в виде целого от 0 до 59. Необязательный аргумент.

секунды

Секунды в минутах, указанные в виде целого от 0 до 59. Необязательный аргумент.

мс

Миллисекунды в секунде, указанные в виде целого от 0 до 999. Необязательный аргумент.

Методы

У объекта `Date` нет доступных для записи или чтения свойств; вместо этого доступ к значениям даты и времени выполняется через методы. Большинство методов объекта `Date` имеют две формы: одна для работы с локальным временем, другая – с универсальным временем (UTC или GMT). Если в имени метода присутствует строка `"UTC"`, он работает с универсальным временем. Эти пары методов указываются в приведенном далее списке вместе. Например, обозначение `get[UTC]Day()` относится к двум методам: `getDay()` и `getUTCDay()`.

Методы объекта `Date` могут вызываться только для объектов типа `Date` и генерируют исключение `TypeError`, если вызывать их для объектов другого типа.

`get[UTC]Date()`

Возвращает день месяца из объекта `Date` в соответствии с локальным или универсальным временем.

`get[UTC]Day()`

Возвращает день недели из объекта `Date` в соответствии с локальным или универсальным временем.

`get[UTC]FullYear()`

Возвращает год даты в полном четырехзначном формате в локальном или универсальном времени.

`get[UTC]Hours()`

Возвращает поле часов в объекте `Date` в локальном или универсальном времени.

`get[UTC]Milliseconds()`

Возвращает поле миллисекунд в объекте `Date` в локальном или универсальном времени.

`get[UTC]Minutes()`

Возвращает поле минут в объекте `Date` в локальном или универсальном времени.

`get[UTC]Month()`

Возвращает поле месяца в объекте `Date` в локальном или универсальном времени.

`get[UTC]Seconds()`

Возвращает поле секунд в объекте `Date` в локальном или универсальном времени.

`getTime()`

Возвращает внутреннее представление (миллисекунды) объекта `Date`. Обратите внимание: это значение не зависит от часового пояса, следовательно, отдельный метод `getUTCTime()` не нужен.

`getTimezoneOffset()`

Возвращает разницу в минутах между локальным и универсальным представлениями даты. Обратите внимание: возвращаемое значение зависит от того, действует ли для указанной даты летнее время.

`getYear()`

Возвращает поле года в объекте `Date`. Признан устаревшим, рекомендуется вместо него применять метод `getFullYear()`.

`set[UTC]Date()`

Устанавливает день месяца в `Date` в соответствии с локальным или универсальным временем.

`set[UTC]FullYear()`

Устанавливает год (и, возможно, месяц и день) в `Date` в соответствии с локальным или универсальным временем.

`set[UTC]Hours()`

Устанавливает час (и, возможно, поля минут, секунд и миллисекунд) в `Date` в соответствии с локальным или универсальным временем.

`set[UTC]Milliseconds()`

Устанавливает поле миллисекунд в `Date` в соответствии с локальным или универсальным временем.

`set[UTC]Minutes()`

Устанавливает поле минут (и, возможно, поля секунд и миллисекунд) в `Date` в соответствии с локальным или универсальным временем.

`set[UTC]Month()`

Устанавливает поле месяца (и, возможно, дня месяца) в `Date` в соответствии с локальным или универсальным временем.

`set[UTC]Seconds()`

Устанавливает поле секунд (и, возможно, поле миллисекунд) в `Date` в соответствии с локальным или универсальным временем.

`setTime()`

Устанавливает поля объекта `Date` в соответствии с миллисекундным форматом.

`setYear()`

Устанавливает поле года объекта `Date`. Признан устаревшим, вместо него рекомендуется использовать `setFullYear()`.

`toDateString()`

Возвращает строку, представляющую дату из `Date` для локального часового пояса.

`toGMTString()`

Преобразует `Date` в строку, беря за основу часовой пояс GMT. Признан устаревшим, вместо него рекомендован метод `toUTCString()`.

`toLocaleDateString()`

Возвращает строку, представляющую дату из `Date` в локальном часовом поясе в соответствии с локальными соглашениями по форматированию дат.

`toLocaleString()`

Преобразует `Date` в строку в соответствии с локальным часовым поясом и локальными соглашениями о форматировании дат.

`toLocaleTimeString()`

Возвращает строку, представляющую время из `Date` в локальном часовом поясе на основе локальных соглашений о форматировании времени.

`toString()`

Преобразует `Date` в строку в соответствии с локальным часовым поясом.

`toTimeString()`

Возвращает строку, представляющую время из `Date` в локальном часовом поясе.

`toUTCString()`

Преобразует `Date` в строку, используя универсальное время.

`valueOf()`

Преобразует объект `Date` в его внутренний миллисекундный формат.

Статические методы

В дополнение к перечисленным методам экземпляра в объекте `Date` определены два статических метода. Эти методы вызываются через сам конструктор `Date()`, а не через отдельные объекты `Date`:

`Date.parse()`

Анализирует строковое представление даты и времени и возвращает внутреннее представление этой даты в миллисекундах.

`Date.UTC()`

Возвращает миллисекундное представление указанной даты и времени UTC.

Описание

Объект `Date` – это тип данных, встроенный в язык JavaScript. Объекты `Date` создаются с помощью представленного ранее синтаксиса `new Date()`.

После создания объекта `Date` можно воспользоваться его многочисленными методами. Многие из методов позволяют получать и устанавливать поля года, месяца, дня, часа, минуты, секунды и миллисекунды в соответствии либо с локальным временем, либо с временем UTC (универсальным, или GMT). Метод `toString()` и его варианты преобразуют даты в понятные для восприятия строки. `getTime()` и `setTime()` преобразуют количество миллисекунд, прошедших с полуночи (GMT) 1 января 1970 года, во внутреннее представление объекта `Date` и обратно. В этом стандартном миллисекундном формате дата и время представляются одним целым, что делает дату очень простой арифметически. Стандарт ECMAScript требует, чтобы объект `Date` мог представить любые дату и время с миллисекундной точностью в пределах 100 миллионов дней до и после 01.01.1970. Этот диапазон равен ±273 785 лет, поэтому JavaScript-часы будут правильно работать до 275 755 года.

Пример

Известно множество методов, позволяющих работать с созданным объектом `Date`:

```
d = new Date(); // Получает текущую дату и время
document.write('Сегодня: '+d.toLocaleDateString() + '. ');// Показывает дату
document.write('Время: '+ d.toLocaleTimeString());        // Показывает время
var dayOfWeek = d.getDay();                               // День недели
var weekend = (dayOfWeek == 0) || (dayOfWeek == 6);       // Сегодня выходной?
```

Еще одно обычное применение объекта `Date` – это вычитание миллисекундного представления текущего времени из другого времени для определения относительного местоположения двух временных меток. Следующий пример клиентского кода показывает два таких применения:

```
<script language="JavaScript">
today = new Date();      // Запоминаем сегодняшнюю дату
christmas = new Date(); // Получаем дату из текущего года
christmas.setMonth(11); // Устанавливаем месяц декабрь...
christmas.setDate(25);  // и 25-е число
// Если Рождество еще не прошло, вычисляем количество миллисекунд между текущим моментом
// и Рождеством, преобразуем его в количество дней и печатаем сообщение
if (today.getTime() < christmas.getTime()) {
    difference = christmas.getTime() - today.getTime();
    difference = Math.floor(difference / (1000 * 60 * 60 * 24));
    document.write('Всего ' + difference + ' дней до Рождества!<p>');
}
</script>
// ... остальная часть HTML-документа ...
<script language="JavaScript">
// Здесь мы используем объекты Date для измерения времени
// Делим на 1000 для преобразования миллисекунд в секунды
now = new Date();
document.write('<p>Страница загружалась' +
               (now.getTime()-today.getTime())/1000 +
               'секунд.');
</script>
```

См. также `Date.parse()`, `Date.UTC()`

Date.getDate() ECMAScript v1

возвращает день месяца

Синтаксис

дата`.getDate()`

Возвращаемое значение

День месяца в указанном аргументе *дата*, представляющем собой объект `Date`, в соответствии с локальным временем. Возвращаемые значения могут находиться в интервале между 1 и 31.

Date.getDay() ECMAScript v1

возвращает день недели

Синтаксис

дата`.getDay()`

Возвращаемое значение

День недели в указанном аргументе *дата*, представляющем собой объект `Date`, в соответствии с локальным временем. Возвращает числа от 0 (воскресенье) до 6 (суббота).

Date.getFullYear() ECMAScript v1

возвращает год

Синтаксис

дата`.getFullYear()`

Возвращаемое значение

Год, получаемый, когда *дата* выражена в локальном времени. Возвращает четыре цифры, а не сокращение из двух цифр.

Date.getHours() ECMAScript v1

возвращает значение поля часов объекта Date

Синтаксис

дата.getHours()

Возвращаемое значение

Значение поля часов в аргументе *дата*, представляющем собой объект Date, в локальном времени. Возвращаемое значение находится в диапазоне между 0 (полночь) и 23 (11 часов вечера).

Date.getMilliseconds() ECMAScript v1

возвращает значение поля миллисекунд объекта Date

Синтаксис

дата.getMilliseconds()

Возвращаемое значение

Поле миллисекунд в аргументе *дата*, представляющем собой объект Date, вычисленное в локальном времени.

Date.getMinutes() ECMAScript v1

возвращает значение поля минут объекта Date

Синтаксис

дата.getMinutes()

Возвращаемое значение

Поле минут в аргументе *дата*, представляющем собой объект Date, вычисленное в локальном времени. Возвращаемое значение может принимать значения от 0 до 59.

Date.getMonth() ECMAScript v1

возвращает месяц для объекта Date

Синтаксис

дата.getMonth()

Возвращаемое значение

Поле месяца в аргументе *дата*, представляющем собой объект Date, вычисленное в локальном времени. Возвращаемое значение может принимать значения от 0 (январь) до 11 (декабрь).

Date.getSeconds() ECMAScript v1

возвращает значение поля секунд объекта Date

Синтаксис

дата.getSeconds()

Возвращаемое значение

Поле секунд в аргументе *дата*, представляющем собой объект Date, в локальном времени. Возвращаемое значение может принимать значения от 0 до 59.

Date.getTime() ECMAScript v1

возвращает значение даты в миллисекундах

Синтаксис

дата.getTime()

Возвращаемое значение

Миллисекундное представление аргумента *дата*, представляющего собой объект Date, т. е. число миллисекунд между полночью 01.01.1970 и датой/временем, определяемыми *дата*.

Описание

Метод getTime() преобразует дату и время в одно целое значение. Это удобно, когда требуется сравнить два объекта Date или определить время, прошедшее между двумя датами. Обратите внимание: миллисекундное представление даты не зависит от часового пояса, поэтому отсутствует метод getUTCTime(), дополняющий данный. Не путайте метод getTime() с методами getDay() и getDate(), возвращающими соответственно день недели и день месяца.

Методы Date.parse() и Date.UTC() позволяют преобразовать спецификацию даты и времени в миллисекундное представление, обходя избыточное создание объекта Date.

См. также Date, Date.parse(), Date.setTime(), Date.UTC()

Date.getTimezoneOffset() ECMAScript v1

определяет смещение относительно GMT

Синтаксис

дата.getTimezoneOffset()

Возвращаемое значение

Разница в минутах между временем по Гринвичу (GMT) и локальным временем.

Описание

Функция getTimezoneOffset() возвращает разницу в минутах между универсальным и локальным временем, сообщая, в каком часовом поясе исполняется JavaScript-код и действует ли (или будет ли действовать) летнее время для указанной *даты*.

Возвращаемое значение измеряется в минутах, а не в часах, поскольку в некоторых странах имеются часовые пояса, не занимающие целого часового интервала.

Date.getUTCDate() ECMAScript v1

возвращает день месяца (универсальное время)

Синтаксис

дата.getUTCDate()

Возвращаемое значение

День месяца (значение между 1 и 31), полученный при вычислении *даты* в универсальном времени.

Date.getUTCDay() ECMAScript v1

возвращает день недели (универсальное время)

Синтаксис

дата.getUTCDay()

Возвращаемое значение

День недели, получаемый, когда *дата* выражена в универсальном времени. Возвращаемые значения могут находиться в интервале между 0 (воскресенье) и 6 (суббота).

Date.getUTCFullYear() ECMAScript v1

возвращает год (универсальное время)

Синтаксис

дата.getUTCFullYear()

Возвращаемое значение

Год, получаемый, когда *дата* вычисляется в универсальном времени. Возвращаемое значение – четырехзначный номер года, а не сокращение из двух цифр.

Date.getUTCHours() ECMAScript v1

возвращает значение поля часов объекта Date (универсальное время)

Синтаксис

дата.getUTCHours()

Возвращаемое значение

Поле часов для *даты*, вычисленное в универсальном времени. Возвращаемое значение – целое между 0 (полночь) и 23 (11 часов вечера).

Date.getUTCMilliseconds() ECMAScript v1

возвращает значение поля миллисекунд объекта Date (универсальное время)

Синтаксис

дата.getUTCMilliseconds()

Возвращаемое значение

Поле миллисекунд для *даты*, выраженное в универсальном времени.

Date.getUTCMinutes() ECMAScript v1

возвращает значение поля минут объекта Date (универсальное время)

Синтаксис

дата.getUTCMinutes()

Возвращаемое значение

Поле минут для *даты* в универсальном времени. Возвращает целое между 0 и 59.

Date.getUTCMonth() ECMAScript v1

возвращает месяц года (универсальное время)

Синтаксис

дата.getUTCMonth()

Возвращаемое значение

Месяц года, получающийся, когда *дата* вычислена в универсальном времени. Возвращает целое между 0 (январь) и 11 (декабрь). Обратите внимание: объект Date обозначает первый день месяца цифрой 1, но первому месяцу года соответствует цифра 0.

Date.getUTCSeconds() ECMAScript v1

возвращает значение поля секунд объекта Date (универсальное время)

Синтаксис

дата.getUTCSeconds()

Возвращаемое значение

Поле секунд *даты* в универсальном времени. Возвращает целое между 0 и 59.

Date.getYear() ECMAScript v1; устарел в ECMAScript v3

возвращает значение поля года объекта Date (универсальное время)

Синтаксис

дата.getYear()

Возвращаемое значение

Поле года для указанного аргумента *дата*, представляющего собой объект Date, минус 1900.

Описание

Метод getYear() возвращает поле года для указанного объекта Date минус 1900. По спецификации ECMAScript v3 этот метод не является обязательным в совместимых реализациях JavaScript; используйте вместо него метод getFullYear().

Date.parse() ECMAScript v1

синтаксический разбор строки даты/времени

Синтаксис

Date.parse(*дата*)

Аргументы

дата Строка для разбора, содержащая дату и время.

Возвращаемое значение

Количество миллисекунд между указанными датой/временем и полночью 1 января 1970 года по Гринвичу.

Описание

Метод `Date.parse()` — это статический метод объекта `Date`. Он всегда вызывается через конструктор `Date` как `Date.parse()`, а не через объект `Date` как *дата*`.parse()`. Метод `Date.parse()` принимает один строковый аргумент, анализирует дату, содержащуюся в строке, и возвращает ее в виде числа миллисекунд, которое может использоваться непосредственно для создания нового объекта `Date` или для установки даты в существующем объекте `Date` с помощью `Date.setTime()`.

Стандарт ECMAScript не определяет формат строк, которые могут быть разобраны методом `Date.parse()`, за исключением того, что он может разбирать строки, возвращаемые методами `Date.toString()` и `Date.toUTCString()`. К сожалению, эти функции форматируют даты зависимым от реализации образом, поэтому нет универсального способа написания дат, гарантированно понятного любым реализациям JavaScript.

См. также `Date`, `Date.setTime()`, `Date.toGMTString()`, `Date.UTC()`

Date.setDate() ECMAScript v1

устанавливает день месяца

Синтаксис

дата`.setDate(`*день_месяца*`)`

Аргументы

день_месяца

Целое между 1 и 31, выступающее в качестве нового значения (в локальном времени) поля *день_месяца* объекта *дата*.

Возвращаемое значение

Миллисекундное представление измененной даты. До выхода стандарта ECMAScript этот метод ничего не возвращал.

Date.setFullYear() ECMAScript v1

устанавливает год и, возможно, месяц и день месяца

Синтаксис

дата`.setFullYear(`*год*`)`

дата`.setFullYear(`*год, месяц*`)`

дата`.setFullYear(`*год, месяц, день*`)`

Аргументы

год

Год, выраженный в локальном времени, который должен быть установлен в *дате*. Этот аргумент должен быть целым, включающим век, например 1999; не может быть сокращением, таким как 99.

месяц

Необязательное целое между 0 и 11, используемое для установки нового значения поля месяца (в локальном времени) для *даты*.

день

Необязательное целое между 1 и 31, выступающее в качестве нового значения поля «день месяца» для *даты* (в локальном времени).

Возвращаемое значение

Внутреннее миллисекундное представление измененной даты.

Date.setHours() ECMAScript v1

устанавливает значения полей часов, минут, секунд и миллисекунд объекта Date

Синтаксис

```
дата.setHours(часы)
дата.setHours(часы, минуты)
дата.setHours(часы, минуты, секунды)
дата.setHours(часы, минуты, секунды, миллисекунды)
```

Аргументы

часы

Целое между 0 (полночь) и 23 (11 часов вечера) локального времени, устанавливаемое в качестве нового значения часов в *дате*.

минуты

Необязательное целое между 0 и 59, используемое в качестве нового значения поля минут в *дате* (в локальном времени). Этот аргумент не поддерживался до выхода стандарта ECMAScript.

секунды

Необязательное целое между 0 и 59. Представляет собой новое значение поля секунд в *дате* (в локальном времени). Этот аргумент не поддерживался до выхода стандарта ECMAScript.

миллисекунды

Необязательное целое между 0 и 999, выступающее в качестве нового значения поля миллисекунд в *дате* (в локальном времени). Этот аргумент не поддерживался до выхода стандарта ECMAScript.

Возвращаемое значение

Миллисекундное представление измененной даты. До выхода стандарта ECMAScript этот метод ничего не возвращал.

Date.setMilliseconds() ECMAScript v1

устанавливает значение поля миллисекунд объекта Date

Синтаксис

```
дата.setMilliseconds(миллисекунды)
```

Аргументы

миллисекунды

Поле миллисекунд, выраженное в локальном времени, для установки в *дате*. Этот аргумент должен быть целым между 0 и 999.

Возвращаемое значение

Миллисекундное представление измененной даты.

Date.setMinutes() ECMAScript v1

устанавливает значения полей минут, секунд и миллисекунд объекта Date

Синтаксис

дата.setMinutes(*минуты*)

дата.setMinutes(*минуты, секунды*)

дата.setMinutes(*минуты, секунды, миллисекунды*)

Аргументы

минуты

Целое между 0 и 59, устанавливаемое в качестве значения минут (в локальном времени) в аргументе *дата*, представляющем собой объект `Date`.

секунды

Необязательное целое между 0 и 59, выступающее в качестве нового значения поля секунд *даты* (в локальном времени). Этот аргумент не поддерживался до выхода стандарта ECMAScript.

миллисекунды

Необязательное целое между 0 и 999, представляющее собой новое значение (в локальном времени) поля миллисекунд *даты*. Этот аргумент не поддерживался до выхода стандарта ECMAScript.

Возвращаемое значение

Миллисекундное представление измененной даты. До выхода стандарта ECMAScript этот метод ничего не возвращал.

Date.setMonth() ECMAScript v1

устанавливает месяц и день месяца объекта Date

Синтаксис

дата.setMonth(*месяц*)

дата.setMonth(*месяц, день*)

Аргументы

месяц

Целое между 0 (январь) и 11 (декабрь), устанавливаемое в качестве значения месяца для аргумента *дата*, представляющего собой объект `Date`, в локальном времени. Обратите внимание: месяцы нумеруются, начиная с 0, а дни в месяце – с 1.

день

Необязательное целое между 1 и 31, выступающее в качестве нового значения поля дня месяца в *дате* (в локальном времени). Этот аргумент не поддерживался до выхода стандарта ECMAScript.

Возвращаемое значение

Миллисекундное представление измененной даты. До выхода стандарта ECMAScript этот метод ничего не возвращал.

Date.setSeconds() ECMAScript v1

устанавливает значения полей секунд и миллисекунд объекта Date

Синтаксис

дата.setSeconds(*секунды*)

дата.setSeconds(*секунды*, *миллисекунды*)

Аргументы

секунды

Целое между 0 и 59, устанавливаемое как значение секунд в аргументе *дата*, представляющем собой объект `Date`.

миллисекунды

Необязательное целое между 0 и 999, выступающее в качестве нового значения поля миллисекунд в *дате* (в локальном времени). Этот аргумент не поддерживался до выхода стандарта ECMAScript.

Возвращаемое значение

Миллисекундное представление измененной даты. До выхода стандарта ECMAScript этот метод ничего не возвращал.

Date.setTime() ECMAScript v1

устанавливает значение даты в миллисекундах

Синтаксис

дата.setTime(*миллисекунды*)

Аргументы

миллисекунды

Количество миллисекунд между требуемыми датой/временем и полночью по Гринвичу 1 января 1970 года. Подобное миллисекундное значение может быть также передано конструктору `Date()` и получено при вызове методов `Date.UTC()` и `Date.parse()`. Представление даты в миллисекундном формате делает ее независимой от часового пояса.

Возвращаемое значение

Аргумент *миллисекунды*. До выхода стандарта ECMAScript метод ничего не возвращал.

Date.setUTCDate() ECMAScript v1

устанавливает день месяца (универсальное время)

Синтаксис

дата.setUTCDate(*день_месяца*)

Аргументы

день_месяца

День месяца, выраженный в универсальном времени и устанавливаемый в *дате*. Этот аргумент должен быть целым между 1 и 31.

Возвращаемое значение

Внутреннее миллисекундное представление измененной даты.

Date.setUTCFullYear() ECMAScript v1

устанавливает год, месяц и день месяца (универсальное время)

Синтаксис

дата.setUTCFullYear(*год*)

дата.setUTCFullYear(*год, месяц*)

дата.setUTCFullYear(*год, месяц, день*)

Аргументы

год

Год, выраженный в универсальном времени, для установки в *дате*. Этот аргумент должен быть целым, включающим век, например 1999, а не сокращением, как 99.

месяц

Необязательное целое между 0 и 11, выступающее в качестве нового значения в поле месяца *даты* (в универсальном времени). Обратите внимание: месяцы нумеруются, начиная с 0, тогда как нумерация дней месяцев начинается с 1.

день

Необязательное целое между 1 и 31, представляет собой новое значение (в универсальном времени) поля «день месяца» в *дате*.

Возвращаемое значение

Внутреннее миллисекундное представление измененной даты.

Date.setUTCHours() ECMAScript v1

устанавливает значения полей часов, минут, секунд и миллисекунд (универсальное время)

Синтаксис

дата.setUTCHours(*часы*)

дата.setUTCHours(*часы, минуты*)

дата.setUTCHours(*часы, минуты, секунды*)

дата.setUTCHours(*часы, минуты, секунды, миллисекунды*)

Аргументы

часы

Поле «часы», выраженное в универсальном времени, для установки в *дате*. Этот аргумент должен быть целым между 0 (полночь) и 23 (11 часов вечера).

минуты

Необязательное целое между 0 и 59, выступающее в качестве нового значения поля минут в *дате* (в универсальном времени).

секунды

Необязательное целое между 0 и 59, представляет собой новое значение поля секунд в *дате* (в универсальном времени).

миллисекунды

Необязательное целое между 0 и 999, используемое в качестве нового значения для поля миллисекунд в *дате* (в универсальном времени).

Возвращаемое значение

Внутреннее миллисекундное представление измененной даты.

Date.setUTCMilliseconds() ECMAScript v1

устанавливает значение поля миллисекунд в объекте Date (универсальное время)

Синтаксис

дата.setUTCMilliseconds(*миллисекунды*)

Аргументы

миллисекунды

Поле миллисекунд, выраженное в универсальном времени, которое должно быть установлено в *дате*. Этот аргумент должен быть целым между 0 и 999.

Возвращаемое значение

Внутреннее миллисекундное представление измененной даты.

Date.setUTCMinutes() ECMAScript v1

устанавливает значения полей минут, секунд и миллисекунд (универсальное время)

Синтаксис

дата.setUTCMinutes(*минуты*)

дата.setUTCMinutes(*минуты, секунды*)

дата.setUTCMinutes(*минуты, секунды, миллисекунды*)

Аргументы

минуты

Поле минут, выраженное в универсальном времени, для установки в *дате*. Этот аргумент должен принимать значение между 0 и 59.

секунды

Необязательное целое между 0 и 59, выступающее в качестве нового значения поля секунд в *дате* (в универсальном времени).

миллисекунды

Необязательное целое между 0 и 999, представляет собой новое значение (в универсальном времени) поля миллисекунд в *дате*.

Возвращаемое значение

Внутреннее миллисекундное представление измененной даты.

Date.setUTCMonth() ECMAScript v1

устанавливает месяц и день месяца (универсальное время)

Синтаксис

дата.setUTCMonth(*месяц*)

дата.setUTCMonth(*месяц, день*)

Аргументы

месяц

Месяц, выраженный в универсальном времени, для установки в *дате*. Этот аргумент должен быть целым между 0 (январь) и 11 (декабрь). Обратите внимание: месяцы нумеруются, начиная с 0, а дни в месяце – с 1.

день

Необязательное целое между 1 и 31, выступающее в качестве нового значения поля дня месяца в *дате* (в универсальном времени).

Возвращаемое значение

Внутреннее миллисекундное представление измененной даты.

Date.setUTCSeconds() ECMAScript v1

устанавливает значения полей секунд и миллисекунд (универсальное время)

Синтаксис

дата.setUTCSeconds(*секунды*)

дата.setUTCSeconds(*секунды, миллисекунды*)

Аргументы

секунды

Поле секунд, выраженное в универсальном времени, для установки в *дате*. Этот аргумент должен быть целым между 0 и 59.

миллисекунды

Необязательное целое между 0 и 999, используемое в качестве нового значения поля миллисекунд *даты* (в универсальном времени).

Возвращаемое значение

Внутреннее миллисекундное представление измененной даты.

Date.setYear() ECMAScript v1; устарел в ECMAScript v3

устанавливает год в объекте Date

Синтаксис

дата.setYear(*год*)

Аргументы

год

Целое, устанавливаемое в качестве значения года (в локальном времени) для аргумента *дата*, представляющего собой объект Date. Если это значение находится между 0 и 99, к нему добавляется 1900, и оно рассматривается как год между 1900 и 1999.

Возвращаемое значение

Миллисекундное представление измененной даты. До выхода стандарта ECMAScript этот метод ничего не возвращал.

Описание

Метод setYear() устанавливает поле год в указанном объекте Date, особым образом обрабатывая интервал времени между 1900 и 1999 годами.

По спецификации ECMAScript v3 этот метод не является обязательным в совместимых реализациях JavaScript; вместо него рекомендован метод setFullYear().

Date.toDateString() ECMAScript v3

возвращает дату из объекта Date в виде строки

Синтаксис

дата.toDateString()

Возвращаемое значение

Зависящее от реализации и понятное человеку строковое представление даты (без времени), указанное в аргументе *дата*, представляющем собой объект Date, в локальном времени.

См. также

Date.toLocaleDateString(), Date.toLocaleString(), Date.toLocaleTimeString(), Date.toString(), Date.toTimeString()

Date.toGMTString() ECMAScript v1; устарел в ECMAScript v3

преобразует Date в строку универсального времени

Синтаксис

дата.toGMTString()

Возвращаемое значение

Строковое представление даты и времени, указанное в аргументе *дата*, представляющем собой объект Date. Перед преобразованием в строку дата переводится из локального времени во время по Гринвичу.

Описание

Метод `toGMTString()` признан устаревшим, вместо него рекомендуется использовать аналогичный метод `Date.toUTCString()`.

По спецификации ECMAScript v3 совместимые реализации JavaScript больше не обязаны предоставлять этот метод; используйте вместо него метод `toUTCString()`.

См. также `Date.toUTCString()`

Date.toLocaleDateString() ECMAScript v3

возвращает дату из Date в виде строки с учетом региональных настроек

Синтаксис

дата`.toLocaleDateString()`

Возвращаемое значение

Зависящее от реализации и понятное человеку строковое представление даты (без времени) из объекта *дата*, выраженное в локальном времени и отформатированное в соответствии с региональными настройками.

См. также

`Date.toDateString()`, `Date.toLocaleString()`, `Date.toLocaleTimeString()`, `Date.toString()`, `Date.toTimeString()`

Date.toLocaleString() ECMAScript v1

преобразует дату в строку с учетом региональных настроек

Синтаксис

дата`.toLocaleString()`

Возвращаемое значение

Строковое представление даты и времени, заданных аргументом *дата*. Дата и время представлены в локальном часовом поясе и отформатированы в соответствии с региональными настройками.

Порядок использования

Метод `toLocaleString()` преобразует дату в строку в соответствии с локальным часовым поясом. При форматировании даты и времени используются региональные настройки, поэтому формат может отличаться на разных платформах и в разных странах. Метод `toLocaleString()` возвращает строку, отформатированную в соответствии с предпочтительным для пользователя форматом даты и времени.

См. также

`Date.toLocaleDateString()`, `Date.toLocaleTimeString()`, `Date.toString()`, `Date.toUTCString()`

Date.toLocaleTimeString() ECMAScript v3

возвращает время из Date в виде строки с учетом региональных настроек

Синтаксис

дата`.toLocaleTimeString()`

Возвращаемое значение

Зависящее от реализации и понятное человеку строковое представление данных о времени из объекта *дата*, выраженное в локальном часовом поясе и отформатированное в соответствии с региональными настройками.

См. также

`Date.toDateString()`, `Date.toLocaleDateString()`, `Date.toLocaleString()`, `Date.toString()`, `Date.toTimeString()`

Date.toString() ECMAScript v1

преобразует объект Date в строку **переопределяет Object.toString()**

Синтаксис

дата`.toString()`

Возвращаемое значение

Понятное человеку строковое представление *даты* в локальном часовом поясе.

Описание

Метод `toString()` возвращает понятное человеку и зависящее от реализации строковое представление *даты*. В отличие от `toUTCString()`, метод `toString()` вычисляет дату в локальном часовом поясе. В отличие от `toLocaleString()`, метод `toString()` может представлять дату и время без учета региональных настроек.

См. также

`Date.parse()`, `Date.toDateString()`, `Date.toLocaleString()`, `Date.toTimeString()`, `Date.toUTCString()`

Date.toTimeString() ECMAScript v3

возвращает время из объекта Date в виде строки

Синтаксис

дата`.toTimeString()`

Возвращаемое значение

Зависящее от реализации, понятное человеку строковое представление данных о времени из объекта *дата*, выраженное в локальном часовом поясе.

См. также

`Date.toString()`, `Date.toDateString()`, `Date.toLocaleDateString()`, `Date.toLocaleString()`, `Date.toLocaleTimeString()`

Date.toUTCString() ECMAScript v1

преобразует объект Date в строку (универсальное время)

Синтаксис

дата`.toUTCString()`

Возвращаемое значение

Понятное человеку строковое представление даты, выраженное в универсальном времени.

Описание

Метод `toUTCString()` возвращает зависящую от реализации строку, представляющую дату в универсальном времени.

См. также `Date.toLocaleString()`, `Date.toString()`

Date.UTC() ECMAScript v1

преобразует спецификацию даты в миллисекунды

Синтаксис

`Date.UTC(`*год, месяц, день, часы, минуты, секунды, мс*`)`

Аргументы

год

Год в четырехзначном формате. Если аргумент находится между 0 и 99, к нему добавляется 1900,и он рассматривается как год между 1900 и 1999.

месяц

Месяц, заданный в виде целого от 0 (январь) до 11 (декабрь).

день

День месяца, заданный в виде целого от 1 до 31. Обратите внимание: наименьшее значение этого аргумента равно 1, наименьшее значение других аргументов – 0. Этот аргумент не является обязательным.

часы

Час, заданный в виде целого от 0 (полночь) до 23 (11 часов вечера). Этот аргумент может отсутствовать.

минуты

Минуты в часе, заданные в виде целого от 0 до 59. Этот аргумент может отсутствовать.

секунды

Секунды в минутах, заданные в виде целого от 0 до 59. Этот аргумент может отсутствовать.

мс

Количество миллисекунд. Этот аргумент может отсутствовать; игнорировался до выхода стандарта ECMAScript.

Возвращаемое значение

Миллисекундное представление указанного универсального времени. Метод возвращает количество миллисекунд между полночью по Гринвичу 1 января 1970 года и указанным временем.

Описание

Метод `Date.UTC()` – это статический метод, который вызывается через конструктор `Date()`, а не через отдельный объект `Date`.

Аргументы Date.UTC() задают дату и время и подразумевают время в формате UTC. Указанное время UTC преобразуется в миллисекундный формат, который может использоваться методом-конструктором Date() и методом Date.setTime().

Метод-конструктор Date() может принимать аргументы даты и времени, идентичные тем, что принимает метод Date.UTC(). Разница в том, что конструктор Date() подразумевает локальное время, а Date.UTC() – время по Гринвичу (GMT). Создать объект Date, используя спецификацию времени в UTC, можно следующим образом:

```
d = new Date(Date.UTC(1996, 4, 8, 16, 30));
```

См. также Date, Date.parse(), Date.setTime()

Date.valueOf() ECMAScript v1

преобразует объект Date в миллисекунды **переопределяет Object.valueOf()**

Синтаксис

дата.valueOf()

Возвращаемое значение

Миллисекундное представление *даты*. Возвращаемое значение совпадает со значением, возвращаемым Date.getTime().

decodeURI() ECMAScript v3

декодирует символы в URI

Синтаксис

decodeURI(*uri*)

Аргументы

uri

Строка, содержащая в закодированном виде URI (Uniform Resource Identifier – унифицированный идентификатор ресурса) или другой текст, подлежащий декодированию.

Возвращаемое значение

Копия аргумента *uri*, в которой все шестнадцатеричные управляющие последовательности заменены на символы, которые они представляют.

Исключения

URIError

Означает, что одна или несколько управляющих последовательностей в *uri* имеют неверный формат и не могут быть правильно декодированы.

Описание

decodeURI() – это глобальная функция, возвращающая декодированную копию аргумента *uri*. Она выполняет действие, обратное действию функции encodeURI(); подробности см. в описании этой функции.

См. также

decodeURIComponent(), encodeURI(), encodeURIComponent(), escape(), unescape()

decodeURIComponent() ECMAScript v3

декодирует управляющие последовательности символов в компоненте URI

Синтаксис

```
decodeURIComponent(s)
```

Аргументы

s

Строка, содержащая закодированный компонент URI или другой текст, который должен быть декодирован.

Возвращаемое значение

Копия аргумента *s*, в которой шестнадцатеричные управляющие последовательности заменены представляемыми ими символами.

Исключения

URIError

Означает, что одна или несколько управляющих последовательностей в аргументе *s* имеют неверный формат и не могут быть правильно декодированы.

Описание

decodeURIComponent() — глобальная функция, возвращающая декодированную копию своего аргумента *s*. Ее действие обратно кодированию, выполняемому функцией encodeURIComponent; подробности см. в справочной статье по этой функции.

См. также decodeURI(), encodeURI(), encodeURIComponent(), escape(), unescape()

encodeURI() ECMAScript v3

выполняет кодирование URI с помощью управляющих последовательностей

Синтаксис

```
encodeURI(uri)
```

Аргументы

uri

Строка, содержащая URI или другой текст, который должен быть закодирован.

Возвращаемое значение

Копия аргумента *uri*, в которой некоторые символы заменены шестнадцатеричными управляющими последовательностями.

Исключения

URIError

Указывает, что строка *uri* содержит искаженные пары Unicode-символов и не может быть закодирована.

Описание

encodeURI() — это глобальная функция, возвращающая закодированную копию аргумента *uri*. Не кодируются символы, цифры и следующие знаки пунктуации кода ASCII:

```
- _ . ! ~ * ' ( )
```

Функция encodeURI() кодирует URI целиком, поэтому следующие символы пунктуации, имеющие в URI специальное значение, также не кодируются:

```
; / ? : @ & = + $ , #
```

Любые другие символы в *uri* заменяются путем преобразования символа в его код UTF-8 и последующего кодирования каждого из полученных байтов шестнадцатеричной управляющей последовательностью в формате %xx. В этой схеме кодирования ASCII-символы заменяются одной последовательностью %xx, символы с кодами от \u0080 до \u07ff – двумя управляющими последовательностями, а все остальные 16-разрядные Unicode-символы – тремя управляющими последовательностями.

При использовании этого метода для кодирования URI необходимо быть уверенным, что ни один из компонентов URI (например, строка запроса) не содержит символов-разделителей URI, таких как ? или #. Если компоненты могут содержать эти символы, необходимо кодировать каждый компонент отдельно с помощью функции encodeURIComponent().

Метод decodeURI() предназначен для выполнения действия, обратного кодированию. До выхода ECMAScript v3 с помощью методов escape() и unescape(), сейчас признанных устаревшими, выполнялись сходные кодирование и декодирование.

Пример

```
// Возвращает http://www.isp.com/app.cgi?arg1=1&arg2=hello%20world
encodeURI("http://www.isp.com/app.cgi?arg1=1&arg2=hello world");
encodeURI("\u00a9"); // Символ копирайта кодируется в %C2%A9
```

См. также

decodeURI(), decodeURIComponent(), encodeURIComponent(), escape(), unescape()

encodeURIComponent() ECMAScript v3

выполняет кодирование компонентов URI с помощью управляющих последовательностей

Синтаксис

encodeURIComponent(*s*)

Аргументы

s Строка, содержащая фрагмент URI или другой текст, подлежащий кодированию.

Возвращаемое значение

Копия *s*, в которой определенные символы заменены шестнадцатеричными управляющими последовательностями.

Исключения

URIError

Указывает, что строка *s* содержит искаженные пары Unicode-символов и не может быть закодирована.

Описание

encodeURIComponent() – это глобальная функция, возвращающая закодированную копию своего аргумента *s*. Не кодируются буквы, цифры и следующие знаки пунктуации из кода ASCII:

```
- _ . ! ~ * ' ( )
```

Все остальные символы, в том числе такие символы пунктуации, как /, :, #, служащие для разделения различных компонентов URI, заменяются одной или несколькими шестнадцатеричными управляющими последовательностями. Описание используемой схемы кодирования см. в статье, посвященной функции encodeURI().

Обратите внимание на разницу между encodeURIComponent() и encodeURI(): функция encodeURIComponent() предполагает, что ее аргументом является фрагмент URI (такой как протокол, имя хоста, путь или строка запроса). Поэтому она преобразует символы пунктуации, используемые для разделения фрагментов URI.

Пример

```
encodeURIComponent("hello world?"); // Возвращает hello%20world%3F
```

См. также decodeURI(), decodeURIComponent(), encodeURI(), escape(), unescape()

Error ECMAScript v3

обобщенное исключение **Object→Error**

Конструктор

new Error()

new Error(*сообщение*)

Аргументы

сообщение

Необязательное сообщение об ошибке, предоставляющее дополнительную информацию об исключении.

Возвращаемое значение

Вновь созданный объект Error. Если задан аргумент *сообщение*, объект Error будет использовать его в качестве значения своего свойства message; в противном случае он возьмет в качестве значения этого свойства предлагаемую по умолчанию строку, определенную реализацией. Когда конструктор Error() вызывается как функция (без оператора new), он ведет себя так же, как при вызове с оператором new.

Свойства

message

Сообщение об ошибке, предоставляющее дополнительную информацию об исключении. В этом свойстве хранится строка, переданная конструктору, или предлагаемая по умолчанию строка, определенная реализацией.

name

Строка, задающая тип исключения. Для экземпляров класса Error и всех его подклассов это свойство задает имя конструктора, с помощью которого был создан экземпляр.

Методы

toString()

Возвращает строку, определенную в реализации, которая представляет этот объект Error.

Описание

Экземпляры класса Error представляют ошибки или исключения и обычно используются с инструкциями throw и try/catch. Свойство name задает тип исключения, а посредством свойства message можно создать и отправить пользователю сообщение с подробной информацией об исключении.

Интерпретатор JavaScript никогда непосредственно не создает объект Error. Вместо этого он создает экземпляры одного из подклассов Error, таких как SyntaxError или RangeError. В вашем собственном коде для предупреждения об исключении может быть удобнее создавать объекты Error или просто выдавать сообщение об ошибке или ее код в виде элементарного строкового или числового значения.

Обратите внимание: спецификация ECMAScript определяет для класса Error метод toString() (он наследуется всеми подклассами Error), но не требует, чтобы этот метод возвращал строку, содержащую значение свойства message. Поэтому не следует ожидать, что метод toString() преобразует объект Error в строку, понятную человеку. Чтобы выдать пользователю сообщение об ошибке, необходимо явно использовать свойства name и message объекта Error.

Пример

Предупредить об исключении можно так:

```
function factorial(x) {
if (x < 0) throw new Error("factorial: x должно быть >= 0");
if (x <= 1) return 1; else return x * factorial(x-1);
}
```

Перехватывая исключение, можно сообщить о нем пользователю с помощью следующего кода (содержащего клиентский метод Window.alert()):

```
try { &*(&/* здесь возникает ошибка */ }
catch(e) {
    if (e instanceof Error) { // Это экземпляр Error или подкласса?
        alert(e.name + ": " + e.message);
    }
}
```

См. также

EvalError, RangeError, ReferenceError, SyntaxError, TypeError, URIError

Error.message ECMAScript v3

сообщение об ошибке

Синтаксис

error.message

Описание

Свойство message объекта Error (или экземпляра любого подкласса Error) предназначено для хранения понятной человеку строки, содержащей подробные сведения о возникшей ошибке или исключении. Если конструктору Error() передан аргумент *message*, он становится значением свойства message. Если аргумент *message* передан не был, объект Error наследует для этого свойства значение по умолчанию, определенное реализацией (которое может быть пустой строкой).

Error.name

ECMAScript v3

тип ошибки

Синтаксис

error.name

Описание

Свойство name объекта Error (или экземпляра любого подкласса Error) задает тип произошедшей ошибки или исключения. Все объекты Error наследуют это свойство от своего конструктора. Значение свойства совпадает с именем конструктора. Другими словами, у объектов SyntaxError свойство name равно «SyntaxError», а у объектов EvalError — «EvalError».

Error.toString

ECMAScript v3

преобразует объект Error в строку **переопределяет Object.toString()**

Синтаксис

error.toString()

Возвращаемое значение

Строка, определенная реализацией. Стандарт ECMAScript ничего не говорит о возвращаемом этим методом значении, за исключением того, что оно должно быть строкой. Стоит отметить, что он не требует, чтобы возвращаемая строка содержала имя ошибки или сообщение об ошибке.

escape()

ECMAScript v1; устарело в ECMAScript v3

кодирует строку

Синтаксис

escape(*s*)

Аргументы

s Строка, которая должна быть закодирована (с помощью управляющих последовательностей).

Возвращаемое значение

Закодированная копия *s*, в которой определенные символы заменены шестнадцатеричными управляющими последовательностями.

Описание

escape() — глобальная функция, которая возвращает новую строку, содержащую закодированную версию аргумента *s*. Сама строка *s* не меняется.

Функция escape() возвращает строку, в которой все символы *s*, отличные от букв, цифр и символов пунктуации (@, *, _, +, -, . и /) кода ASCII заменены управляющими последовательностями в формате %xx или %uxxxx (где x обозначает шестнадцатеричную цифру). Unicode-символы от \u0000 до \u00ff заменяются управляющей последовательностью %xx, все остальные Unicode-символы — последовательностью %uxxxx.

Строка, закодированная с помощью escape(), декодируется функцией unescape().

Хотя функция escape() стандартизована в первой версии ECMAScript, она была признана устаревшей и удалена из стандарта в ECMAScript v3. Реализации ECMAScript обычно поддерживают эту функцию, хотя это не обязательно. Вместо escape() следует использовать функции encodeURI() и encodeURIComponent().

Пример

```
escape("Hello World!"); // Возвращает "Hello%20World%21"
```

См. также encodeURI(), encodeURIComponent()

eval() ECMAScript v1

исполняет содержащийся в строке JavaScript-код

Синтаксис

eval(*код*)

Аргументы

код

Строка, содержащая выполняемое выражение или инструкции.

Возвращаемое значение

Значение, полученное в результате исполнения *кода*, если оно есть.

Исключения

SyntaxError

Указывает, что аргумент *код* не содержит корректного JavaScript-кода.

EvalError

Указывает, что функция eval() была вызвана некорректно, например через идентификатор, отличный от «eval». В следующем разделе описываются ограничения, налагаемые на эту функцию.

Другое исключение

Если JavaScript-код, переданный в eval(), генерирует исключение, eval() передает его вызывающей стороне.

Описание

Метод eval() – это глобальный метод, вычисляющий строку, в которой содержится *код* на языке JavaScript. Если *код* содержит JavaScript-выражение, eval вычисляет выражение и возвращает его значение. Если *код* содержит одну или несколько JavaScript-инструкций, eval() исполняет эти инструкции и возвращает то значение (если оно есть), которое возвращает последняя инструкция. Если *код* не возвращает никакого значения, eval() возвращает значение undefined. И наконец, если *код* генерирует исключение, eval() передает это исключение вызывающей стороне.

Возможности метода eval() в отношении языка JavaScript очень мощные, тем не менее этот метод не часто используется в реальных программах. Очевидной областью его применения являются программы, работающие как рекурсивные интерпретаторы JavaScript или динамически генерирующие и выполняющие JavaScript-код.

Большинство JavaScript-функций и JavaScript-методов, принимающих строковые аргументы, принимают также аргументы других типов и перед обработкой просто

преобразуют эти значения в строки. Метод eval() ведет себя по-другому. Если аргумент *код* не является элементарным строковым значением, он возвращается в неизменном виде. Поэтому будьте внимательны, чтобы случайно не передать функции eval() объект String вместо элементарного строкового значения.

Ради эффективности реализации стандарт ECMAScript v3 налагает на применение метода eval() необычное ограничение. Реализация ECMAScript разрешает генерировать исключение EvalError, если вы пытаетесь переписать свойство eval или присваиваете метод eval() другому свойству и пытаетесь вызвать его через это свойство.

Пример

```
eval("1+2");   // Возвращает 3
// Этот код использует клиентские JavaScript-методы для запроса выражения
// от пользователя и вывода результатов его вычисления.
// Подробности см. в описаниях клиентских методов Window.alert()
// и Window.prompt().
try {
    alert("Результат: " + eval(prompt("Введите выражение:","")));
}
catch(exception) {
    alert(exception);
}
var myeval = eval; // Может генерировать исключение EvalError
myeval("1+2");     // Может генерировать исключение EvalError
```

EvalError ECMAScript v3

генерируется, когда некорректно используется метод eval() **Object→Error→EvalError**

Конструктор

new EvalError()

new EvalError(*сообщение*)

Аргументы

сообщение

Необязательное сообщение об ошибке, предоставляющее дополнительную информацию об исключении. Если этот аргумент указан, он принимается в качестве значения свойства message объекта EvalError.

Возвращаемое значение

Вновь созданный объект EvalError. Если задан аргумент *сообщение*, объект Error берет его в качестве значения своего свойства message; в противном случае в качестве значения этого свойства используется предлагаемая по умолчанию строка, определенная реализацией. Когда конструктор EvalError() вызывается в качестве функции (без оператора new), он ведет себя точно так же, как при вызове с оператором new.

Свойства

message

Сообщение об ошибке, предоставляющее дополнительную информацию об исключении. В этом свойстве хранится строка, переданная конструктору, или предлагаемая по умолчанию строка, определенная реализацией. Подробности см. в статье с описанием свойства Error.message.

name

Строка, определяющая тип исключения. Для всех объектов EvalError значение этого свойства равно "EvalError".

Описание

Экземпляры класса EvalError могут создаваться, когда глобальная функция eval() вызывается с любым другим именем. Ограничения на способы вызова функции eval() рассмотрены в ее описании. Информация о генерации и перехвате исключений приводится в статье, посвященной классу Error.

См. также

Error, Error.message, Error.name

Function

ECMAScript v1

функция JavaScript **Object→Function**

Синтаксис

```
function имя_функции(список_имен_аргументов) // Инструкция определения функции
{
  тело
}
function(список_имен_аргументов) { тело }     // Неименованный функциональный литерал
имя_функции(список_значений_аргументов)       // Вызов функции
```

Конструктор

```
new Function(имена_аргументов..., тело)
```

Аргументы

имена_аргументов...

Любое количество строковых аргументов, которые присваивают имя одному или нескольким аргументам создаваемого объекта Function.

тело

Строка, задающая тело функции. Она может содержать любое количество JavaScript-инструкций, разделенных точками с запятой, и ссылаться на любые имена аргументов, заданные ранее в конструкторе.

Возвращаемое значение

Вновь созданный объект Function. Вызов функции приводит к выполнению JavaScript-кода, составляющего аргумент *тело*.

Исключения

SyntaxError

Указывает, что в аргументе *тело* или в одном из аргументов из перечня *имена_аргументов* имеется синтаксическая JavaScript-ошибка.

Свойства

arguments[]

Массив аргументов, переданных функции. Признано устаревшим.

`caller`

Ссылка на объект `Function`, вызвавший данную функцию, или `null`, если функция была вызвана из кода верхнего уровня. Признано устаревшим.

`length`

Число именованных аргументов, указанных при объявлении функции.

`prototype`

Объект, определяющий для функции конструктора свойства и методы, совместно используемые всеми объектами, созданными с помощью этого конструктора.

Методы

`apply()`

Вызывает функцию как метод заданного объекта, передавая ей указанный массив аргументов.

`call()`

Вызывает функцию как метод заданного объекта, передавая ей аргументы.

`toString()`

Возвращает строковое представление функции.

Описание

Функция в JavaScript – это фундаментальный тип данных. В главе 8 рассказывается, как определять и использовать функции, а в главе 9 рассматриваются близкие темы, касающиеся методов, конструкторов и свойств прототипов функций. Подробности см. в этих главах. Обратите внимание: функциональные объекты могут создаваться с помощью описанного здесь конструктора `Function()`, но это не эффективно, поэтому в большинстве случаев предпочтительным способом определения функции является инструкция определения функции или функциональный литерал.

В JavaScript 1.1 и более поздних версиях тело функции автоматически получает локальную переменную по имени `arguments`, которая ссылается на объект `Arguments`. Этот объект представляет собой массив значений, переданных функции в качестве аргументов. Не путайте его с устаревшим свойством `arguments[]`, описанным ранее. Подробности см. в статье об объекте `Arguments`.

См. также

`Arguments`; главы 8 и 9

Function.apply() ECMAScript v3

вызывает функцию как метод объекта

Синтаксис

функция.apply(*этот_объект, аргументы*)

Аргументы

этот_объект

Объект, к которому должна быть применена функция. В теле функции аргумент *этот_объект* становится значением ключевого слова `this`. Если указанный аргумент содержит значение `null`, используется глобальный объект.

аргументы

Массив значений, которые должны передаваться в качестве аргументов функции.

Возвращаемое значение

Значение, возвращаемое при вызове *функции*.

Исключения

TypeError

Генерируется, если метод вызывается для объекта, не являющегося функцией, или с аргументом *аргументы*, не являющимся массивом или объектом Arguments.

Описание

Метод apply() вызывает указанную функцию, как если бы она была методом объекта, заданного аргументом *этот_объект*, передавая ей аргументы, которые содержатся в массиве *аргументы*. Метод возвращает значение, возвращаемое при вызове функции. В теле функции ключевое слово this ссылается на объект *этот_объект*.

Аргумент *аргументы* должен быть массивом или объектом Arguments. Если аргументы функции должны передаваться в виде отдельных аргументов, а не в виде массива, следует использовать вызов Function.call().

Пример

```
// Применяет метод Object.toString(), предлагаемый по умолчанию для объекта,
// переопределяющего его собственной версией метода. Обратите внимание
// на отсутствие аргументов.
Object.prototype.toString.apply(o);
// Вызывает метод Math.max(), используемый для нахождения максимального
// элемента в массиве. Обратите внимание: в этом случае первый
// аргумент не имеет значения.
var data = [1,2,3,4,5,6,7,8];
Math.max.apply(null, data);
```

См. также

Function.call()

Function.arguments[]

ECMAScript v1; устарело в ECMAScript v3

аргументы, переданные функции

Синтаксис

функция.arguments[*i*]

функция.arguments.length

Описание

Свойство arguments объекта Function представляет собой массив аргументов, переданных функции. Этот массив определен только во время выполнения функции. Свойство arguments.length указывает количество элементов в массиве.

Этот свойство признано устаревшим, вместо него рекомендуется использовать объект Arguments. Хотя ECMAScript v1 поддерживает свойство Function.arguments, оно убрано из ECMAScript v3 и совместимые реализации могут больше его не поддерживать. Таким образом, его никогда не следует использовать в новых JavaScript-сценариях.

См. также

Arguments

Function.call() ECMAScript v3

вызывает функцию как метод объекта

Синтаксис

функция.call(*этот_объект, аргументы...*)

Аргументы

этот_объект

Объект, для которого должна быть вызвана *функция*. В теле функции аргумент *этот_объект* становится значением ключевого слова this. Если этот аргумент содержит значение null, используется глобальный объект.

аргументы...

Любое количество аргументов, передаваемых *функции*.

Возвращаемое значение

Значение, возвращаемое при вызове функции.

Исключения

TypeError

Генерируется, если метод вызывается для объекта, не являющегося функцией.

Описание

call() вызывает указанную *функцию*, как если бы она была методом объекта, указанного аргументом *этот_объект*, передавая ей любые аргументы, расположенные в списке аргументов после аргумента *этот_объект*. Вызов call() возвращает то, что возвращает вызываемая функция. Внутри тела функции ключевое слово this ссылается на объект *этот_объект* или на глобальный объект, если аргумент *этот_объект* содержит значение null.

Если требуется задать аргументы для передачи в функцию в виде массива, используйте функцию Function.apply().

Пример

```
// Вызывает метод Object.toString(), по умолчанию предлагаемый для объекта,
// переопределяющего его собственной версией метода. Обратите внимание
// на отсутствие аргументов.
Object.prototype.toString.call(o);
```

См. также

Function.apply()

Function.caller JavaScript 1.0; устарело в ECMAScript

функция, вызвавшая данную

Синтаксис

функция.caller

Описание

В ранних версиях JavaScript свойство caller объекта Function представляло собой ссылку на функцию, вызвавшую текущую функцию. Если функция вызывается из JavaScript-программы верхнего уровня, значение свойства caller равно null. Это

свойство может использоваться только внутри функции (т. е. свойство `caller` определено для функции, только пока она исполняется).

Свойство `Function.caller` не является частью стандарта ECMAScript и не обязательно для совместимых реализаций. Его не следует использовать.

Function.length ECMAScript v1

количество аргументов в объявлении функции

Синтаксис

функция`.length`

Описание

Свойство `length` функции указывает количество именованных аргументов, объявленных при определении функции. Фактически функция может вызываться с большим или меньшим количеством аргументов. Не путайте это свойство объекта `Function` со свойством `length` объекта `Arguments`, указывающим количество аргументов, фактически переданных функции. Пример имеется в статье о свойстве `Arguments.length`.

См. также

`Arguments.length`

Function.prototype ECMAScript v1

прототип класса объектов

Синтаксис

функция`.prototype`

Описание

Свойство `prototype` применяется тогда, когда функция вызывается как конструктор. Оно ссылается на объект, являющийся прототипом для целого класса объектов. Любой объект, созданный с помощью конструктора, наследует все свойства объекта, на который ссылается свойство `prototype`.

Обсуждение функций-конструкторов, свойства `prototype` и определений JavaScript-классов находится в главе 9.

См. также

Глава 9

Function.toString() ECMAScript v1

преобразует функцию в строку

Синтаксис

функция`.toString()`

Возвращаемое значение

Строка, представляющая функцию.

Исключения

`TypeError`

Генерируется, если метод вызывается для объекта, не являющегося функцией.

Описание

Метод `toString()` объекта `Function` преобразует функцию в строку способом, зависящим от реализации. В большинстве реализаций, например в Firefox и IE, данный метод возвращает строку JavaScript-кода, которая включает ключевое слово `function`, список аргументов, полное тело функции и т. д. В этих реализациях результат работы метода `toString()` может передаваться в виде аргумента функции `eval()`. Однако такое поведение не оговаривается спецификациями и на него не следует полагаться.

getClass() LiveConnect

возвращает объект JavaClass объекта JavaObject

Синтаксис

`getClass(`*`объект_java`*`)`

Аргументы

`объект_java`

Объект JavaObject.

Возвращаемое значение

Объект `JavaClass` объекта `JavaObject` (*`объект_java`*).

Описание

`getClass()` – это функция, которая в качестве аргумента получает объект `JavaObject` (*`объект_java`*) и возвращает объект `JavaClass` этого объекта `JavaObject`, т. е. возвращает объект `JavaClass`, который является представлением Java-класса Java-объекта, представленного указанным объектом `JavaObject`.

Порядок использования

Не следует путать JavaScript-функцию `getClass()` с методом `getClass`, которым обладают все Java-объекты. Точно так же не надо путать JavaScript-объект `JavaObject` с Java-классом `java.lang.Class`.

Рассмотрим Java-объект `Rectangle`, созданный следующим образом:

```
var r = new java.awt.Rectangle();
```

Здесь `r` – это переменная JavaScript, в которой хранится объект `JavaObject`. Обращение к JavaScript-функции `getClass()` дает в результате объект `JavaClass`, который представляет класс `java.awt.Rectangle`:

```
var c = getClass(r);
```

Убедиться в этом можно будет, сравнив этот объект `JavaClass` с `java.awt.Rectangle`:

```
if (c == java.awt.Rectangle) ...
```

Java-метод `getClass()` вызывается иначе и решает иные задачи:

```
c = r.getClass();
```

После исполнения этой строки кода в переменной `c` окажется объект `JavaObject`, который будет представлять объект `java.lang.Class`. Этот объект будет являться представлением Java-класса `java.awt.Rectangle`. За информацией об использовании класса `java.lang.Class` обращайтесь к документации по языку Java.

Подводя итоги, можно заметить, что следующее выражение всегда будет возвращать значение `true` для любого Java-объекта `o`:

```
(getClass(o.getClass()) == java.lang.Class)
```

См. также `JavaArray`, `JavaClass`, `JavaObject`, `JavaPackage`; главы 12 и 23

Global

ECMAScript v1

глобальный объект **Object→Global**

Синтаксис

```
this
```

Глобальные свойства

Глобальный объект – это не класс, поэтому для следующих глобальных свойств имеются отдельные справочные статьи под собственными именами. То есть подробные сведения о свойстве `undefined` можно найти под заголовком «undefined», а не «Global.undefined». Обратите внимание, что все переменные верхнего уровня также представляют собой свойства глобального объекта.

`Infinity`

Числовое значение, обозначающее положительную бесконечность.

`java`

Объект `JavaPackage`, который представляет иерархию *java.** пакетов.

`NaN`

Нечисловое значение.

`undefined`

Значение `undefined`.

Глобальные функции

Глобальный объект – это не класс, поэтому перечисленные далее глобальные функции не являются методами какого-либо объекта, и справочные статьи приведены под именами функций. Так, функция `parseInt()` подробно описывается под заголовком «parseInt()», а не «Global.parseInt()».

`decodeURI()`

Декодирует строку, закодированную с помощью функции `encodeURI()`.

`decodeURIComponent()`

Декодирует строку, закодированную с помощью функции `encodeURIComponent()`.

`encodeURI`

Кодирует URI, заменяя определенные символы управляющими последовательностями.

`encodeURIComponent`

Кодирует компонент URI, заменяя определенные символы управляющими последовательностями.

`escape()`

Кодирует строку, заменяя определенные символы управляющими последовательностями.

`eval()`

Вычисляет строку JavaScript-кода и возвращает результат.

`getClass()`

Возвращает объект `JavaClass` для объекта `JavaObject`.

`isFinite()`

Проверяет, является ли значение конечным числом.

`isNaN`

Проверяет, является ли значение нечисловым (`NaN`).

`parseFloat()`

Выбирает число из строки.

`parseInt()`

Выбирает целое из строки.

`unescape()`

Декодирует строку, закодированную вызовом `escape()`.

Глобальные объекты

В дополнение к перечисленным ранее глобальным свойствам и функциям, глобальный объект определяет свойства, ссылающиеся на все остальные предопределенные JavaScript-объекты. Все эти свойства являются функциями-конструкторами, определяющими классы, за исключением `Math`, которое представляет собой ссылку на объект, не являющийся конструктором.

`Array`

Конструктор `Array()`.

`Boolean`

Конструктор `Boolean()`.

`Date`

Конструктор `Date()`.

`Error`

Конструктор `Error()`.

`EvalError`

Конструктор `EvalError()`.

`Function`

Конструктор `Function()`.

`Math`

Ссылка на объект, определяющий математические функции.

`Number`

Конструктор `Number()`.

`Object`

Конструктор `Object()`.

`RangeError`

Конструктор `RangeError()`.

`ReferenceError`

Конструктор `ReferenceError()`.

`RegExp`

Конструктор `RegExp()`.

`String`

Конструктор `String()`.

`SyntaxError`

Конструктор `SyntaxError()`.

`TypeError`

Конструктор `TypeError()`.

`URIError`

Конструктор `URIError()`.

Описание

Глобальный объект – это предопределенный объект, который в JavaScript служит для размещения глобальных свойств и функций. Все остальные предопределенные объекты, функции и свойства доступны через глобальный объект. Глобальный объект не является свойством любого другого объекта, поэтому у него нет имени. (Заголовок справочной статьи выбран просто для удобства и не указывает на то, что глобальный объект имеет имя «Global».) В JavaScript-коде верхнего уровня можно ссылаться на глобальный объект посредством ключевого слова `this`. Однако этот способ обращения к глобальному объекту требуется редко, т. к. глобальный объект выступает в качестве начала цепочки областей видимости, поэтому поиск неуточненных имен переменных и функций выполняется среди свойств этого объекта. Когда JavaScript-код ссылается, например, на функцию `parseInt()`, он ссылается на свойство `parseInt` глобального объекта. Тот факт, что глобальный объект находится в начале цепочки областей видимости, означает также, что все переменные, объявленные в JavaScript-коде верхнего уровня, становятся свойствами глобального объекта.

Глобальный объект – это просто объект, а не класс. У него нет конструктора `Global()` и нет способа создать экземпляр нового глобального объекта.

Когда JavaScript-код встраивается в определенную среду, глобальному объекту обычно придаются дополнительные свойства, специфические для этой среды. На самом деле тип глобального объекта в стандарте ECMAScript не указан, и в конкретной реализации JavaScript в качестве глобального может выступать объект любого типа, если этот объект определяет перечисленные здесь основные свойства и функции. Например, в реализациях JavaScript, поддерживающих возможность взаимодействия с Java посредством механизма LiveConnect или подобных ему технологий, глобальному объекту придаются свойства `java` и `Packages`, а также упомянутый здесь метод `getClass()`. В клиентском JavaScript глобальным объектом является объект `Window`, представляющий окно веб-броузера, внутри которого исполняется JavaScript-код.

Пример

В базовом JavaScript ни одно из предопределенных свойств глобального объекта не является перечисляемым, благодаря чему можно получить список всех явно и неявно объявленных глобальных переменных с помощью следующего цикла `for/in`:

```
var variables = ""
for(var name in this)
    variables += name + "\n";
```

См. также Window (см. часть IV книги); глава 4

Infinity ECMAScript v1

числовое свойство, обозначающее бесконечность

Синтаксис

```
Infinity
```

Описание

Infinity – это глобальное свойство, содержащее специальное числовое значение, которое обозначает положительную бесконечность. Свойство Infinity не перечисляется циклами for/in и не может быть удалено с помощью оператора delete. Следует отметить, что Infinity не является константой и может быть установлено равным какому-либо другому значению, но лучше этого не делать. (В то же время Number.POSITIVE_INFINITY – это константа.)

См. также isFinite(), NaN, Number.POSITIVE_INFINITY

isFinite() ECMAScript v1

определяет, является ли число конечным

Синтаксис

```
isFinite(n)
```

Аргументы

n Проверяемое число.

Возвращаемое значение

Если *n* является конечным числом (или может быть преобразовано в него) – true, если *n* является нечислом (NaN) или плюс/минус бесконечностью – false.

См. также

Infinity, isNaN(), NaN, Number.NaN, Number.NEGATIVE_INFINITY, Number.POSITIVE_INFINITY

isNaN() ECMAScript v1

определяет, является ли аргумент нечисловым значением

Синтаксис

```
isNaN(x)
```

Аргументы

x Проверяемое значение.

Возвращаемое значение

Если *x* является специальным нечисловым значением (или может быть в него преобразовано) – true, если *x* является любым другим значением – false.

Описание

isNaN() проверяет свой аргумент, чтобы определить, не является ли он нечислом (NaN), т. е. некорректным числом (например, появившемся в результате деления на

ноль). Эта функция нужна, т. к. сравнение NaN с любым значением, включая и себя, всегда возвращает false, поэтому проверять на равенство NaN, используя оператор == или ===, нельзя.

Обычно функция isNaN() служит для проверки результатов, возвращаемых функциями parseFloat() и parseInt(), с целью определить, представляют ли эти результаты корректные числа. Функция isNaN() также может использоваться для проверки наличия арифметических ошибок, таких как деление на ноль.

Пример

```
isNaN(0);                 // Возвращает false
isNaN(0/0);               // Возвращает true
isNaN(parseInt("3"));     // Возвращает false
isNaN(parseInt("hello")); // Возвращает true
isNaN("3");               // Возвращает false
isNaN("hello");           // Возвращает true
isNaN(true);              // Возвращает false
isNaN(undefined);         // Возвращает true
```

См. также isFinite(), NaN, Number.NaN, parseFloat(), parseInt()

java LiveConnect

Объект JavaPackage, представляющий иерархию пакетов java.*

Синтаксис

java

Описание

В реализациях JavaScript, которые поддерживают механизм LiveConnect или другие технологии взаимодействия с Java, глобальное свойство java содержит ссылку на объект JavaPackage, который представляет иерархию пакетов *java.**. Наличие этого свойства означает, что, например, выражение java.util будет ссылаться на Java-пакет *java.util*. Что касается Java-пакетов, которые не попадают в иерархию *java.**, см. статью с описанием глобального свойства Property.

См. также JavaPackage, Packages; глава 12

JavaArray LiveConnect

представление Java-массивов в JavaScript

Синтаксис

массив_java.length // Длина массива

массив_java[index] // Чтение и запись элемента массива

Свойства

length

Целое число, доступное только для чтения и определяющее количество элементов в Java-массиве, который представляет объект JavaArray.

Описание

Объект JavaArray — это представление Java-массива, которое позволяет JavaScript-сценарию читать и записывать элементы массива с использованием привычного синтаксиса для работы с массивами, принятого в JavaScript. Кроме того, объект JavaArray обладает свойством length, которое содержит количество элементов в Java-массиве.

В процессе чтения/записи из/в элементы массива все необходимые преобразования данных между Java и JavaScript выполняются системой автоматически. Подробности см. в главе 12.

Порядок использования

Обратите внимание: Java-массивы имеют несколько существенных отличий от JavaScript-массивов. Во-первых, длина Java-массивов фиксирована и определяется при создании массива. По этой причине свойство length объекта JavaArray доступно только для чтения. Второе важное отличие заключается в том, что в языке Java массивы являются *типизированными* (т. е. все элементы массива должны иметь один и тот же тип данных). Попытка записать в элемент массива значение неверного типа приведет в JavaScript к ошибке или возбуждению исключения.

Пример

Пусть java.awt.Polygon — это объект JavaClass. Тогда объект JavaObject, который будет представлять экземпляр класса, можно создать следующим образом:

```
p = new java.awt.Polygon();
```

Объект p обладает свойствами xpoints и ypoints, которые являются объектами JavaArray, представляющими Java-массивы целых чисел. Вы можете инициализировать эти массивы из JavaScript-сценария следующим образом:

```
for(var i = 0; i < p.xpoints.length; i++)
    p.xpoints[i] = Math.round(Math.random()*100);
for(var i = 0; i < p.ypoints.length; i++)
    p.ypoints[i] = Math.round(Math.random()*100);
```

См. также getClass(), JavaClass, JavaObject, JavaPackage; глава 12

JavaClass LiveConnect

представление Java-класса в JavaScript

Синтаксис

```
класс_java.static_member  // Чтение и запись значения статического поля
                          // или метода в Java
new класс_java(...)       // Создает новый Java-объект
```

Свойства

Каждый объект JavaClass содержит свойства, имена которых совпадают с именами общедоступных статических полей и методов Java-класса, представляемых этим объектом. Указанные свойства позволяют читать и изменять значения статических полей класса и вызывать статические методы. Каждый объект JavaClass обладает различным набором свойств; для любого объекта JavaClass они могут быть перечислены с использованием цикла for/in.

Описание

Объект JavaClass – это представление Java-класса в сценариях JavaScript. Свойства объекта JavaClass являются отображением общедоступных статических полей и методов (иногда их называют полями и методами класса) представляемого класса. Обратите внимание: объект JavaClass не имеет полей, являющихся полями *экземпляра* Java-класса, – отдельные экземпляры Java-классов в JavaScript представляет объект JavaObject.

Объект JavaClass реализует функциональность механизма LiveConnect, что позволяет в JavaScript-программах выполнять чтение и запись значений статических переменных Java-классов с использованием привычного синтаксиса языка JavaScript. Кроме того, объект JavaClass обеспечивает возможность вызова статических методов Java-класса.

Чтобы позволить JavaScript-сценариям читать и записывать значения Java-переменных и Java-методов, объект JavaClass дает JavaScript-программам возможность создавать Java-объекты (представленные объектами JavaObject) с помощью ключевого слова new и вызова метода-конструктора объекта JavaClass.

Все преобразования типов данных, необходимость которых возникает в процессе взаимодействия между JavaScript и Java через объект JavaObject, выполняется в рамках технологии LiveConnect автоматически. Подробнее о преобразовании типов данных рассказывается в главе 12.

Порядок использования

Не забывайте, что язык программирования Java является *типизированным*. Это означает, что каждое поле объекта имеет определенный тип и в это поле может быть записано значение только определенного типа. Попытка записать в поле значение неверного типа приведет в JavaScript к появлению ошибки или возбуждению исключения. К тому же приведет попытка вызвать метод с аргументами неверных типов.

Пример

Пусть java.lang.System – это объект JavaClass, который представляет Java-класс java.lang.System. Тогда можно обращаться к статическим полям этого класса, например, так:

```
var java_console = java.lang.System.out;
```

Можно также вызывать статические методы этого класса, например:

```
var version = java.lang.System.getProperty("java.version");
```

Наконец, объект JavaClass позволяет создавать новые Java-объекты:

```
var java_date = new java.lang.Date();
```

См. также getClass(), JavaArray, JavaObject, JavaPackage; глава 12

JavaObject **LiveConnect**

представление Java-объекта в JavaScript

Синтаксис

Объект_java.член // Чтение/запись значения поля экземпляра или метода

Свойства

Каждый объект JavaObject содержит свойства, имена которых совпадают с именами общедоступных полей и методов экземпляра (но не статических полей и методов класса) представляемого им Java-объекта. Эти свойства позволяют читать и записывать значения общедоступных полей и вызывать общедоступные методы. Обычно перечень свойств, которыми обладает конкретный объект JavaObject, зависит от типа представляемого им Java-объекта. Перечислить свойства любого заданного объекта JavaObject можно с помощью цикла for/in.

Описание

Объект JavaObject – это представление Java-объекта в JavaScript-сценарии. Свойства объекта JavaObject являются представлением общедоступных полей и методов экземпляра, определенных в Java-объекте. (Статические поля и методы, или поля и методы класса, представляет объект JavaClass.)

Объект JavaObject реализует функциональность механизма LiveConnect, которая дает возможность выполнять в JavaScript-программах операции чтения и записи значений общедоступных полей Java-объектов с использованием привычного синтаксиса JavaScript. Кроме того, она предоставляет возможность вызывать общедоступные методы Java-объектов. Преобразования типов данных, необходимость которых возникает в процессе взаимодействия между JavaScript и Java, выполняется механизмом LiveConnect автоматически. Подробнее о преобразовании типов данных рассказывается в главе 12.

Порядок использования

Не забывайте, что язык программирования Java является *типизированным*. Это означает, что каждое поле объекта имеет определенный тип, и в это поле может быть записано значение только определенного типа. Например, поле width объекта java.awt.Rectangle – это целочисленное поле, и попытка записать в него строку приведет в JavaScript к появлению ошибки или возбуждению исключения.

Пример

Пусть java.awt.Rectangle – это объект JavaClass, который представляет класс jawa.awt.Rectangle. Тогда создать объект JavaObject, который будет представлять экземпляр этого класса, можно следующим образом:

```
var r = new java.awt.Rectangle(0,0,4,5);
```

После этого можно выполнять чтение переменных экземпляра этого объекта r, например:

```
var perimeter = 2*r.width + 2*r.height;
```

Кроме этого, можно устанавливать значения общедоступных переменных экземпляра объекта r с использованием синтаксиса JavaScript:

```
r.width = perimeter/4;
r.height = perimeter/4;
```

См. также getClass(), JavaArray, JavaClass, JavaPackage; глава 12

JavaPackage LiveConnect

представление Java-пакета в JavaScript

Синтаксис

```
пакет.имя_пакета      // Ссылка на другой объект JavaPackage
пакет.имя_класса      // Ссылка на объект JavaClass
```

Свойства

Свойствами объекта JavaPackage являются имена объектов JavaPackage и JavaClass, которые в нем содержатся. Каждый отдельный объект JavaPackage обладает отличающимся набором свойств. Следует отметить, что имена свойств объекта JavaPackage не поддаются перечислению в цикле for/in. Чтобы выяснить, какие пакеты и классы содержатся в каждом конкретном пакете, необходимо обратиться к справочным руководствам по языку программирования Java.

Описание

Объект JavaPackage – это представление Java-пакета в JavaScript-сценарии. В языке Java пакет – это коллекция родственных классов. В JavaScript объект JavaPackage может содержать классы (представленные объектами JavaClass) и другие объекты JavaPackage.

Глобальный объект имеет свойство JavaPackage с именем java, которое представляет иерархию пакетов *java.**. В данном объекте JavaPackage определены свойства, которые ссылаются на другие объекты JavaPackage. Например, java.lang и java.net ссылаются на пакеты *java.lang* и *java.net*.

Объект JavaPackage с именем java.awt содержит свойства с именами Frame и Button, которые являются ссылками на объекты JavaClass и представляют классы java.awt.Frame и java.awt.Button.

Глобальный объект определяет также свойство Packages, которое является корнем для всех свойств, представляющих корневые элементы всех известных иерархий пакетов. Например, выражение Packages.javax.swing ссылается на Java-пакет *javax.swing*.

Невозможно с помощью цикла for/in определить имена пакетов и классов, содержащихся внутри JavaPackage. Эта информация должна быть известна заранее. Найти ее можно в справочных руководствах по языку программирования Java или проследив иерархию Java-классов.

Дополнительные сведения о работе с Java-пакетами, Java-классами и Java-объектами приводятся в главе 12.

См. также java, JavaArray, JavaClass, JavaObject, Packages; глава 12

JSObject

см. описание объекта JSObject в части IV книги

Math ECMAScript v1

математические функции и константы

Синтаксис

```
Math.константа
Math.функция()
```

Константы

`Math.E`

Константа *e*, основание натуральных логарифмов.

`Math.LN10`

Натуральный логарифм числа 10.

`Math.LN2`

Натуральный логарифм числа 2.

`Math.LOG10E`

Десятичный логарифм числа *e*.

`Math.LOG2E`

Логарифм числа *e* по основанию 2.

`Math.PI`

Константа ?.

`Math.SQRT1_2`

Единица, деленная на корень квадратный из 2.

`Math.SQRT2`

Квадратный корень из 2.

Статические функции

`Math.abs()`

Вычисляет абсолютное значение.

`Math.acos()`

Вычисляет арккосинус.

`Math.asin()`

Вычисляет арксинус.

`Math.atan()`

Вычисляет арктангенс.

`Math.atan2()`

Вычисляет угол между осью X и точкой.

`Math.ceil()`

Округляет число вверх.

`Math.cos()`

Вычисляет косинус.

`Math.exp()`

Вычисляет степень числа *e*.

`Math.floor()`

Округляет число вниз.

`Math.log()`

Вычисляет натуральный логарифм.

`Math.max()`

Возвращает большее из двух чисел.

`Math.min()`

Возвращает меньшее из двух чисел.

`Math.pow()`

Вычисляет x в степени y.

`Math.random()`

Возвращает случайное число.

`Math.round()`

Округляет до ближайшего целого.

`Math.sin()`

Вычисляет синус.

`Math.sqrt()`

Вычисляет квадратный корень.

`Math.tan()`

Вычисляет тангенс.

Описание

`Math` – это объект, определяющий свойства, которые ссылаются на полезные математические функции и константы. Эти функции и константы вызываются с помощью следующего синтаксиса:

```
y = Math.sin(x);
area = radius * radius * Math.PI;
```

Здесь `Math` – это не класс объектов, как `Date` и `String`. Конструктора `Math()` у объекта `Math` нет, поэтому такие функции, как `Math.sin()`, – это просто функции, а не методы объекта.

См. также

`Number`

Math.abs() ECMAScript v1

вычисляет абсолютное значение

Синтаксис

`Math.abs(`*x*`)`

Аргументы

x Любое число.

Возвращаемое значение

Абсолютное значение *x*.

Math.acos() ECMAScript v1

вычисляет арккосинус

Синтаксис

`Math.acos(`*x*`)`

Аргументы

x Число от -1,0 до 1,0.

Возвращаемое значение

Арккосинус указанного числа *x*. Возвращаемое значение может находиться в интервале от 0 до π радиан.

Math.asin() ECMAScript v1

вычисляет арксинус

Синтаксис

```
Math.asin(x)
```

Аргументы

x Число от -1,0 до 1,0.

Возвращаемое значение

Арксинус указанного значения *x*. Это возвращаемое значение может находиться в интервале от $-\pi/2$ до $\pi/2$ радиан.

Math.atan() ECMAScript v1

вычисляет арктангенс

Синтаксис

```
Math.atan(x)
```

Аргументы

x Любое число.

Возвращаемое значение

Арктангенс указанного значения *x*. Возвращаемое значение может находиться в интервале от $-\pi/2$ до $\pi/2$ радиан.

Math.atan2() ECMAScript v1

вычисляет угол между осью X и точкой

Синтаксис

```
Math.atan2(y, x)
```

Аргументы

y Координата Y точки.

x Координата X точки.

Возвращаемое значение

Значение, лежащее между $-\pi$ и π радиан и указывающее на угол по направлению, обратному часовой стрелке, между положительной осью X и точкой (*x*,*y*).

Описание

Функция `Math.atan2()` вычисляет арктангенс отношения *y*/*x*. Аргумент *y* может рассматриваться как координата Y (или «рост») точки, а аргумент *x* – как координата X (или «пробег») точки. Обратите внимание на необычный порядок следования аргументов этой функции: координата Y передается до координаты X.

Math.ceil() ECMAScript v1

округляет число вверх

Синтаксис

```
Math.ceil(x)
```

Аргументы

x Числовое значение или выражение.

Возвращаемое значение

Ближайшее целое, большее или равное *x*.

Описание

Функция `Math.ceil()` вычисляет наименьшее целое, т. е. возвращает ближайшее целое, большее или равное аргументу функции. Функция `Math.ceil()` отличается от `Math.round()` тем, что округляет всегда вверх, а не к ближайшему целому. Обратите внимание также, что `Math.ceil()` не округляет отрицательные числа к большим по абсолютному значению отрицательным целым; функция округляет их по направлению к нулю.

Пример

```
a = Math.ceil(1.99);  // Результат равен 2.0
b = Math.ceil(1.01);  // Результат равен 2.0
c = Math.ceil(1.0);   // Результат равен 1.0
d = Math.ceil(-1.99); // Результат равен -1.0
```

Math.cos() ECMAScript v1

вычисляет косинус

Синтаксис

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

Аргументы

x Угол в радианах. Чтобы преобразовать градусы в радианы, нужно умножить значение в градусах на 0,017453293 ($2\pi/360$).

Возвращаемое значение

Косинус указанного значения *x*. Это возвращаемое значение может находиться в интервале от -1,0 до 1,0.

Math.E ECMAScript v1

математическая константа e

Синтаксис

```
Math.E
```

Описание

`Math.E` — это математическая константа *e*, база натуральных логарифмов, приблизительно равная 2,71828.

Math.exp() ECMAScript v1

вычисляет e^x

Синтаксис

```
Math.exp(x)
```

Аргументы

x Число или выражение, которое должно использоваться как экспонента.

Возвращаемое значение

e^x – это число e, возведенное в степень указанной экспоненты x, где e – это основание натуральных логарифмов, примерно равное 2,71828.

Math.floor() ECMAScript v1

округляет число вниз

Синтаксис

```
Math.floor(x)
```

Аргументы

x Числовое значение или выражение.

Возвращаемое значение

Ближайшее целое, меньшее или равное *x*.

Описание

Округление вниз, другими словами, функция возвращает ближайшее целое значение, меньшее или равное аргументу функции.

Функция `Math.floor()` округляет вещественное число вниз, в отличие от функции `Math.round()`, выполняющей округление до ближайшего целого. Обратите внимание: `Math.floor()` округляет отрицательные числа вниз (т. е. дальше от нуля), а не вверх (т. е. ближе к нулю).

Пример

```
a = Math.floor(1.99);  // Результат равен 1.0
b = Math.floor(1.01);  // Результат равен 1.0
c = Math.floor(1.0);   // Результат равен 1.0
d = Math.floor(-1.01); // Результат равен -2.0
```

Math.LN10 ECMAScript v1

математическая константа $\log_e 10$

Синтаксис

```
Math.LN10
```

Описание

`Math.LN10` – это $\log_e 10$, натуральный логарифм числа 10. Эта константа имеет значение, приблизительно равное 2,3025850929940459011.

Math.LN2 ECMAScript v1

математическая константа $\log_e 2$

Синтаксис

```
Math.LN2
```

Описание

`Math.LN2` – это $\log_e 2$, натуральный логарифм числа 2. Эта константа имеет значение, приблизительно равное 0,6931471805599452862З.

Math.log() ECMAScript v1

вычисляет натуральный логарифм

Синтаксис

```
Math.log(x)
```

Аргументы

x Любое числовое значение, большее или равное нулю.

Возвращаемое значение

Натуральный логарифм *x*.

Описание

`Math.log()` вычисляет натуральный логарифм своего аргумента. Аргумент должен быть больше нуля.

Логарифмы числа по основанию 10 и 2 можно вычислить по следующим формулам:

$$\log_{10} x = \log_{10} e \cdot \log_e x$$
$$\log_2 x = \log_2 e \cdot \log_e x$$

Эти формулы транслируются в следующие JavaScript-функции:

```
function log10(x) { return Math.LOG10E * Math.log(x); }
function log2(x) { return Math.LOG2E * Math.log(x); }
```

Math.LOG10E ECMAScript v1

математическая константа $\log_{10} e$

Синтаксис

```
Math.LOG10E
```

Описание

`Math.LOG10E` – это $\log_{10} e$, логарифм по основанию 10 константы e. Его значение приблизительно равно 0,4342944819032518166З.

Math.LOG2E ECMAScript v1

математическая константа $\log_2 e$

Синтаксис

```
Math.LOG2E
```

Описание

`Math.LOG2E` — это $\log_2 e$, логарифм по основанию 2 константы e. Его значение приблизительно равно 1,442695040888963387.

Math.max()

ECMAScript v1; расширен в ECMAScript v3

возвращает наибольший аргумент

Синтаксис

`Math.max(`*аргументы...*`)`

Аргументы

аргументы...

Ноль или более значений. До выхода стандарта ECMAScript v3 этот метод мог принимать ровно два аргумента.

Возвращаемое значение

Наибольший из аргументов. Возвращает `-Infinity`, если аргументов нет. Возвращает `NaN`, если какой-либо из аргументов равен `NaN` или является нечисловым значением, которое не может быть преобразовано в число.

Math.min()

ECMAScript v1; расширен в ECMAScript v3

возвращает наименьший аргумент

Синтаксис

`Math.min(`*аргументы...*`)`

Аргументы

аргументы...

Любое количество аргументов. До выхода стандарта ECMAScript v3 эта функция принимала ровно два аргумента.

Возвращаемое значение

Наименьший из указанных аргументов. Возвращает `Infinity`, если аргументов нет. Возвращает `NaN`, если какой-либо из аргументов представляет собой значение `NaN` или нечисловое значение и не может быть преобразован в число.

Math.PI

ECMAScript v1

математическая константа π

Синтаксис

`Math.PI`

Описание

`Math.PI` — это константа π, т. е. отношение длины окружности к ее диаметру. Имеет значение, примерно равное 3,14159265358979.

Math.pow() ECMAScript v1

вычисляет x^y

Синтаксис

```
Math.pow(x, y)
```

Аргументы

x Число, которое должно быть возведено в степень.

y Степень, в которую должно быть возведено число *x*.

Возвращаемое значение

x в степени *y* (x^y).

Описание

`Math.pow()` вычисляет *x* в степени *y*. Значения *x* и *y* могут быть любыми. Однако если результат является мнимым или комплексным числом, `Math.pow()` возвращает `NaN`. На практике это означает, что если *x* отрицательно, то *y* должно быть положительным или отрицательным целым. Также имейте в виду, что большие экспоненты легко приводят к вещественному переполнению и возвращают значение `Infinity`.

Math.random() ECMAScript v1

возвращает псевдослучайное число

Синтаксис

```
Math.random()
```

Возвращаемое значение

Псевдослучайное число от 0,0 до 1,0.

Math.round() ECMAScript v1

округляет число до ближайшего целого

Синтаксис

```
Math.round(x)
```

Аргументы

x Любое число.

Возвращаемое значение

Целое, ближайшее к *x*.

Описание

`Math.round()` округляет аргумент вверх или вниз до ближайшего целого. Число 0,5 округляется вверх. Например, число 2,5 округляется до 3, а число –2,5 округляется до –2.

Math.sin() ECMAScript v1

вычисляет синус

Синтаксис

```
Math.sin(x)
```

Аргументы

x Угол в радианах. Для преобразования градусов в радианы умножьте число на 0,017453293 (2π/360).

Возвращаемое значение

Синус *x* – число в диапазоне от -1,0 до 1,0.

Math.sqrt() ECMAScript v1

вычисляет квадратный корень

Синтаксис

```
Math.sqrt(x)
```

Аргументы

x Числовое значение, большее или равное 0.

Возвращаемое значение

Квадратный корень из *x*. Возвращает `NaN`, если *x* меньше нуля.

Описание

`Math.sqrt()` вычисляет квадратный корень из числа. Следует заметить, что произвольные корни из чисел можно вычислять посредством функции `Math.pow()`. Например:

```
Math.cuberoot = function(x){ return Math.pow(x,1/3); }
Math.cuberoot(8); // Возвращает 2
```

Math.SQRT1_2 ECMAScript v1

математическая константа 1/√2

Синтаксис

```
Math.SQRT1_2
```

Описание

`Math.SQRT1_2` – это $1/\sqrt{2}$, величина, обратная корню квадратному из 2. Эта константа примерно равна 0,7071067811865476.

Math.SQRT2 ECMAScript v1

математическая константа √2

Синтаксис

```
Math.SQRT2
```

Описание

`Math.SQRT2` – это $\sqrt{2}$, корень квадратный из 2. Эта константа имеет значение, примерно равное 1,414213562373095.

Math.tan() ECMAScript v1

вычисляет тангенс

Синтаксис

```
Math.tan(x)
```

Аргументы

x Угол, измеряемый в радианах. Чтобы преобразовать градусы в радианы, нужно умножить значение в градусах на 0,017453293 (2π/360).

Возвращаемое значение

Тангенс указанного угла *x*.

NaN ECMAScript v1

свойство «нечисло»

Синтаксис

```
NaN
```

Описание

`NaN` – это глобальное свойство, ссылающееся на специальное числовое значение «нечисло». Свойство `NaN` не перечисляется циклами `for/in` и не может быть удалено оператором `delete`. Обратите внимание: `NaN` – это не константа, и оно может быть установлено в любое значение, но лучше этого не делать.

Определить, является ли значение нечислом, можно с помощью функции `isNaN()`, т. к. `NaN` всегда при сравнении оказывается не равным любой другой величине, включая самого себя!

См. также

`Infinity`, `isNaN()`, `Number.NaN`

Number ECMAScript v1

поддержка чисел **Object→Number**

Конструктор

```
new Number(значение)
Number(значение)
```

Аргументы

значение

Числовое значение создаваемого объекта `Number` или значение, которое может быть преобразовано в число.

Возвращаемое значение

Когда функция `Number()` используется в качестве конструктора (с оператором `new`), она возвращает вновь созданный объект `Number`. Когда функция `Number()` вызывается как функция (без оператора `new`), она преобразует свой аргумент в элементарное числовое значение и возвращает это значение (или `NaN`, если преобразование не удалось).

Константы

`Number.MAX_VALUE`

Наибольшее представимое число.

`Number.MIN_VALUE`

Наименьшее представимое число.

`Number.NaN`

Нечисло.

`Number.NEGATIVE_INFINITY`

Отрицательная бесконечность, возвращается в случае переполнения.

`Number.POSITIVE_INFINITY`

Положительная бесконечность; возвращается при переполнении.

Методы

`toString()`

Преобразует число в строку в указанной системе счисления.

`toLocaleString()`

Преобразует число в строку, руководствуясь локальными соглашениями о форматировании чисел.

`toFixed()`

Преобразует число в строку, содержащую указанное число цифр после десятичной точки.

`toExponential()`

Преобразует числа в строки в экспоненциальной нотации с указанным количеством цифр после десятичной точки.

`toPrecision()`

Преобразует число в строку, записывая в нее указанное количество значащих цифр. Нотация экспоненциальная или с фиксированной точкой в зависимости от размера числа и заданного количества значащих цифр.

`valueOf()`

Возвращает элементарное числовое значение объекта `Number`.

Описание

Числа – это базовый элементарный тип данных в JavaScript. В JavaScript поддерживается также объект `Number`, представляющий собой обертку вокруг элементарного числового значения. Интерпретатор JavaScript при необходимости автоматически выполняет преобразование между элементарной и объектной формами. Существует возможность явно создать объект `Number` посредством конструктора `Number()`, хотя в этом редко возникает необходимость.

Конструктор `Number()` может также вызываться как функция преобразования (без оператора `new`). В этом случае она пытается преобразовать свой аргумент в число и возвращает элементарное числовое значение (или `NaN`), полученное при преобразовании.

Конструктор `Number()` используется также для размещения пяти полезных числовых констант: максимального и минимального представимых чисел, положительной и отрицательной бесконечности, а также специального значения «нечисло». Обрати-

те внимание: эти значения представляют собой свойства самой функции-конструктора Number(), а не отдельных числовых объектов. Например, свойство MAX_VALUE можно использовать следующим образом:

```
var biggest = Number.MAX_VALUE
```

А такая запись *неверна*:

```
var n = new Number(2);
var biggest = n.MAX_VALUE
```

В то же время toString() и другие методы объекта Number являются методами каждого объекта Number, а не функции-конструктора Number(). Как уже говорилось, JavaScript по необходимости автоматически выполняет преобразования между элементарными числовыми значениями и объектами Number. То есть методы класса Number могут работать с элементарными числовыми значениями так же, как с объектами Number:

```
var value = 1234;
var binary_value = n.toString(2);
```

См. также Infinity, Math, NaN

Number.MAX_VALUE ECMAScript v1

максимальное числовое значение

Синтаксис

Number.MAX_VALUE

Описание

Number.MAX_VALUE – это наибольшее число, представимое в JavaScript. Его значение примерно равно 1,79E+308.

Number.MIN_VALUE ECMAScript v1

минимальное числовое значение

Синтаксис

Number.MIN_VALUE

Описание

Number.MIN_VALUE – это наименьшее число (ближайшее к нулю, а не самое отрицательное), представимое в JavaScript. Его значение примерно равно 5E–324.

Number.NAN ECMAScript v1

специальное нечисловое значение

Синтаксис

Number.NaN

Описание

Number.NaN – это специальное значение, указывающее, что результат некоторой математической операции (например, извлечения квадратного корня из отрицательного

числа) не является числом. Функции parseInt() и parseFloat() возвращают это значение, когда не могут преобразовать указанную строку в число; программист может использовать Number.NaN аналогичным образом, чтобы указать на ошибочное условие для какой-либо функции, обычно возвращающей допустимое число.

JavaScript выводит значение Number.NaN как NaN. Обратите внимание: при сравнении значение NaN всегда не равно любому другому числу, включая само значение NaN. Следовательно, невозможно проверить значение на «нечисло», сравнив его с Number.NaN. Для этого предназначена функция isNaN(). В стандарте ECMAScript v1 и более поздних версиях вместо Number.NaN допускается использование предопределенной глобальной константы NaN.

См. также isNaN(), NaN

Number.NEGATIVE_INFINITY ECMAScript v1

отрицательная бесконечность

Синтаксис

Number.NEGATIVE_INFINITY

Описание

Number.NEGATIVE_INFINITY – специальное числовое значение, возвращаемое, если арифметическая операция или математическая функция генерирует отрицательное число, большее чем максимальное представимое в JavaScript число (т. е. отрицательное число, меньшее чем -Number.MAX_VALUE).

JavaScript выводит значение NEGATIVE_INFINITY как -Infinity. Это значение математически ведет себя как бесконечность. Например, все что угодно, умноженное на бесконечность, является бесконечностью, а все, деленное на бесконечность, – нулем. В ECMAScript v1 и более поздних версиях можно также использовать предопределенную глобальную константу -Infinity вместо Number.NEGATIVE_INFINITY.

См. также Infinity, isFinite()

Number.POSITIVE_INFINITY ECMAScript v1

бесконечность

Синтаксис

Number.POSITIVE_INFINITY

Описание

Number.POSITIVE_INFINITY – это специальное числовое значение, возвращаемое, когда арифметическая операция или математическая функция приводит к переполнению или генерирует значение, превосходящее максимальное представимое в JavaScript число (т. е. Number.MAX_VALUE). Обратите внимание: если происходит потеря значимости или число становится меньше, чем Number.MIN_VALUE, JavaScript преобразует его в ноль.

JavaScript выводит значение POSITIVE_INFINITY как Infinity. Это значение ведет себя математически так же, как бесконечность. Например, что-либо, умноженное на бесконечность, – это бесконечность, а что-либо, деленное на бесконечность, – ноль.

В ECMAScript v1 и более поздних версиях вместо `Number.POSITIVE_INFINITY` можно также использовать предопределенную глобальную константу `Infinity`.

См. также `Infinity`, `isFinite()`

Number.toExponential() ECMAScript v3

форматирует число в экспоненциальную форму представления

Синтаксис

число`.toExponential(`*цифры*`)`

Аргументы

цифры

Количество цифр после десятичной точки. Может быть значением от 0 до 20 включительно, конкретные реализации могут поддерживать больший диапазон значений. Если аргумент отсутствует, то цифр будет столько, сколько необходимо.

Возвращаемое значение

Строковое представление числа в экспоненциальной нотации с одной цифрой перед десятичной точкой и с количеством цифр, указанным в аргументе *цифры*, после нее. Дробная часть, если это необходимо, округляется или дополняется нулями, чтобы она имела указанную длину.

Исключения

`RangeError`

Генерируется, если аргумент *цифры* слишком велик или слишком мал. Значения между 0 и 20 включительно не приводят к ошибке `RangeError`. Реализациям также разрешено поддерживать большее или меньшее количество цифр.

`TypeError`

Генерируется, если метод вызывается для объекта, не являющегося объектом `Number`.

Пример

```
var n = 12345.6789;
n.toExponential(1);  // Возвращает 1.2e+4
n.toExponential(5);  // Возвращает 1.23457e+4
n.toExponential(10); // Возвращает 1.2345678900e+4
n.toExponential();   // Возвращает 1.23456789e+4
```

См. также

`Number.toFixed()`, `Number.toLocaleString()`, `Number.toPrecision()`, `Number.toString()`

Number.toFixed() ECMAScript v3

форматирует число в форму представления с фиксированной точкой

Синтаксис

число`.toFixed(`*цифры*`)`

Аргументы

цифры

Количество цифр после десятичной точки; оно может быть значением от 0 до 20 включительно; конкретные реализации могут поддерживать больший диапазон значений. Если этот аргумент отсутствует, он считается равным 0.

Возвращаемое значение

Строковое представление числа, которое не использует экспоненциальную нотацию и в котором количество цифр после десятичной точки равно аргументу *цифры*. По необходимости число округляется, а дробная часть дополняется нулями до указанной длины. Если число больше, чем 1e+21, этот метод вызывает функцию `Number.toString()` и возвращает строку в экспоненциальной нотации.

Исключения

`RangeError`

Генерируется, если аргумент цифры слишком велик или слишком мал. Значения от 0 до 20 включительно не приводят к исключению `RangeError`. Конкретные реализации допускают большие или меньшие значения.

`TypeError`

Генерируется, если метод вызывается для объекта, не являющегося объектом `Number`.

Пример

```
var n = 12345.6789;
n.toFixed();           // Возвращает 12346: обратите внимание на округление
                       // и отсутствие дробной части
n.toFixed(1);          // Возвращает 12345.7: обратите внимание на округление
n.toFixed(6);          // Возвращает 12345.678900: обратите внимание
                       // на добавление нулей
(1.23e+20).toFixed(2); // Возвращает 123000000000000000000.00
(1.23e-10).toFixed(2)  // Возвращает 0.00
```

См. также

`Number.toExponential()`, `Number.toLocaleString()`, `Number.toPrecision()`, `Number. toString()`

Number.toLocaleString() ECMAScript v3

преобразует число в строку в соответствии с региональными настройками

Синтаксис

число`.toLocaleString()`

Возвращаемое значение

Зависящее от реализации строковое представление числа, отформатированное в соответствии с региональными настройками, на которое могут влиять, например, символы пунктуации, выступающие в качестве десятичной точки и разделителя тысяч.

Исключения

`TypeError`

Генерируется, если метод вызван для объекта, не являющегося объектом `Number`.

См. также

Number.toExponential(), Number.toFixed(), Number.toPrecision(), Number.toString()

Number.toPrecision() ECMAScript v3

форматирует значащие цифры числа

Синтаксис

число.toPrecision(*точность*)

Аргументы

точность

Количество значащих цифр в возвращаемой строке. Оно может быть значением от 1 до 21 включительно. Конкретные реализации могут поддерживать большие и меньшие значения *точности*. Если этот аргумент отсутствует, для преобразования в десятичное число используется метод toString().

Возвращаемое значение

Строковое представление *числа*, содержащее количество значащих цифр, определяемое аргументом *точность*. Если *точность* достаточна для включения всех цифр целой части *числа*, возвращаемая строка записывается в нотации с фиксированной точкой. В противном случае запись осуществляется в экспоненциальной нотации с одной цифрой перед десятичной точкой и количеством цифр *точность*-1 после десятичной точки. Число при необходимости округляется или дополняется нулями.

Исключения

RangeError

Генерируется, если аргумент *точность* слишком мал или слишком велик. Значения от 1 до 21 включительно не приводят к исключению RangeError. Конкретные реализации могут поддерживать большие и меньшие значения.

TypeError

Генерируется, если метод вызывается для объекта, не являющегося объектом Number.

Пример

```
var n = 12345.6789;
n.toPrecision(1);  // Возвращает 1e+4
n.toPrecision(3);  // Возвращает 1.23e+4
n.toPrecision(5);  // Возвращает 12346: обратите внимание на округление
n.toPrecision(10); // Возвращает 12345.67890: обратите внимание на добавление нуля
```

См. также

Number.toExponential(), Number.toFixed(), Number.toLocaleString(), Number.toString()

Number.toString() ECMAScript v3

преобразует число в строку **переопределяет Object.toString()**

Синтаксис

число.toString(*основание*)

Аргументы

основание

Необязательный аргумент, задающий основание системы счисления (между 2 и 36), в которой должно быть представлено число. Если аргумент отсутствует, то основание равно 10. Следует заметить, что спецификация ECMAScript разрешает реализации возвращать любое значение, если этот аргумент равен любому значению, отличному от 10.

Возвращаемое значение

Строковое представление числа.

Исключения

`TypeError`

Генерируется, если метод вызывается для объекта, не являющегося объектом `Number`.

Описание

Метод `toString()` объекта `Number` преобразует число в строку. Если аргумент *основание* опущен или указано значение 10, число преобразуется в строку по основанию 10. Несмотря на то, что спецификация ECMAScript не требует от реализаций корректно реагировать на любые другие значения аргумента *основание*, тем не менее все распространенные реализации принимают значения основания в диапазоне от 2 до 36.

См. также

`Number.toExponential()`, `Number.toFixed()`, `Number.toLocaleString()`, `Number.toPrecision()`

Number.valueOf() ECMAScript v1

преобразует число в строку **переопределяет Object.valueOf()**

Синтаксис

число`.valueOf()`

Возвращаемое значение

Элементарное числовое значение объекта `Number`. В явном вызове этого метода редко возникает необходимость.

Исключения

`TypeError`

Генерируется, если метод вызывается для объекта, не являющегося объектом `Number`.

См. также `Object.valueOf()`

Object ECMAScript v1

надкласс, реализующий общие возможности всех JavaScript-объектов

Конструктор

`new Object()`

`new Object(`*значение*`)`

Аргументы

значение

В этом необязательном аргументе указано элементарное JavaScript-значение – число, логическое значение или строка, которое должно быть преобразовано в объект `Number`, `Boolean` или `String`.

Возвращаемое значение

Если передан аргумент *значение*, конструктор возвращает вновь созданный экземпляр `Object`. Если указан аргумент *значение* элементарного типа, конструктор создает объект-обертку `Number`, `Boolean` или `String` для указанного элементарного значения. Если конструктор `Object()` вызывается как функция (без оператора `new`), он ведет себя точно так же, как при вызове с оператором `new`.

Свойства

`constructor`

Ссылка на JavaScript-функцию, которая была конструктором объекта.

Методы

`hasOwnProperty()`

Проверяет, имеет ли объект локально определенное (не унаследованное) свойство с указанным именем.

`isPrototypeOf()`

Проверяет, является ли данный объект прототипом для указанного объекта.

`propertyIsEnumerable()`

Проверяет, существует ли свойство с указанным именем и будет ли оно перечислено циклом `for/in`.

`toLocaleString()`

Возвращает локализованное строковое представление объекта. Реализация по умолчанию этого метода просто вызывает метод `toString()`, но подклассы могут переопределять его для выполнения локализации.

`toString()`

Возвращает строковое представление объекта. Реализация этого метода в классе `Object` является очень общей и возвращает не много полезной информации. Подклассы `Object` обычно переопределяют этот метод собственным методом `toString()`, возвращающим более полезный результат.

`valueOf()`

Возвращает элементарное значение объекта, если оно существует. Для объектов типа `Object` этот метод просто возвращает сам объект. Подклассы `Object`, такие как `Number` и `Boolean`, переопределяют этот метод, чтобы можно было получить элементарное значение, связанное с объектом.

Описание

Класс `Object` – это встроенный тип данных языка JavaScript. Он выступает в качестве надкласса для всех остальных JavaScript-объектов; следовательно, методы и поведение класса `Object` наследуются всеми остальными объектами. О базовом поведении JavaScript-объектов рассказывается в главе 7.

В дополнение к показанному ранее конструктору Object(), объекты могут создаваться и инициализироваться с помощью синтаксиса объектных литералов, описанного в главе 7.

См. также Array, Boolean, Function, Function.prototype, Number, String; глава 7

Object.constructor ECMAScript v1

функция-конструктор объекта

Синтаксис

объект.constructor

Описание

Свойство constructor любого объекта – это ссылка на функцию, являющуюся конструктором этого объекта. Например, если массив a создается с помощью конструктора Array(), то свойство a.constructor будет равно Array:

```
a = new Array(1,2,3);  // Создает объект
a.constructor == Array // Равно true
```

Одно из распространенных применений свойства constructor состоит в определении типа неизвестных объектов. Оператор typeof позволяет определить, является ли неизвестный объект элементарным значением или объектом. Если это объект, то посредством свойства constructor можно определить, каков тип этого объекта. Например, следующая функция позволяет узнать, является ли данное значение массивом:

```
function isArray(x) {
    return ((typeof x == "object") && (x.constructor == Array));
}
```

Однако следует отметить, что хотя этот прием эффективен для объектов, встроенных в базовый JavaScript, его работа с «хост-объектами» клиентского JavaScript, такими как объект Window, не гарантируется. Реализация метода Object.toString(), предлагаемая по умолчанию, дает еще один способ определения типа неизвестного объекта.

См. также Object.toString()

Object.hasOwnProperty() ECMAScript v3

проверяет, является ли свойство унаследованным

Синтаксис

объект.hasOwnProperty(*имя_свойства*)

Аргументы

имя_свойства

Строка, содержащая имя свойства объекта.

Возвращаемое значение

Возвращает true, если объект имеет неунаследованное свойство с именем, заданным в *имени_свойства*. Возвращает false, если объект не имеет свойства с указанным именем или если он наследует это свойство от своего объекта-прототипа.

Описание

В главе 9 говорится, что JavaScript-объекты могут иметь собственные свойства, а также наследовать свойства от своих объектов-прототипов. Метод `hasOwnProperty()` предоставляет способ, позволяющий установить различия между унаследованными свойствами и не унаследованными локальными свойствами.

Пример

```
var o = new Object();         // Создаем объект
o.x = 3.14;                   // Определяем неунаследованное локальное свойство
o.hasOwnProperty("x");        // Возвращает true: x - это локальное свойство o
o.hasOwnProperty("y");        // Возвращает false: o не имеет свойства y
o.hasOwnProperty("toString"); // Возвращает false: свойство toString унаследовано
```

См. также `Function.prototype`, `Object.propertyIsEnumerable()`; глава 9

Object.isPrototypeOf() ECMAScript v3

проверяет, является ли один объект прототипом другого объекта

Синтаксис

объект`.isPrototypeOf(`*o*`)`

Аргументы

o Любой объект.

Возвращаемое значение

Возвращает `true`, если данный *объект* представляет собой прототип объекта *o*. Возвращает `false`, если *o* не является объектом или если данный *объект* не является прототипом объекта *o*.

Описание

Как объяснялось в главе 9, JavaScript-объекты наследуют свойства от своих объектов-прототипов. К прототипу объекта можно обращаться с помощью свойства `prototype` функции-конструктора, которая использовалась для создания и инициализации объекта. Метод `isPrototypeOf()` предоставляет возможность определить, является ли один объект прототипом другого. Этот прием может применяться для определения класса объекта.

Пример

```
var o = new Object();                          // Создание объекта
Object.prototype.isPrototypeOf(o)              // true: o - это объект
Function.prototype.isPrototypeOf(o.toString); // true: toString - это функция
Array.prototype.isPrototypeOf([1,2,3]);        // true: [1,2,3] - это массив
// Ту же проверку можно выполнить другим способом
(o.constructor == Object);               // true: o был создан с помощью конструктора Object()
(o.toString.constructor == Function);    // true: o.toString - это функция
// Объекты-прототипы сами имеют прототипы. Следующий вызов возвращает
// true, показывая, что объекты-функции наследуют свойства
// от Function.prototype, а также от Object.prototype.
Object.prototype.isPrototypeOf(Function.prototype);
```

См. также `Function.prototype`, `Object.constructor`; глава 9

Object.propertyIsEnumerable() ECMAScript v3

проверяет, будет ли свойство видимо для цикла for/in

Синтаксис

объект.propertyIsEnumerable(*имя_свойства*)

Аргументы

имя_свойства

Строка, содержащая имя свойства объекта.

Возвращаемое значение

Возвращает `true`, если у *объекта* есть не унаследованное свойство с именем, указанным в аргументе *имя_свойства*, и если это свойство «перечисляемое», т. е. оно может быть перечислено циклом `for/in` для *объекта*.

Описание

Инструкция `for/in` выполняет цикл по «перечисляемым» свойствам объекта. Однако не все свойства объекта являются перечисляемыми: свойства, добавленные в объект JavaScript-кодом, перечисляемы, а предопределенные свойства (например, методы) встроенных объектов обычно не перечисляемы. Метод `propertyIsEnumerable()` служит для установления различия между перечисляемыми и не перечисляемыми свойствами. Однако следует заметить: спецификация ECMAScript утверждает, что `propertyIsEnumerable()` не проверяет цепочку прототипов, т. е. этот метод годится только для локальных свойств объекта и не предоставляет способа для проверки перечисляемости унаследованных свойств.

Пример

```
var o = new Object();                 // Создает объект
o.x = 3.14;                           // Определяет свойство
o.propertyIsEnumerable("x");          // true: свойство x локальное и перечисляемое
o.propertyIsEnumerable("y");          // false: o не имеет свойства y
o.propertyIsEnumerable("toString");   // false: свойство toString является унаследованным
Object.prototype.propertyIsEnumerable("toString"); // false: неперечисляемое свойство
```

См. также `Function.prototype`, `Object.hasOwnProperty()`; глава 7

Object.toLocaleString() ECMAScript v3

возвращает локализованное строковое представление объекта

Синтаксис

объект.toString()

Возвращаемое значение

Строковое представление объекта.

Описание

Этот метод предназначен для получения строкового представления объекта, локализованного в соответствии с текущими региональными настройками. Метод `toLocaleString()`, предоставляемый по умолчанию классом `Object`, просто вызывает метод `toString()` и возвращает полученную от него нелокализованную строку. Однако обра-

тите внимание, что другие классы, в том числе Array, Date и Number, определяют собственные версии этого метода для локализованного преобразования в строку. Вы также можете переопределить этот метод собственными классами.

См. также

Array.toLocaleString(), Date.toLocaleString(), Number.toLocaleString(), Object.toString()

Object.toString() ECMAScript v1

возвращает строковое представление объекта

Синтаксис

объект.toString()

Возвращаемое значение

Строка, представляющая объект.

Описание

Метод toString() не относится к тем, которые часто вызываются в JavaScript-программах явно. Программист определяет этот метод в своих объектах, а система вызывает метод, когда требуется преобразовать объект в строку.

JavaScript вызывает метод toString() для преобразования объекта в строку всякий раз, когда объект используется в строковом контексте. Например, если объект преобразуется в строку при передаче в функцию, требующую строкового аргумента:

```
alert(my_object);
```

Подобным же образом объекты преобразуются в строки, когда они конкатенируются со строками с помощью оператора +:

```
var msg = 'Мой объект: ' + my_object;
```

Метод toString() вызывается без аргументов и должен возвращать строку. Для того чтобы от возвращаемой строки была какая-то польза, эта строка должна каким-либо образом базироваться на значении объекта, для которого был вызван метод.

Определяя в JavaScript специальный класс, целесообразно определить для него метод toString(). Если этого не сделать, объект наследует метод toString(), определенный по умолчанию в классе Object. Этот стандартный метод возвращает строку в формате:

```
[object класс]
```

где *класс* – это класс объекта: значение, такое как «Object», «String», «Number», «Function», «Window», «Document» и т. д. Такое поведение стандартного метода toString() иногда бывает полезно для определения типа или класса неизвестного объекта. Однако большинство объектов имеют собственную версию toString(), поэтому для произвольного объекта o необходимо явно вызывать метод Object.toString() с помощью следующего кода:

```
Object.prototype.toString.apply(o);
```

Обратите внимание, что этот способ идентификации неизвестных объектов годится только для встроенных объектов. Если вы определяете собственный класс объектов, то класс для него будет соответствовать значению «Object». В этом случае дополнительную информацию об объекте позволит получить свойство Object.constructor.

Метод toString() может быть очень полезен при отладке JavaScript-программ – он позволяет выводить объекты и видеть их значения. По одной только этой причине есть смысл определять метод toString() для каждого создаваемого вами класса. Несмотря на то, что метод toString() обычно вызывается системой автоматически, бывают случаи, когда его требуется вызвать явно. Например, чтобы выполнить явное преобразование объекта в строку, если JavaScript не делает это автоматически:

```
y = Math.sqrt(x);    // Вычисляет число
ystr = y.toString(); // Преобразует его в строку
```

В этом примере следует обратить внимание на то, что числа имеют встроенный метод toString(), обеспечивающий принудительное преобразование.

В других обстоятельствах предпочтительным может оказаться метод toString() – даже в таком контексте, когда JavaScript выполняет преобразование автоматически. Явное использование метода toString() может сделать код более понятным:

```
alert(my_obj.toString());
```

См. также Object.constructor(), Object.toLocaleString(), Object.valueOf()

Object.valueOf() ECMAScript v1

элементарное значение указанного объекта

Синтаксис

объект.valueOf()

Возвращаемое значение

Элементарное значение, связанное с *объектом*, если оно есть. Если с *объектом* не связано значение, метод возвращает сам *объект*.

Описание

Метод valueOf() объекта возвращает элементарное значение, связанное с этим объектом, если оно есть. Для объектов типа Object элементарное значение отсутствует, и метод такого объекта возвращает сам объект.

Однако для объектов типа Number метод valueOf() возвращает элементарное числовое значение, представляемое объектом. Аналогично он возвращает элементарное логическое значение, связанное с объектом Boolean, или строку, связанную с объектом String.

Программисту редко приходится явно вызывать метод valueOf(). Интерпретатор JavaScript делает это автоматически всякий раз, встретив объект там, где ожидается элементарное значение. Из-за автоматического вызова метода valueOf() фактически трудно даже провести различие между элементарными значениями и соответствующими им объектами. Оператор typeof, например, показывает различие между строками и объектами String, но с практической точки зрения они работают в JavaScript-коде эквивалентным образом.

Метод valueOf() объектов Number, Boolean и String преобразует эти объекты-обертки в представляемые ими элементарные значения. Конструктор Object() выполняет противоположную операцию при вызове с числовым, логическим или строковым аргументом: он «заворачивает» элементарное значение в соответствующий объект-обертку. В большинстве случаев JavaScript берет это преобразование «элементарное значение – объект» на себя, поэтому необходимость в таком вызове конструктора Object() возникает редко.

Иногда программисту требуется определить специальный метод valueOf() для собственных объектов. Например, определить объектный JavaScript-тип для представления комплексных чисел (вещественное число плюс мнимое число). Как часть этого объектного типа можно определить методы для выполнения комплексного сложения, умножения и т. д. Еще может потребоваться возможность рассматривать комплексные числа как обычные вещественные путем отбрасывания мнимой части. Для этого можно сделать примерно следующее:

```
Complex.prototype.valueOf = new Function("return this.real");
```

Определив метод valueOf() для собственного объектного типа Complex, можно, например, передавать объекты комплексных чисел в функцию Math.sqrt(), которая вычислит квадратный корень из вещественной части комплексного числа.

См. также Object.toString()

Packages LiveConnect

корень иерархии JavaPackage

Синтаксис

Packages

Описание

В реализациях JavaScript, включающих механизм LiveConnect или подобные ему технологии взаимодействия с Java, глобальное свойство Packages – это объект JavaPackage, свойствами которого являются все корневые элементы всех известных иерархий Java-пакетов. Например, Packages.javax.swing ссылается на Java-пакет – *javax.swing*. Глобальное свойство java является сокращением от Packages.java.

См. также java, JavaPackage; глава 12

parseFloat() ECMAScript v1

преобразует строку в число

Синтаксис

parseFloat(*s*)

Аргументы

s Строка, для которой должен быть выполнен синтаксический разбор и которая должна быть преобразована в число.

Возвращаемое значение

Возвращает выделенное из строки число или NaN, если *s* не начинается с допустимого числа. В JavaScript 1.0, если *s* не может быть преобразовано в число, parseFloat() возвращает 0 вместо NaN.

Описание

Функция parseFloat() выполняет синтаксический разбор строки и возвращает первое число, найденное в *s*. Разбор прекращается и значение возвращается, когда parseFloat() встречает в *s* символ, который не является допустимой частью числа. Если *s* не начинается с числа, которое parseFloat() может разобрать, функция возвращает

значение «нечисло» (NaN). Проверка на это возвращаемое значение выполняется функцией isNaN(). Чтобы выделить только целую часть числа, используется функция parseInt(), а не parseFloat().

См. также isNaN(), parseInt()

parseInt() ECMAScript v1

преобразует строку в целое число

Синтаксис

parseInt(*s*)

parseInt(*s*, *основание*)

Аргументы

s Разбираемая строка.

основание

Необязательный строковый аргумент, представляющий основание системы счисления анализируемого числа. Если этот аргумент отсутствует или равен 0, извлекается десятичное или шестнадцатеричное (если число начинается с префикса «0x» или «0X») число. Если этот аргумент меньше 2 или больше 36, parseInt() возвращает NaN.

Возвращаемое значение

Возвращается извлекаемое число (NaN, если *s* не начинается с корректного целого). В JavaScript 1.0, если невозможно выполнить синтаксический разбор строки *s*, parseInt() возвращает 0 вместо NaN.

Описание

Функция parseInt() выполняет синтаксический разбор строки *s* и возвращает первое число (с необязательным начальным знаком «минус»), найденное в *s*. Разбор останавливается и значение возвращается, когда parseInt() встречает в *s* символ, не являющийся допустимой цифрой для указанного основания. Если *s* не начинается с числа, которое может быть проанализировано функцией parseInt(), функция возвращает значение «нечисло» (NaN). Проверка на это возвращаемое значение выполняется функцией isNaN().

Аргумент *основание* задает основание извлекаемого числа. При основании, равном 10, parseInt() извлекает десятичное число. Если этот аргумент равен 8, то извлекается восьмеричное число (состоящее из цифр от 0 до 7), а если 16 – шестнадцатеричное (цифры от 0 до 9 и буквы от A до F). Аргумент *основание* может быть любым числом от 2 до 36.

Если основание равно 0 или не указано, parseInt() пытается определить систему счисления по строке *s*. Если *s* начинается (после необязательного знака «минус») с префикса «0x», parseInt() разбирает оставшуюся часть *s* как шестнадцатеричное число. Если *s* начинается с цифры 0, стандарт ECMAScript v3 разрешает интерпретировать следующие символы либо как восьмеричное, либо как десятичное число. Если *s* начинается с цифры от 1 до 9, parseInt() разбирает строку как десятичное число.

Пример

```
parseInt("19", 10); // Возвращает 19 (10 + 9)
parseInt("11", 2);  // Возвращает 3 (2 + 1)
```

```
parseInt("17", 8);  // Возвращает 15 (8 + 7)
parseInt("1f", 16); // Возвращает 31 (16 + 15)
parseInt("10");     // Возвращает 10
parseInt("0x10");   // Возвращает 16
parseInt("010");    // Неоднозначно: возвращает либо 10, либо 8
```

Ошибки

Когда *основание* не указано, ECMAScript v3 разрешает анализировать строку, начинающуюся с цифры 0 (но не с префикса «0x» или «0X»), либо как восьмеричное, либо как десятичное число. Во избежание двусмысленности необходимо явно указать основание или не указывать его только в тех случаях, когда наверняка известно, что все разбираемые числа являются десятичными или шестнадцатеричными с префиксами «0x» или «0X».

См. также

`isNaN()`, `parseFloat()`

RangeError

ECMAScript v3

генерируется, когда число выходит из допустимого диапазона

Object→Error→RangeError

Конструктор

`new RangeError()`

`new RangeError(`*сообщение*`)`

Аргументы

сообщение

Необязательное сообщение об ошибке, предоставляющее дополнительную информацию об исключении. Если этот аргумент указан, он используется в качестве значения свойства `message` объекта `RangeError`.

Возвращаемое значение

Вновь созданный объект `RangeError`. Если аргумент *сообщение* указан, то для объекта `RangeError` он будет выступать в качестве значения свойства `message`; в противном случае `RangeError` возьмет в качестве значения этого свойства предлагаемую по умолчанию строку, определенную в реализации. Конструктор `RangeError()`, вызываемый как функция (без оператора `new`), ведет себя точно так же, как и при вызове с оператором `new`.

Свойства

`message`

Сообщение об ошибке, предоставляющее дополнительную информацию об исключении. Это свойство содержит строку, переданную конструктору, или предлагаемую по умолчанию строку, определенную в реализации. Дополнительные сведения см. в статье о свойстве `Error.message`.

`name`

Строка, задающая тип исключения. Все объекты `RangeError` наследуют для этого свойства строку `"RangeError"`.

Описание

Экземпляр класса `RangeError` создается, когда числовое значение оказывается вне допустимого диапазона. Например, установка длины массива равной отрицательному

числу приводит к генерации исключения RangeError. Дополнительные сведения о генерации и перехвате исключений см. в статье об объекте Error.

См. также Error, Error.message, Error.name

ReferenceError ECMAScript v3

генерируется при попытке чтения несуществующей переменной **Object→Error→ReferenceError**

Конструктор

new ReferenceError()

new ReferenceError(*сообщение*)

Аргументы

сообщение

Необязательное сообщение об ошибке, предоставляющее дополнительную информацию об исключении. Если этот аргумент указан, он выступает в качестве значения свойства message объекта ReferenceError.

Возвращаемое значение

Вновь созданный объект ReferenceError. Если аргумент *сообщение* указан, объект ReferenceError берет его в качестве значения своего свойства message; в противном случае он берет предлагаемую по умолчанию строку, определенную в реализации. Конструктор ReferenceError(), вызываемый как функция (без оператора new), ведет себя точно так же, как при вызове с оператором new.

Свойства

message

Сообщение об ошибке, предоставляющее дополнительную информацию об исключении. Это свойство содержит строку, переданную конструктору, или предлагаемую по умолчанию строку, определенную в реализации. Дополнительные сведения см. в статье о свойстве Error.message.

name

Строка, задающая тип исключения. Все объекты ReferenceError наследуют для этого свойства строку "ReferenceError".

Описание

Экземпляр класса ReferenceError создается при попытке прочитать значение несуществующей переменной. Дополнительные сведения о генерации и перехвате исключений см. в статье об объекте Error.

См. также Error, Error.message, Error.name

RegExp ECMAScript v3

регулярные выражения для поиска по шаблону **Object→RegExp**

Синтаксис литерала

/маска/атрибуты

Конструктор

new RegExp(*шаблон, атрибуты*)

Аргументы

шаблон

Строка, задающая шаблон регулярного выражения или другое регулярное выражение.

атрибуты

Необязательная строка, содержащая любые из атрибутов g, i и m, задающих глобальный, нечувствительный к регистру и многострочный поиск соответственно. До выхода стандарта ECMAScript атрибут m не был доступен. Если аргумент *шаблон* — это регулярное выражение, а не строка, данный аргумент может отсутствовать.

Возвращаемое значение

Возвращается новый объект RegExp с указанными шаблоном и атрибутами. Если аргумент *шаблон* представляет собой регулярное выражение, а не строку, конструктор RegExp() создает новый объект RegExp, используя те же шаблон и атрибуты, что и в указанном объекте RegExp. Если RegExp() вызывается как функция (без оператора new), то ведет себя точно так же, как с оператором new, кроме случая, когда *шаблон* уже является объектом RegExp; тогда функция возвращает аргумент *шаблон*, а не создает новый объект RegExp.

Исключения

SyntaxError

Генерируется, если *шаблон* не является допустимым регулярным выражением или если аргумент *атрибуты* содержит символы, отличные от g, i и m.

TypeError

Генерируется, если *шаблон* — это объект RegExp, и аргумент *атрибуты* не опущен.

Свойства экземпляра

global

Признак присутствия в RegExp атрибута g.

ignoreCase

Признак присутствия в RegExp атрибута i.

lastIndex

Позиция символа при последнем обнаружении соответствия; используется для поиска в строке нескольких соответствий.

multiline

Признак присутствия в RegExp атрибута m.

source

Исходный текст регулярного выражения.

Методы

exec()

Выполняет мощный универсальный поиск по шаблону.

test()

Проверяет, содержит ли строка данный шаблон.

Описание

Объект RegExp представляет регулярное выражение – мощное средство для поиска в строках по шаблону. Синтаксис и применение регулярных выражений полностью описаны в главе 11.

См. также

Глава 11

RegExp.exec() ECMAScript v3

универсальный поиск по шаблону

Синтаксис

regexp.exec(*строка*)

Аргументы

строка

Строка, в которой выполняется поиск.

Возвращаемое значение

Массив, содержащий результаты поиска или значение null, если соответствия не найдено. Формат возвращаемого массива описан далее.

Исключения

TypeError

Генерируется, если метод вызывается для объекта, не являющегося объектом RegExp.

Описание

Метод exec() – наиболее мощный из всех методов объектов RegExp и String для поиска по шаблону. Это универсальный метод, использовать который несколько сложнее, чем методы RegExp.test(), String.search(), String.replace() и String.match().

Метод exec() ищет в *строке* текст, соответствующий выражению *regexp*. И если находит, то возвращает массив результатов; в противном случае возвращается значение null. Элемент 0 полученного массива представляет собою искомый текст. Элемент 1 – это текст, соответствующий первому подвыражению в скобках внутри *regexp*, если оно есть. Элемент 2 соответствует второму подвыражению и т. д. Свойство length массива как обычно определяет количество элементов в массиве. В дополнение к элементам массива и свойству length значение, возвращаемое exec(), имеет еще два свойства. Свойство index указывает позицию первого символа искомого текста. Свойство input ссылается на *строку*. Этот возвращаемый массив совпадает с массивом, возвращаемым методом String.match(), когда он вызывается для неглобального объекта RegExp.

Когда метод exec() вызывается для неглобального шаблона, он выполняет поиск и возвращает описанный выше результат. Однако если *regexp* – глобальное регулярное выражение, exec() ведет себя несколько сложнее. Он начинает поиск в *строке* с символьной позиции, заданной свойством regexp.lastIndex. Найдя соответствие, метод устанавливает свойство lastIndex равным позиции первого символа после найденного соответствия. Это значит, что exec() можно вызвать несколько раз, чтобы выполнить цикл по всем соответствиям в строке. Если метод exec() больше не находит соответствий, он возвращает значение null и сбрасывает свойство lastIndex в ноль. Начиная поиск непосредственно после успешного нахождения соответствия в другой строке, необходимо соблюдать внимательность и вручную установить свойство lastIndex равным нулю.

Обратите внимание: exec() всегда включает полную информацию для найденного соответствия в возвращаемый им массив независимо от того, является *regexp* глобальным шаблоном или нет. Этим exec() отличается от метода String.match(), который возвращает намного меньше информации при работе с глобальными шаблонами. Вызов exec() в цикле – единственный способ получить полную информацию о результатах поиска для глобального шаблона.

Пример

Для нахождения всех соответствий в строке метод exec() можно запускать в цикле:

```
var pattern = /\bJava\w*\b/g;
var text = "JavaScript is more fun than Java or JavaBeans!";
var result;
while((result = pattern.exec(text)) != null) {
    alert("Matched `" + result[0] +
          "` at position " + result.index +
          " next search begins at position " + pattern.lastIndex);
}
```

См. также

RegExp.lastIndex, RegExp.test(), String.match(), String.replace(), String.search(); глава 11

RegExp.global ECMAScript v3

выполняется ли глобальный поиск по данному регулярному выражению

Синтаксис

regexp.global

Описание

global – это логическое свойство объектов RegExp, доступное только для чтения. Оно указывает, выполняет ли данное регулярное выражение глобальный поиск, т. е. было ли оно создано с атрибутом g.

RegExp.ignoreCase ECMAScript v3

чувствительно ли регулярное выражение к регистру

Синтаксис

regexp.ignoreCase

Описание

ignoreCase – это логическое свойство объектов RegExp, доступное только для чтения. Оно указывает, выполняет ли данное регулярное выражение поиск без учета регистра, т. е. было ли оно создано с атрибутом i.

RegExp.lastIndex ECMAScript v3

начальная позиция следующего поиска

Синтаксис

regexp.lastIndex

Описание

lastIndex — это доступное для чтения и записи свойство объектов RegExp. Для регулярных выражений с установленным атрибутом g оно содержит целое, указывающее позицию в строке символа, который следует непосредственно за последним соответствием, найденным с помощью методов RegExp.exec() и RegExp.test(). Эти методы используют данное свойство в качестве начальной точки при следующем поиске. Благодаря этому данные методы можно вызывать повторно для выполнения цикла по всем соответствиям в строке. Обратите внимание: lastIndex не используется объектами RegExp, не имеющими атрибута g и не представляющими собой глобальные шаблоны.

Это свойство доступно для чтения и для записи, поэтому можно установить его в любой момент, чтобы указать, где в целевой строке должен быть начат следующий поиск. Методы exec() и test() автоматически сбрасывают свойство lastIndex в 0, когда не могут найти какого-либо (или следующего) соответствия. Начиная поиск в новой строке после успешного поиска в предыдущей, необходимо явно установить это свойство равным 0.

См. также RegExp.exec(), RegExp.test()

RegExp.source ECMAScript v3

текст регулярного выражения

Синтаксис

regexp.source

Описание

source — доступное только для чтения строковое свойство объектов RegExp, содержащее текст шаблона RegExp. Текст не включает ограничивающие символы слэша, используемые в литералах регулярных выражений, а также не включает атрибуты g, i и m.

RegExp.test() ECMAScript v3

проверяет, соответствует ли строка шаблону

Синтаксис

regexp.test(*строка*)

Аргументы

строка

Проверяемая строка.

Возвращаемое значение

Возвращает true, если *строка* содержит текст, соответствующий *regexp*, и false — в противном случае.

Исключения

TypeError

Генерируется, если метод вызывается для объекта, не являющегося объектом RegExp.

Описание

Метод test() проверяет *строку*, чтобы увидеть, содержит ли она текст, который соответствует *regexp*. Если да, он возвращает true, в противном случае – false. Вызов метода test() для регулярного выражения r и передача ему строки s эквивалентны следующему выражению:

```
(r.exec(s) != null)
```

Пример

```
var pattern = /java/i;
pattern.test("JavaScript"); // Возвращает true
pattern.test("ECMAScript"); // Возвращает false
```

См. также

RegExp.exec(), RegExp.lastIndex, String.match(), String.replace(), String.substring(); глава 11

RegExp.toString() ECMAScript v3

преобразует регулярное выражение в строку **переопределяет Object.toString()**

Синтаксис

regexp.toString()

Возвращаемое значение

Строковое представление *regexp*.

Исключения

TypeError

Генерируется, если метод вызывается для объекта, который не является объектом RegExp.

Описание

Метод RegExp.toString() возвращает строковое представление регулярного выражения в форме литерала регулярного выражения.

Обратите внимание: от реализаций не требуется обязательного добавления управляющих последовательностей, гарантирующих, что возвращаемая строка будет корректным литералом регулярных выражений. Рассмотрим регулярное выражение, созданное с помощью следующей конструкции:

```
new RegExp("/", "g")
```

Реализация RegExp.toString() может вернуть для регулярного выражения ///g либо добавить управляющую последовательность и вернуть /\//g.

String ECMAScript v1

поддержка строк **Object→String**

Конструктор

new String(*s*) // Функция-конструктор

String(*s*) // Функция преобразования

Аргументы

s Значение, подлежащее сохранению в объекте `String` или преобразованию в элементарное строковое значение.

Возвращаемое значение

Когда функция `String()` вызывается в качестве конструктора (с оператором `new`), она возвращает объект `String`, содержащий строку *s* или строковое представление *s*. Конструктор `String()`, вызванный без оператора `new`, преобразует *s* в элементарное строковое значение и возвращает преобразованное значение.

Свойства

`length`

Количество символов в строке.

Методы

`charAt()`

Извлекает из строки символ, находящийся в указанной позиции.

`charCodeAt()`

Возвращает код символа, находящегося в указанной позиции.

`concat()`

Выполняет конкатенацию одного или нескольких значений со строкой.

`indexOf()`

Выполняет поиск символа или подстроки в строке.

`lastIndexOf()`

Выполняет поиск символа или подстроки в строке с конца.

`localeCompare()`

Сравнивает строки с учетом порядка следования символов национальных алфавитов.

`match()`

Выполняет поиск по маске с помощью регулярного выражения.

`replace()`

Выполняет операцию поиска и замены с помощью регулярного выражения.

`search()`

Ищет в строке подстроку, соответствующую регулярному выражению.

`slice()`

Возвращает фрагмент строки или подстроку в строке.

`split()`

Разбивает строку на массив строк по указанной строке-разделителю или регулярному выражению.

`substr()`

Извлекает подстроку из строки. Аналог метода `substring()`.

`substring()`

Извлекает подстроку из строки.

`toLowerCase()`

Возвращает копию строки, в которой все символы переведены в нижний регистр.

`toString()`

Возвращает элементарное строковое значение.

`toUpperCase()`

Возвращает копию строки, в которой все символы переведены в верхний регистр.

`valueOf()`

Возвращает элементарное строковое значение.

Статические методы

`String.fromCharCode()`

Создает новую строку, помещая в нее принятые в качестве аргументов коды символов.

HTML-методы

С первых дней создания JavaScript в классе `String` определено несколько методов, которые возвращают строку, измененную путем добавления к ней HTML-тегов. Эти методы никогда не были стандартизованы в ECMAScript, но позволяют динамически генерировать HTML-код и в клиентском, и в серверном JavaScript-коде. Если вы готовы к использованию нестандартных методов, можете следующим образом создать исходный HTML-текст для гиперссылки, выделенной полужирным шрифтом красного цвета:

```
var s = "щелкни здесь!";
var html = s.bold().link("javascript:alert('hello')").fontcolor("red");
```

Поскольку эти методы не стандартизованы, для них отсутствуют отдельные справочные статьи:

`anchor(`*имя*`)`

Возвращает копию строки в окружении тега `<a name=>`.

`big()`

Возвращает копию строки в окружении тега `<big>`.

`blink()`

Возвращает копию строки в окружении тега `<blink>`.

`bold()`

Возвращает копию строки в окружении тега `<b>`.

`fixed()`

Возвращает копию строки в окружении тега `<tt>`.

`fontcolor(`*цвет*`)`

Возвращает копию строки в окружении тега `<font color=>`.

`fontsize(`*размер*`)`

Возвращает копию строки в окружении тега `<font size=>`.

`italics()`

Возвращает копию строки в окружении тега `<i>`.

`link(`*url*`)`

Возвращает копию строки в окружении тега `<a href=>`.

`small()`

Возвращает копию строки в окружении тега `<small>`.

`strike()`

Возвращает копию строки в окружении тега `<strike>`.

`sub()`

Возвращает копию строки в окружении тега `<sub>`.

`sup()`

Возвращает копию строки в окружении тега `<sup>`.

Описание

Строки – это элементарный тип данных в JavaScript. Класс `String` предоставляет методы для работы с элементарными строковыми значениями. Свойство `length` объекта `String` указывает количество символов в строке. Класс `String` определяет немало методов для работы со строками. Например, имеются методы для извлечения символа или подстроки из строки или для поиска символа или подстроки. Обратите внимание: строки JavaScript не изменяются – ни один из методов, определенных в классе `String`, не позволяет изменять содержимое строки. Зато методы, подобные `String.toUpperCase()`, возвращают абсолютно новую строку, не изменяя исходную.

В реализациях JavaScript, основанных на базе программного кода, разработанного в компании Netscape (как, например, в Firefox), строки ведут себя как массивы символов, доступные только для чтения. Например, чтобы извлечь третий символ из строки `s`, можно написать `s[2]` вместо более стандартной записи `s.charAt(2)`. Кроме того, инструкция `for/in`, примененная к строке, позволяет перечислить индексы массива для каждого символа в строке. (Однако следует заметить, что свойство `length` не перечисляется, как должно было бы быть по спецификации ECMAScript.) Поскольку поведение строк как массивов является нестандартным, его следует избегать.

См. также

Глава 3

String.charAt() ECMAScript v1

возвращает n-й символ строки

Синтаксис

строка`.charAt(`*n*`)`

Аргументы

n Индекс символа, который должен быть извлечен из *строки*.

Возвращаемое значение

n-й символ *строки*.

Описание

Метод `String.charAt()` возвращает *n*-й символ *строки*. Номер первого символа в строке равен нулю. Если *n* не находится между 0 и `string.length`–1, этот метод возвращает пустую строку. Обратите внимание: в JavaScript нет символьного типа данных, отличного от строкового, поэтому извлеченный символ представляет собой строку длиной 1.

См. также

`String.charCodeAt()`, `String.indexOf()`, `String.lastIndexOf()`

String.charCodeAt() ECMAScript v1

возвращает код n-го символа строки

Синтаксис

строка.charCodeAt(*n*)

Аргументы

n Индекс символа, код которого должен быть получен.

Возвращаемое значение

Код Unicode *n*-го символа в *строке* – 16-разрядное целое между 0 и 65 535.

Описание

Метод charCodeAt() аналогичен charAt(), за исключением того, что возвращает код символа, находящегося в определенной позиции, а не подстроку, содержащую сам символ. Если *n* отрицательно либо меньше или равно длине строки, charCodeAt() возвращает NaN.

Создание строки по коду Unicode символа описано в статье, рассказывающей о методе String.fromCharCode().

См. также String.charAt(), String.fromCharCode()

String.concat() ECMAScript v3

объединяет строки

Синтаксис

строка.concat(*значение, ...*)

Аргументы

значение, ...

Одно или более значений, объединяемых со *строкой*.

Возвращаемое значение

Новая строка, полученная при объединении всех аргументов со *строкой*.

Описание

concat() преобразует все свои аргументы в строки (если это нужно) и добавляет их по порядку в конец строки. Возвращает полученную объединенную строку. Обратите внимание: сама *строка* при этом не меняется.

Метод String.concat() представляет собой аналог Array.concat(). Следует отметить, что конкатенацию строк часто проще выполнить с помощью оператора +.

См. также Array.concat()

String.fromCharCode() ECMAScript v1

создает строку из кодов символов

Синтаксис

String.fromCharCode(*c1, c2, ...*)

Аргументы

c1, c2, ...

Ноль или более целых значений, задающих коды Unicode для символов создаваемой строки.

Возвращаемое значение

Новая строка, содержащая символы с указанными кодами.

Описание

Этот статический метод обеспечивает создание строки из отдельных числовых кодов ее символов, заданных в качестве аргументов. Следует заметить, что статический метод `fromCharCode()` представляет собой свойство конструктора `String()` и фактически не является строковым методом или методом объектов `String`.

Парным для описываемого метода является метод экземпляра `String.charCodeAt()`, который служит для получения кодов отдельных символов строки.

Пример

```
// Создает строку "hello"
var s = String.fromCharCode(104, 101, 108, 108, 111);
```

См. также `String.charCodeAt()`

String.indexOf() ECMAScript v1

поиск подстроки в строке

Синтаксис

строка.indexOf(*подстрока*)
строка.indexOf(*подстрока, start*)

Аргументы

подстрока

Подстрока, которая должна быть найдена в строке.

start

Необязательный целый аргумент, задающий позицию в строке, с которой следует начать поиск. Допустимые значения от 0 (позиция первого символа в строке) до `string.length`–1 (позиция последнего символа в строке). Если этот аргумент отсутствует, поиск начинается с первого символа строки.

Возвращаемое значение

Позиция первого вхождения *подстроки* в строке, начиная с позиции *start*, если подстрока найдена, или –1, если не найдена.

Описание

`String.indexOf()` выполняет поиск в строке от начала к концу, чтобы увидеть, содержит ли она искомую *подстроку*. Поиск начинается с позиции *start* в строке или с начала строки, если аргумент *start* не указан. Если подстрока найдена, `String.indexOf()` возвращает позицию первого символа первого вхождения *подстроки* в строке. Позиции символов в строке нумеруются с нуля.

Если *подстрока* в строке не найдена, `String.indexOf()` возвращает –1.

См. также `String.charAt()`, `String.lastIndexOf()`, `String.substring()`

String.lastIndexOf() ECMAScript v1

поиск подстроки в строке, начиная с конца

Синтаксис

строка`.lastIndexOf(`*подстрока*`)`
строка`.lastIndexOf(`*подстрока, start*`)`

Аргументы

подстрока

Подстрока, которая должна быть найдена в строке.

start

Необязательный целый аргумент, задающий позицию в строке, с которой следует начать поиск. Допустимые значения: от 0 (позиция первого символа в строке) до *строка*`.length`–1 (позиция последнего символа в строке). Если этот аргумент отсутствует, поиск начинается с последнего символа строки.

Возвращаемое значение

Позиция последнего вхождения *подстроки* в строке, начиная с позиции *start*, если подстрока найдена, или –1, если такое вхождение не найдено.

Описание

`String.lastIndexOf()` просматривает строку от конца к началу, чтобы увидеть, содержит ли она *подстроку*. Поиск выполняется с позиции *start* внутри строки или с конца строки, если аргумент *start* не указан. Если подстрока найдена, `String.lastIndexOf()` возвращает позицию первого символа этого вхождения. Метод выполняет поиск от конца к началу, поэтому первое найденное вхождение является последним в строке, расположенным до позиции *start*.

Если *подстрока* не найдена, `String.lastIndexOf()` возвращает –1.

Обратите внимание: хотя метод `String.lastIndexOf()` ищет строку от конца к началу, он все равно нумерует позиции символов в *строке* с начала. Первый символ строки занимает позицию с номером 0, а последний – *строка*`.length`–1.

См. также `String.charAt()`, `String.indexOf()`, `String.substring()`

String.length ECMAScript v1

длина строки

Синтаксис

строка`.length`

Описание

Свойство `String.length` – это доступное только для чтения целое, указывающее количество символов в заданной *строке*. Для любой строки `s` индекс последнего символа равен `s.length`–1. Свойство `length` строки не перечисляется циклом `for/in` и не может быть удалено с помощью оператора `delete`.

String.localeCompare() ECMAScript v3

сравнивает строки с учетом порядка следования символов национальных алфавитов

Синтаксис

строка.localeCompare(*целевая_строка*)

Аргументы

целевая_строка

Строка, сравниваемая со *строкой* с учетом порядка следования символов национальных алфавитов.

Возвращаемое значение

Число, обозначающее результат сравнения. Если строка «меньше» *целевой_строки*, localeCompare() возвращает отрицательное число. Если строка «больше» *целевой_строки*, метод возвращает положительное число. Если строки идентичны или неразличимы в соответствии с региональными соглашениями о сортировке, метод возвращает 0.

Описание

Когда к строкам применяются операторы < и >, сравнение выполняется только по кодам этих символов в Unicode; порядок сортировки, принятый в текущем регионе, не учитывается. Сортировка, выполняемая подобным образом, не всегда оказывается верной. Возьмем, например, испанский язык, в котором буквы «ch» сортируются как одна буква, расположенная между буквами «c» и «d».

Метод localeCompare() служит для сравнения строк с учетом порядка сортировки, по умолчанию определяемого региональными настройками. Стандарт ECMAScript не описывает, как должно выполняться сравнение с учетом региона; в нем просто указано, что эта функция руководствуется порядком сортировки, определенным операционной системой.

Пример

Отсортировать массив строк в порядке, учитывающем региональные параметры, можно следующим образом:

```
var strings; // Сортируемый массив строк; инициализируется в другом месте
strings.sort(function(a,b) { return a.localeCompare(b) });
```

String.match() ECMAScript v3

находит одно или более соответствий регулярному выражению

Синтаксис

строка.match(*regexp*)

Аргументы

regexp

Объект RegExp, задающий шаблон для поиска. Если этот аргумент не является объектом RegExp, он сначала преобразуется с помощью конструктора RegExp().

Возвращаемое значение

Массив, содержащий результаты поиска. Содержимое массива зависит от того, установлен ли в *regexp* глобальный атрибут g. Далее это возвращаемое значение описано подробно.

Описание

Метод match() ищет в строке соответствия маске, определенной выражением *regexp*. Поведение этого метода существенно зависит от того, установлен ли атрибут g в выражении *regexp* или нет. Регулярные выражения подробно рассмотрены в главе 11.

Если в *regexp* нет атрибута g, match() ищет одно соответствие в строке. Если соответствие не найдено, match() возвращает null. В противном случае метод возвращает массив, содержащий информацию о найденном соответствии. Элемент массива с индексом 0 содержит найденный текст. Оставшиеся элементы содержат текст, соответствующий всем подвыражениям внутри регулярного выражения. В дополнение к этим обычным элементам массива возвращаемый массив имеет еще два объектных свойства. Свойство index массива указывает позицию начала найденного текста внутри строки. Кроме того, свойство input возвращаемого массива содержит ссылку на саму строку.

Если в *regexp* имеется флаг g, match() выполняет глобальный поиск, находя в строке все соответствующие подстроки. Метод возвращает null, если соответствие не найдено, или массив, если найдено одно и более соответствий. Однако содержимое полученного массива при глобальном поиске существенно отличается. В этом случае элементы массива содержат все найденные подстроки в *строке*, а сам массив не имеет свойств index и input. Обратите внимание: для глобального поиска match() не предоставляет информации о подвыражениях в скобках и не указывает, в каком месте *строки* было найдено каждое из соответствий. Эту информацию для глобального поиска можно получить методом RegExp.exec().

Пример

Следующий код, реализующий глобальный поиск, находит все числа в строке:

```
"1 plus 2 equals 3".match(/\d+/g) // Возвращает ["1", "2", "3"]
```

Следующий код, реализующий неглобальный поиск, выполняется при помощи более сложного регулярного выражения с несколькими подвыражениями. Он находит URL-адрес, а подвыражения регулярного выражения соответствуют протоколу, хосту и пути в URL:

```
var url = /(\w+):\/\/([\w.]+)\/(\S*)/;
var text = "Visit my home page at http://www.isp.com/~david";
var result = text.match(url);
if (result != null) {
    var fullurl = result[0];  // Содержит "http://www.isp.com/~david"
    var protocol = result[1]; // Содержит "http"
    var host = result[2];     // Содержит "www.isp.com"
    var path = result[3];     // Содержит "~david"
}
```

См. также

RegExp, RegExp.exec(), RegExp.test(), String.replace(), String.search(); глава 11

String.replace() ECMAScript v3

заменяет подстроку (подстроки), соответствующую регулярному выражению

Синтаксис

строка.replace(*regexp*, *замена*)

Аргументы

regexp

Объект RegExp, задающий маску заменяемой подстроки. Если этот аргумент представляет собой строку, то она выступает в качестве текстового шаблона для поиска и не преобразуется в объект RegExp.

замена

Строка, задающая текст замены, или функция, вызываемая для генерации текста замены. Подробности см. в подразделе «Описание».

Возвращаемое значение

Новая строка, в которой первое или все соответствия регулярному выражению *regexp* заменены строкой *замена*.

Описание

Метод replace() выполняет операцию поиска и замены для строки. Он ищет в строке одну или несколько подстрок, соответствующих регулярному выражению *regexp*, и заменяет их значением аргумента *замена*. Если в *regexp* указан глобальный атрибут g, replace() заменяет все найденные подстроки. В противном случае метод заменяет только первую найденную подстроку.

Параметр *замена* может быть либо строкой, либо функцией. Если это строка, то каждое найденное соответствие заменяется указанной строкой. Исключение, однако, составляет символ $, который имеет особый смысл в строке *замена*. Как показано в следующей таблице, он показывает, что для замены используется строка, производная от найденного соответствия.

Символы	Замена
$1, $2, ..., $99	Текст, соответствующий подвыражению с номером от 1 до 99 внутри регулярного выражения *regexp*
$&	Подстрока, соответствующая *regexp*
$`	Текст слева от найденной подстроки
$'	Текст справа от найденной подстроки
$$	Символ доллара

В стандарте ECMAScript v3 определено, что аргумент *замена* может быть функцией, а не строкой. В этом случае функция вызывается для каждого найденного соответствия, а возвращаемая ею строка выступает в качестве текста для замены. Первый аргумент, передаваемый функции, представляет собой строку, соответствующую маске. Следующие за ним аргументы – это строки, соответствующие любым подвыражениям внутри шаблона. Таких аргументов может быть ноль или более. Следующий аргумент – это целое, указывающее позицию внутри строки, в которой было найдено соответствие, а последний аргумент функции *замена* – это сама строка.

Пример

Обеспечение правильного регистра букв в слове «JavaScript»:

```
text.replace(/javascript/i, "JavaScript");
```

Преобразование имени из формата «Doe, John» в формат «John Doe»:

```
name.replace(/(\w+)\s*,\s*(\w+)/, "$2 $1");
```

Замена всех двойных кавычек двумя одинарными закрывающими и двумя одинарными открывающими кавычками:

```
text.replace(/"([^"]*)"/g, "``$1''");
```

Перевод первых букв всех слов в строке в верхний регистр:

```
text.replace(/\b\w+\b/g, function(word) {
                             return word.substring(0,1).toUpperCase() +
                                    word.substring(1);
                         });
```

См. также

RegExp, RegExp.exec(), RegExp.test(), String.match(), String.search(); глава 11

String.search() ECMAScript v3

поиск соответствия регулярному выражению

Синтаксис

строка.search(*regexp*)

Аргументы

regexp

Объект RegExp, задающий шаблон, который будет использоваться для поиска в *строке*. Если этот аргумент не является объектом RegExp, он сначала преобразуется путем передачи его конструктору RegExp().

Возвращаемое значение

Позиция начала первой подстроки в строке, соответствующая выражению *regexp*, или –1, если соответствие не найдено.

Описание

Метод search() ищет подстроку в *строке*, соответствующую регулярному выражению *regexp*, и возвращает позицию первого символа найденной подстроки или –1, если соответствие не найдено.

Метод не выполняет глобального поиска, игнорируя флаг g. Он также игнорирует свойство regexp.lastIndex и всегда выполняет поиск с начала строки, следовательно, всегда возвращает позицию первого соответствия, найденного в *строке*.

Пример

```
var s = "JavaScript is fun";
s.search(/script/i) // Возвращает 4
s.search(/a(.)a/)   // Возвращает 1
```

См. также

RegExp, RegExp.exec(), RegExp.test(), String.match(), String.replace(); глава 11

String.slice() ECMAScript v3

извлечение подстроки

Синтаксис

строка.slice(*start, end*)

Аргументы

start

Индекс в *строке*, с которой должен начинаться фрагмент. Если этот аргумент отрицателен, он обозначает позицию, измеряемую от конца строки. То есть –1 обозначает последний символ, –2 – второй с конца и т. д.

end

Индекс символа исходной *строки* непосредственно после конца извлекаемого фрагмента. Если он не указан, фрагмент включает все символы от позиции *start* до конца строки. Если этот аргумент отрицателен, он обозначает позицию, отсчитываемую от конца *строки*.

Возвращаемое значение

Новая строка, которая содержит все символы *строки*, начиная с символа в позиции *start* (и включая его) и заканчивая символом в позиции *end* (но не включая его).

Описание

Метод `slice()` возвращает строку, содержащую фрагмент, или подстроку *строки*, но не изменяет *строку*.

Методы `slice()`, `substring()` и признанный устаревшим метод `substr()` объекта `String` возвращают части строки. Метод `slice()` более гибок, чем `substring()`, поскольку допускает отрицательные значения аргументов. Метод `slice()` отличается от `substr()` тем, что задает строку с помощью двух символьных позиций, а `substr()` использует одно значение позиции и длину. Кроме того, `String.slice()` является аналогом `Array.slice()`.

Пример

```
var s = "abcdefg";
s.slice(0,4)   // Возвращает "abcd"
s.slice(2,4)   // Возвращает "cd"
s.slice(4)     // Возвращает "efg"
s.slice(3,-1)  // Возвращает "def"
s.slice(3,-2)  // Возвращает "de"
s.slice(-3,-1) // Должен возвращать "ef"; в IE 4 возвращает "abcdef"
```

Ошибки

Отрицательные значения *start* не работают в Internet Explorer 4 (в более поздних версиях Internet Explorer эта ошибка исправлена). Они обозначают не символьную позицию, отсчитываемую от конца строки, а позицию 0.

См. также

`Array.slice()`, `String.substring()`

String.split() ECMAScript v3

разбивает строку на массив строк

Синтаксис

строка.split(*разделитель, лимит*)

Аргументы

разделитель

Строка или регулярное выражение, по которому разбивается *строка*.

лимит

Необязательное целое, задающее максимальную длину полученного массива. Если этот аргумент указан, то количество возвращенных подстрок не будет превышать указанное. Если он не указан, разбивается вся строка независимо от ее длины.

Возвращаемое значение

Массив строк, создаваемый путем разбиения строки на подстроки по границам, заданным *разделителем*. Подстроки в возвращаемом массиве не включают сам *разделитель*, кроме описываемого далее случая.

Описание

Метод `split()` создает и возвращает массив подстрок указанной *строки*, причем размер возвращаемого массива не превышает указанный *лимит*. Эти подстроки создаются путем поиска текста, соответствующего *разделителю*, в *строке* от начала до конца, и разбиения строки до и после найденного текста. Ограничивающий текст не включается ни в одну из возвращаемых строк, кроме случая, описываемого далее. Следует отметить, что если *разделитель* соответствует началу *строки*, первый элемент возвращаемого массива будет пустой строкой – текстом, присутствующим перед разделителем. Аналогично, если разделитель соответствует концу строки, последний элемент массива (если это не противоречит значению аргумента *лимит*) будет пустой строкой.

Если *разделитель* не указан, строка вообще не разбивается, и возвращаемый массив содержит только один строковый элемент, представляющий собой строку целиком. Если *разделитель* представляет собой пустую строку или регулярное выражение, соответствующее пустой строке, то строка разбивается между каждым символом, а возвращаемый массив имеет ту же длину, что и исходная строка, если не указан меньший *лимит*. (Это особый случай, поскольку пустая строка перед первым и за последним символами отсутствует.)

Ранее отмечалось, что подстроки в массиве, возвращаемом описываемым методом, не содержат текста разделителя, использованного для разбиения строки. Однако если *разделитель* – это регулярное выражение, содержащее подвыражения в скобках, то подстроки, соответствующие этим подвыражениям (но не текст, соответствующий регулярному выражению в целом), включаются в возвращаемый массив.

Метод `String.split()` – обратный методу `Array.join()`.

Пример

Метод `split()` наиболее полезен при работе с сильно структурированными строками:

```
"1:2:3:4:5".split(":"); // Возвращает ["1","2","3","4","5"]
"|a|b|c|".split("|");   // Возвращает ["", "a", "b", "c", ""]
```

Еще одно распространенное применение метода `split()` состоит в разбиении команд и других подобных строк на слова, разделенные пробелами:

```
var words = sentence.split(' ');
```

Проще разбить строку на слова, используя в качестве разделителя регулярное выражение:

```
var words = sentence.split(/\s+/);
```

Чтобы разбить строку на массив символов, возьмите в качестве разделителя пустую строку. Если требуется разбить на массив символов только часть строки, задайте аргумент *лимит*:

```
"hello".split("");     // Возвращает ["h","e","l","l","o"]
"hello".split("", 3); // Возвращает ["h","e","l"]
```

Если необходимо, чтобы разделители или одна и более частей разделителя были включены в результирующий массив, напишите регулярное выражение с подвыражениями в скобках. Так, следующий код разбивает строку по HTML-тегам и включает эти теги в результирующий массив:

```
var text = "hello <b>world</b>";
text.split(/(<[^>]*>)/); // Возвращает ["hello ","<b>","world","</b>",""]
```

См. также `Array.join()`, `RegExp`; глава 11

String.substr() JavaScript 1.2; устарел

извлекает подстроку

Синтаксис

строка`.substr(`*начало, длина*`)`

Аргументы

начало

Начальная позиция подстроки. Если аргумент отрицателен, он обозначает позицию, измеряемую от конца *строки*: −1 обозначает последний символ, −2 – второй символ с конца и т. д.

длина

Количество символов в подстроке. Если этот аргумент отсутствует, возвращаемая строка включает все символы от начальной позиции до конца *строки*.

Возвращаемое значение

Копия фрагмента *строки*, начиная с символа, находящегося в позиции *начало* (включительно); имеет длину, равную аргументу *длина*, или заканчивается концом строки, если *длина* не указана.

Описание

Метод `substr()` извлекает и возвращает подстроку строки, но не изменяет строку. Обратите внимание: метод `substr()` задает нужную подстроку с помощью позиции символа и длины. Благодаря этому появляется удобная альтернатива методам `String.substring()` и `String.splice()`, в которых подстрока задается двумя символьными позициями. При этом следует отметить, что метод не стандартизован в ECMAScript и, следовательно, считается устаревшим.

Пример

```
var s = "abcdefg";
s.substr(2,2);  // Возвращает "cd"
s.substr(3);    // Возвращает "defg"
s.substr(-3,2); // Должен возвращать "ef"; в IE 4 возвращает "ab"
```

Ошибки

Отрицательные значения аргумента *начало* не работают в IE 4 (в более поздних версиях IE эта ошибка исправлена). Они задают не позицию символа, отсчитываемую от конца строки, а позицию 0.

См. также `String.slice()`, `String.substring()`

String.substring() ECMAScript v1

возвращает подстроку строки

Синтаксис

строка.substring(*от, до*)

Аргументы

от

Целое, задающее позицию первого символа требуемой подстроки внутри *строки*.

до

Необязательное целое, на единицу большее позиции последнего символа требуемой подстроки. Если этот аргумент опущен, возвращаемая подстрока заканчивается в конце *строки*.

Возвращаемое значение

Новая строка длиной *до* – *от*, содержащая подстроку *строки*. Новая строка содержит символы, скопированные из *строки*, начиная с позиции *от* и заканчивая позицией *до* –1.

Описание

Метод String.substring() возвращает подстроку *строки*, содержащую символы между позициями *от* и *до*. Символ в позиции *от* включается в подстроку, а символ в позиции *до* не включается.

Если *от* равно *до*, метод возвращает пустую строку (длиной 0). Если *от* больше *до*, метод сначала меняет два аргумента местами, а затем возвращает строку между ними.

Помните, что символ в позиции *от* включается в подстроку, а символ в позиции *до* в нее не включается. Может показаться, что это поведение взято «с потолка» и нелогично, но особенность такой системы состоит в том, что длина возвращаемой подстроки всегда равна *до* – *от*.

Обратите внимание: для извлечения подстрок из строки также могут использоваться метод String.slice() и нестандартный метод String.substr(). В отличие от этих методов, String.substring() не может принимать отрицательные значения аргументов.

См. также

String.charAt(), String.indexOf(), String.lastIndexOf(), String.slice(), String.substr()

String.toLocaleLowerCase() ECMAScript v3

преобразует символы строки в нижний регистр

Синтаксис

строка.toLocaleLowerCase()

Возвращаемое значение

Копия *строки*, преобразованная в нижний регистр с учетом региональных параметров. Только немногие языки, такие как турецкий, имеют специфические для региона соответствия регистров, поэтому данный метод обычно возвращает то же значение, что и метод toLowerCase().

См. также `String.toLocaleUpperCase()`, `String.toLowerCase()`, `String.toUpperCase()`

String.toLocaleUpperCase() ECMAScript v3

преобразует символы строки в верхний регистр

Синтаксис

строка`.toLocaleUpperCase()`

Возвращаемое значение

Копия *строки*, преобразованная в верхний регистр с учетом региональных параметров. Лишь немногие языки, такие как турецкий, имеют специфические для региона соответствия регистров, поэтому данный метод обычно возвращает то же значение, что и метод `toUpperCase()`.

См. также `String.toLocaleLowerCase()`, `String.toLowerCase()`, `String.toUpperCase()`

String.toLowerCase() ECMAScript v1

преобразует символы строки в нижний регистр

Синтаксис

строка`.toLowerCase()`

Возвращаемое значение

Копия *строки*, в которой все символы верхнего регистра преобразованы в эквивалентные им символы нижнего регистра, если такие имеются.

String.toString() ECMAScript v1

возвращает строку **переопределяет Object.toString()**

Синтаксис

строка`.toString()`

Возвращаемое значение

Элементарное строковое значение *строки*. Вызов этого метода требуется редко.

Исключения

`TypeError`

Генерируется, если метод вызывается для объекта, не являющегося объектом `String`.

См. также `String.valueOf()`

String.toUpperCase() ECMAScript v1

преобразует символы строки в верхний регистр

Синтаксис

строка`.toUpperCase()`

Возвращаемое значение

Копия *строки*, в которой все символы нижнего регистра преобразованы в эквивалентные им символы верхнего регистра, если такие имеются.

String.valueOf() ECMAScript v1

возвращает строку **переопределяет Object.valueOf()**

Синтаксис

строка.valueOf()

Возвращаемое значение

Элементарное строковое значение *строки*.

Исключения

TypeError

Генерируется, если метод вызывается для объекта, не являющегося объектом String.

См. также String.toString()

SyntaxError ECMAScript v3

свидетельствует о синтаксической ошибке **Object→Error→SyntaxError**

Конструктор

new SyntaxError()

new SyntaxError(*сообщение*)

Аргументы

сообщение

Необязательное сообщение об ошибке, предоставляющее дополнительную информацию об исключении. Если этот аргумент указан, он выступает в качестве значения свойства message объекта SyntaxError.

Возвращаемое значение

Вновь созданный объект SyntaxError. Если аргумент *сообщение* указан, объект SyntaxError берет его в качестве значения своего свойства message; в противном случае в качестве значения этого свойства он берет предлагаемую по умолчанию строку, определенную в реализации. Конструктор SyntaxError(), вызванный как функция (без оператора new), ведет себя точно так же, как если бы он был вызван с оператором new.

Свойства

message

Сообщение об ошибке, предоставляющее дополнительную информацию об исключении. Это свойство содержит строку, переданную конструктору, или предлагаемую по умолчанию строку, определенную в реализации. Подробнее см. в статье о свойстве Error.message.

name

Строка, задающая тип исключения. Все объекты SyntaxError наследуют для этого свойства строку "SyntaxError".

Описание

Экземпляр класса SyntaxError сигнализирует о синтаксической ошибке в JavaScript-коде. Метод eval(), а также конструкторы Function() и RegExp() могут генерировать исключения этого типа. Подробности о генерации и перехвате исключений см. в статье об объекте Error.

См. также Error, Error.message, Error.name

TypeError ECMAScript v3

генерируется, когда значение имеет неверный тип **Object→Error→TypeError**

Конструктор

new TypeError()
new TypeError(*сообщение*)

Аргументы

сообщение

Необязательное сообщение об ошибке, предоставляющее дополнительную информацию об исключении. Если этот аргумент указан, он выступает в качестве значения свойства message объекта TypeError.

Возвращаемое значение

Вновь созданный объект TypeError. Если аргумент *сообщение* указан, объект TypeError берет его в качестве значения своего свойства message; в противном случае в качестве значения этого свойства он берет предлагаемую по умолчанию строку, определенную в реализации. Конструктор TypeError(), вызванный как функция (без оператора new), ведет себя точно так же, как если бы он был вызван с оператором new.

Свойства

message

Сообщение об ошибке, содержащее дополнительную информацию об исключении. Это свойство содержит строку, переданную конструктору, или предлагаемую по умолчанию строку, определенную в реализации. Подробности см. в статье о свойстве Error.message.

name

Строка, задающая тип исключения. Все объекты TypeError наследуют для этого свойства строку "TypeError".

Описание

Экземпляр класса TypeError создается, когда значение имеет не тот тип, который ожидается. Такое чаще всего происходит при попытке обратиться к свойству null или к неопределенному значению объекта. Это исключение может также возникнуть, если вызван метод, определенный одним классом, для объекта, являющегося экземпляром какого-либо другого класса, или если оператору new передается значение, не являющееся функцией-конструктором. Реализациям JavaScript также разрешено создавать объекты TypeError, когда встроенная функция или метод вызывается с большим числом аргументов, чем ожидается. Генерация и перехват исключений подробно рассмотрены в статье об объекте Error.

См. также Error, Error.message, Error.name

undefined

ECMAScript v3

неопределенное значение

Синтаксис

```
undefined
```

Описание

`undefined` – это глобальное свойство, хранящее JavaScript-значение `undefined`. Это то же самое значение, которое возвращается, когда вы пытаетесь прочитать значение несуществующего свойства объекта. Свойство `undefined` не перечисляется циклами `for/in` и не может быть удалено оператором `delete`. Однако `undefined` не является константой и может быть установлено равным любому другому значению, но лучше этого не делать.

Чтобы проверить, является ли значение неопределенным (`undefined`), следует использовать оператор `===`, поскольку оператор `==` считает значение `undefined` равным значению `null`.

unescape()

ECMAScript v1; устарело в ECMAScript v3

декодирует строку с управляющими последовательностями

Синтаксис

```
unescape(s)
```

Аргументы

s Декодируемая строка.

Возвращаемое значение

Декодированная копия *s*.

Описание

`unescape()` – это глобальная функция, декодирующая строку, закодированную с помощью функции `escape()`. Декодирование строки *s* происходит путем поиска и замены последовательности символов в формате `%xx` и `%uxxxx` (где `x` – шестнадцатеричная цифра) Unicode-символами `\u00xx` и `\uxxxx`.

Несмотря на то, что функция `unescape()` была стандартизована в первой версии ECMAScript, она признана устаревшей и исключена из стандарта в спецификации ECMAScript v3. Реализации ECMAScript могут поддерживать эту функцию, но это необязательное требование. Вместо нее следует использовать `decodeURI()` и `decodeURIComponent()`. Подробности и пример см. в статье о функции `escape()`.

См. также `decodeURI()`, `decodeURIComponent()`, `escape()`, `String`

URIError

ECMAScript v3

генерируется методами кодирования и декодирования URI **Object→Error→URIError**

Конструктор

```
new URIError()
new URIError(сообщение)
```

Аргументы

сообщение

Необязательное сообщение об ошибке, предоставляющее дополнительную информацию об исключении. Если этот аргумент указан, он выступает в качестве значения свойства `message` объекта `URIError`.

Возвращаемое значение

Вновь созданный объект `URIError`. Если аргумент *сообщение* указан, объект `URIError` берет его в качестве значения своего свойства `message`; в противном случае в качестве значения этого свойства он берет предлагаемую по умолчанию строку, определенную в реализации. Конструктор `URIError()`, вызванный как функция (без оператора `new`), ведет себя точно так же, как если бы он был вызван с оператором `new`.

Свойства

`message`

Сообщение об ошибке, предоставляющее дополнительную информацию об исключении. Это свойство содержит строку, переданную конструктору, или предлагаемую по умолчанию строку, определенную в реализации. Подробнее см. в статье о свойстве `Error.message`.

`name`

Строка, задающая тип исключения. Все объекты `URIError` наследуют для этого свой1ства строку `"URIError"`.

Описание

Экземпляр класса `URIError` создается функциями `decodeURI()` и `decodeURIComponent()`, если указанная строка содержит недопустимые шестнадцатеричные управляющие последовательности. Это исключение может генерироваться методами `encodeURI()` и `encodeURIComponent()`, если указанная строка содержит недопустимые пары Unicode-символов. Генерация и перехват исключений подробно рассмотрены в статье об объекте `Error`.

См. также `Error`, `Error.message`, `Error.name`

IV

Справочник по клиентскому JavaScript

Эта часть книги представляет собой полный справочник по всем объектам, свойствам, функциям, методам и обработчикам событий клиентского JavaScript. Описание порядка пользования справочником и пример справочной статьи вы найдете в начале части III книги.

В этой части описываются следующие классы и объекты:

Anchor	DOMException	JSObject	Table
Applet	DOMImplementation	KeyEvent	TableCell
Attr	DOMParser	Link	TableRow
Canvas	Element	Location	TableSection
CanvasGradient	Event	MimeType	Text
CanvasPattern	ExternalInterface	MouseEvent	Textarea
CanvasRenderingContext2D	FlashPlayer	Navigator	UIEvent
CDATASection	Form	Node	Window
CharacterData	Frame	NodeList	XMLHttpRequest
Comment	History	Option	XMLSerializer
CSS2Properties	HTMLCollection	Plugin	XPathExpression
CSSRule	HTMLDocument	ProcessingInstruction	XPathResult
CSSStyleSheet	HTMLElement	Range	XSLTProcessor
Document	IFrame	RangeException	
DocumentFragment	Image	Screen	
DocumentType	Input	Select	

Справочник по клиентскому JavaScript

Данная часть книги представляет собой полный справочник по всем объектам, свойствам, функциям, методам и обработчикам событий клиентского JavaScript. Описание порядка пользования справочником и пример справочной статьи вы найдете в начале части III книги. Не поленитесь и внимательно прочитайте этот материал – вам будет легче искать и использовать нужную информацию!

Материал справочника организован в алфавитном порядке. Справочные статьи методов и свойств классов отсортированы по их полным именам, включающим имена классов, в которых они определены. Например, для того чтобы найти метод `submit()` класса `Form`, надо искать статью под заголовком «Form.submit», а не «submit».

Большинство упоминаемых здесь свойств не имеют отдельных справочных статей, которые есть у всех методов и обработчиков событий, тем не менее каждое простое свойство полностью описано в справочной статье определяющего их класса. Так, свойство `images[]` класса `HTMLDocument` рассмотрено в справочной статье об этом классе. Нетривиальные свойства, требующие обстоятельного пояснения, имеют собственные справочные статьи, ссылки на которые можно найти в справочной статье о классе или интерфейсе, определяющем эти свойства. Например, разыскивая свойство `cookie`, в справочной статье о классе `HTMLDocument` вы найдете краткое описание свойства и ссылку на статью под заголовком «HTMLDocument.cookie».

В клиентском JavaScript имеется несколько глобальных свойств и функций, таких как `window`, `history` и `alert()`, а также глобальный объект `Window`, причем «глобальные» свойства и функции фактически представляют собой свойства класса `Window`. Поэтому в данном справочнике глобальные свойства и функции описаны в справочных статьях, начинающихся со слова «Window», например описание функции `alert()` можно найти под заголовком «Window.alert()».

Найдя справочную статью, вы без труда отыщете на ней нужную информацию. Но вам будет легче работать с этим справочником, если вы поймете, как написаны и организованы справочные статьи. Часть III книги начинается с примера справочной статьи, где вы найдете описание структуры каждой справочной статьи и объяснение, как отыскать требуемую информацию в пределах одной статьи.

Anchor

DOM Level 0

цель гиперссылки

Node→Element→HTMLElement→Anchor

Свойства

`String name`

Содержит имя объекта `Anchor`. Значение этого свойства изначально устанавливается равным атрибуту `name` тега `<a>`.

Методы

`focus()`

Прокручивает документ до позиции, в которой якорный элемент становится видимым.

Синтаксис HTML

Объект `Anchor` создается любым стандартным HTML-тегом `<a>`, в котором установлен атрибут `name`:

```
<a name="name"> // Ссылки могут обращаться к этому якорю по его имени
    ...
</a>
```

Описание

Якорный элемент (якорь) – это именованная область внутри HTML-документа. Якоря создаются с помощью тега `<a>`, в котором указан атрибут `name`. Объект `Document` имеет свойство-массив `anchors[]`, содержащий объекты `Anchor`, которые представляют все якорные элементы в документе.

Чтобы заставить броузер отобразить часть документа, в которой находится якорный элемент, достаточно записать в свойство `hash` объекта `Location` символ `#`, указав следом за ним имя якорного элемента, или просто вызвать метод `focus()` самого объекта `Anchor`.

Обратите внимание: тег `<a>` служит также для создания гиперссылок. Несмотря на то что в терминологии HTML гиперссылки часто называются якорями, в JavaScript они представляются объектами `Link`, а не объектами `Anchor`.

Пример

```
// Прокрутить документ до местоположения якорного элемента с именем "_bottom_"
document.anchors['_bottom_'].focus();
```

См. также

`Document`, `Link`, `Location`

Anchor.focus() DOM Level 0

прокручивает документ, пока местоположение якорного элемента не станет видимым

Синтаксис

```
void focus()
```

Описание

Этот метод выполняет прокрутку документа так, чтобы местоположение якорного элемента оказалось в области видимости.

Applet DOM Level 0

апплет, встроенный в веб-страницу

Синтаксис

document`.applets[i]`

document.имя_апплета

Свойства

Имена свойств объекта Applet совпадают с названиями атрибутов HTML-тега <applet> (подробности см. в статье о HTMLDocument). Кроме того, объект обладает всеми общедоступными полями представляемого им Java-апплета.

Методы

Методы объекта Applet совпадают с общедоступными методами представляемого им Java-апплета.

Описание

Объект Applet представляет Java-апплет, встроенный в HTML-документ. Доступ к объектам Applet можно получить с помощью свойства applets[] объекта Document.

Свойства объекта Applet представляют общедоступные поля апплета, а методы объекта Applet – общедоступные методы апплета. Не забывайте, что Java – это строго типизированный язык программирования. Это означает, что все поля апплета объявляются с определенным типом данных и присваивание им значения какого-либо другого типа приводит к ошибке времени выполнения. То же самое относится и к методам апплета: каждый аргумент имеет определенный тип и аргументы не могут быть опущены, как это можно делать в JavaScript. За дополнительной информацией обращайтесь к главе 23.

См. также JSObject; JavaObject в третьей части книги; главы 12 и 23

Attr

DOM Level 1 Core

атрибут элемента документа **Node→Attr**

Свойства

`readonly String name`

Имя атрибута.

`readonly Element ownerElement` *[DOM Level 2]*

Объект Element, содержащий данный атрибут, или значение null, если объект Attr в настоящий момент не связан ни с каким объектом Element.

`readonly boolean specified`

Значение true, если атрибут явно определен в исходном тесте документа или был установлен сценарием; значение false, если атрибут не был определен явно, но его значение по умолчанию определяется в DTD документа.

`String value`

Значение атрибута. При чтении этого свойства значение атрибута возвращается в виде строки. Когда в это свойство записывается строка, она автоматически преобразуется в узел Text, который содержит текст из строки, а сам текстовый узел делается единственным потомком объекта Attr.

Описание

Объект Attr представляет атрибут узла Element. Объекты Attr связаны с узлами Element, но не являются непосредственной частью дерева документа (значение свойства parentNode у них имеет значение null). Получить объект Attr можно с помощью свойства attributes интерфейса Node или вызвав метод getAttributeNode() или getAttributeNodeNS() интерфейса Element.

Значение атрибута представлено узлами-потомками узла Attr. В HTML-документах узел Attr всегда имеет единственный дочерний узел Text, а свойство value предоставляет возможность упрощенного доступа к этому дочернему текстовому узлу для чтения и записи.

Грамматика XML позволяет XML-документам обладать атрибутами, состоящими из узлов Text и EntityReference, по этой причине полное значение атрибута не может быть представлено единственной строкой. Однако на практике веб-броузеры разворачивают все ссылки на сущности в значения XML-атрибутов и не реализуют интерфейс EntityReference (который в этой книге не описывается). Таким образом, в клиентском JavaScript свойство value – это все, что необходимо для чтения и записи значения атрибута.

Поскольку значения атрибутов могут быть представлены в виде строк, интерфейс Attr обычно вообще не используется. В большинстве случаев проще всего работать с атрибутами с помощью методов Element.getAttribute() и Element.setAttribute(). Эти методы применяют строки в качестве значений атрибутов, а узлы Attr вообще не задействуют.

См. также Element

Button

См. статью об объекте Input

Canvas

Firefox 1.5, Safari 1.3, Opera 9

HTML-элемент для создания графических изображений

Node→Element→HTMLElement→Canvas

Свойства

String height

Высота «холста»[1]. Как и в случае с изображениями, может принимать целое значение в пикселах или в процентах от высоты окна. При изменении этого свойства изображение, которое было нарисовано в этом элементе ранее, стирается. Значение по умолчанию – 300.

String width

Ширина «холста». Как и в случае с изображениями, может принимать целое значение в пикселах или в процентах от ширины окна. При изменении этого свойства изображение, которое было нарисовано в этом элементе ранее, стирается. Значение по умолчанию – 300.

Методы

getContext()

Возвращает объект CanvasRenderingContext2D, с помощью которого выполняются операции рисования на холсте. Этому методу следует передать строку "2d", показывая, что будет создаваться двухмерное графическое изображение.

[1] Canvas – холст, область рисования.

Описание

Объект Canvas представляет HTML-элемент <canvas>. Он не обладает собственным поведением, а определяет API для поддержки операций рисования. С помощью этого объекта можно задать ширину и высоту холста, но основная функциональность обеспечивается объектом CanvasRenderingContext2D. Получить этот объект можно методом getContext() объекта Canvas, передав ему строку "2d" в качестве единственного аргумента.

Тег <canvas> впервые появился в броузере Safari 1.3 и к моменту написания этих строк поддерживался броузарами Firefox 1.5 и Opera 9. Тег <canvas> и его API в IE можно смоделировать при помощи свободно распространяемого пакета ExplorerCanvas (*http://excanvas.sourceforge.net/*).

См. также CanvasRenderingContext2D; глава 22

Canvas.getContext()

возвращает контекст рисования

Синтаксис

```
CanvasRenderingContext2D getContext(String contextID)
```

Аргументы

contextID

Данный аргумент определяет тип изображения, которое будет создаваться на данном холсте. В настоящее время допустимым является единственное значение – строка "2d", которая указывает, что будет создаваться двухмерное изображение, в результате чего метод возвращает объект-контекст, экспортирующий API для создания двухмерных изображений.

Возвращаемое значение

Объект CanvasRenderingContext2D, с помощью которого выполняются операции рисования внутри элемента Canvas.

Описание

Возвращает объект-контекст, представляющий тип контекста, используемого для создания изображения. Суть состоит в том, чтобы обеспечить различные контексты для различных типов изображений (двух- или трехмерных). В настоящее время поддерживается только значение аргумента "2d", при использовании которого возвращается объект CanvasRenderingContext2D, реализующий большую часть методов для работы с элементом Canvas.

См. также CanvasRenderingContext2D

CanvasGradient

Firefox 1.5, Safari 1.3, Opera 9

цветной градиент для использования в элементе Canvas **Object→CanvasGradient**

Методы

addColorStop()

Определяет цвет и позицию градиента.

Описание

Объект CanvasGradient представляет цветовой градиент, который может быть присвоен свойствам strokeStyle и fillStyle объекта CanvasRenderingContext2D. Объект CanvasGradient возвращается методами createLinearGradient() и createRadialGradient() объекта CanvasRenderingContext2D.

После создания объекта CanvasGradient следует вызвать метод addColorStop(), с помощью которого определить, какой цвет в какой позиции должен отображаться внутри градиента. Между заданными позициями цвет будет интерполироваться так, чтобы создать эффект плавного перехода или исчезновения цвета. Позиции в начале и конце градиента с прозрачным черным цветом создаются автоматически.

См. также

CanvasRenderingContext2D.createLinearGradient(),CanvasRenderingContext2D.createRadialGradient()

CanvasGradient.addColorStop()

добавляет цветовой переход в некоторой точке градиента

Синтаксис

```
void addColorStop(float offset, String color)
```

Аргументы

offset

Значение с плавающей точкой в диапазоне от 0,0 до 1,0, которое представляет позицию между началом и концом градиента. Значение 0 соответствует начальной позиции, значение 1 – конечной.

color

Цвет в формате цветовой CSS-строки, отображаемый в заданной позиции. Промежуточные цвета между определенными позициями в градиенте интерполируются на основе значений цветов в этой и других позициях.

Описание

Метод addColorStop() обеспечивает механизм описания цветовых переходов в градиенте. Этот метод может вызываться один или более раз, чтобы изменить цвет в конкретных точках, расположенных между началом и концом градиента.

Если этот метод не будет вызван ни разу, градиент останется прозрачным. Чтобы воспроизвести видимый градиент, необходимо определить хотя бы одну позицию.

CanvasPattern

Firefox 1.5, Safari 1.3, Opera 9

шаблон заполнения холста на основе готового изображения **Object→CanvasPattern**

Описание

Объект CanvasPattern возвращается методом createPattern() объекта CanvasRenderingContext2D. Объект CanvasPattern может использоваться в качестве значения свойств strokeStyle и fillStyle объекта CanvasRenderingContext2D.

Объект CanvasPattern не имеет собственных свойств и методов. Порядок создания объекта вы найдете в статье о методе CanvasRenderingContext2D.createPattern().

См. также CanvasRenderingContext2D.createPattern()

CanvasRenderingContext2D

Firefox 1.5, Safari 1.3, Opera 9

объект, используемый для создания изображений **Object→CanvasRenderingContext2D**

Свойства

readonly Canvas canvas

Элемент Canvas, который будет использоваться для создания изображения.

Object fillStyle

Текущий цвет, шаблон или градиент, используемый для заполнения. Это свойство может принимать строковое значение либо объект CanvasGradient или CanvasPattern. Когда в это свойство записывается строка, она воспринимается как значение цвета в формате CSS, который используется для заливки. Когда в свойство записывается объект CanvasGradient или CanvasPattern, заполнение выполняется с помощью указанного градиента или шаблона. Дополнительные сведения см. в статьях о методах CanvasRenderingContext2D.createLinearGradient(), CanvasRenderingContext2D.createRadialGradient() и CanvasRenderingContext2D.createPattern().

float globalAlpha

Определяет уровень непрозрачности рисунка. Диапазон значений от 0,0 (полностью прозрачный) до 1,0 (полностью непрозрачный). Значение по умолчанию – 1,0.

String globalCompositeOperation

Определяет порядок смешения цветов на холсте. Перечень допустимых значений приводится в справочной статье к этому свойству.

String lineCap

Определяет, как должны оканчиваться отображаемые линии. Допустимые значения "butt", "round" и "square". Значение по умолчанию – "butt". Дополнительная информация представлена в справочной статье к этому свойству.

String lineJoin

Определяет характер вывода точек сопряжения линий. Допустимые значения "round", "bevel" и "miter". Значение по умолчанию – "miter". Дополнительная информация представлена в справочной статье к этому свойству.

float lineWidth

Определяет толщину линий для операций рисования. Значение по умолчанию – 1,0; значение этого свойства должно быть больше 0,0. Широкие линии центрируются по воображаемой линии рисования на половину толщины в одну сторону и на половину толщины в другую.

float miterLimit

Когда свойство lineJoin имеет значение "miter", данное свойство определяет отношение длины среза к толщине линии. Дополнительная информация представлена в справочной статье к этому свойству.

float shadowBlur

Определяет степень размытия краев тени. Значение по умолчанию – 0. Тени поддерживаются в броузере Safari и не поддерживаются в броузерах Firefox 1.5 и Opera 9.

`String shadowColor`

Определяет цвет теней в виде CSS- или веб-строк и может включать альфа-компонент, описывающий прозрачность. Значение по умолчанию – `"black"`. Тени поддерживаются в броузере Safari и не поддерживаются в броузерах Firefox 1.5 и Opera 9.

`float shadowOffsetX, shadowOffsetY`

Определяют горизонтальное и вертикальное смещение теней. Чем больше смещение, тем выше объект с тенью кажется расположенным над фоном. Значение по умолчанию – 0. Тени поддерживаются в броузере Safari и не поддерживаются в броузерах Firefox 1.5 и Opera 9.

`Object strokeStyle`

Определяет цвет, шаблон или градиент, используемый для рисования линий. Это свойство может принимать строковое значение или объект `CanvasGradient` или `CanvasPattern`. Когда в это свойство записывается строка, она воспринимается как значение цвета в формате CSS, который используется для заливки линии. Когда в свойство записывается объект `CanvasGradient` или `CanvasPattern`, заливка линии выполняется с помощью указанного градиента или шаблона. Дополнительные сведения см. в статьях о методах `CanvasRenderingContext2D.createLinearGradient()`, `CanvasRenderingContext2D.createRadialGradient()`, `CanvasRenderingContext2D.createPattern()`.

Методы

`arc()`

Добавляет в текущий подпуть перемещения пера дугу с указанными точкой центра и радиусом.

`arcTo()`

Добавляет в текущий подпуть перемещения пера дугу заданного радиуса между начальной и конечной точками.

`beginPath()`

Начинает создание нового описания пути пера (или набора вложенных путей) по холсту.

`bezierCurveTo()`

Добавляет кривую Безье третьего порядка в текущий подпуть.

`clearRect()`

Стирает пикселы в прямоугольной области холста.

`clip()`

Использует текущий путь как область отсечки для последующих операций рисования.

`closePath()`

Закрывает определение текущего пути, если оно еще не было закрыто.

`createLinearGradient()`

Возвращает объект `CanvasGradient`, который представляет линейный градиент.

`createPattern()`

Возвращает объект `CanvasPattern`, который представляет изображение для мозаичного заполнения.

`createRadialGradient()`

Возвращает объект `CanvasGradient`, который представляет радиальный градиент.

`drawImage()`

Рисует изображение.

`fill()`

Заполняет текущий путь движения пера цветом, градиентом или шаблоном, определяемым свойством `fillStyle`.

`fillRect()`

Закрашивает или заполняет прямоугольник.

`lineTo()`

Добавляет отрезок прямой линии в текущее описание пути движения пера.

`moveTo()`

Устанавливает текущую позицию и начало нового вложенного пути движения пера.

`quadraticCurveTo()`

Добавляет в текущий подпуть движения пера кривую Безье второго порядка.

`rect()`

Добавляет прямоугольник в текущий подпуть движения пера.

`restore()`

Восстанавливает последние сохраненные значения параметров холста.

`rotate()`

Выполняет поворот холста.

`save()`

Сохраняет свойства, область отсечки и матрицу преобразования объекта `CanvasRenderingContext2D`.

`scale()`

Масштабирует пользовательскую систему координат холста.

`stroke()`

Рисует линию, следуя заданному пути движения пера. Помимо прочих при рисовании линии учитываются значения свойств `lineWidth`, `lineJoin`, `lineCap` и `strokeStyle`.

`strokeRect()`

Рисует (но не заполняет) прямоугольник.

`translate()`

Преобразует пользовательскую систему координат холста.

Описание

Объект `CanvasRenderingContext2D` предоставляет набор графических функций для выполнения операций рисования. Хотя в нем, к сожалению, отсутствуют функции для работы с текстом, тем не менее набор доступных функций отличается богатством и разнообразием. Они делятся на несколько категорий.

Рисование прямоугольников

Нарисовать и заполнить прямоугольник можно с помощью методов `strokeRect()` и `fillRect()`. Кроме того, методом `clearRect()` можно очистить прямоугольную область.

Рисование изображений

В API объекта Canvas изображения задаются с помощью объектов Image, которые представляют HTML-теги <img> или невидимые изображения, созданные с помощью конструктора Image() (дополнительную информацию см в справочной статье об объекте Image). Кроме того, в качестве объекта-источника изображения может использоваться объект Canvas.

Добавить изображение в элемент Canvas можно с помощью метода drawImage(), который в наиболее общем случае позволяет масштабировать и выводить на экран произвольный прямоугольный участок исходного изображения.

Создание и отображение пути движения пера

Очень мощная особенность элемента Canvas заключается в возможности строить фигуры с помощью элементарных операций рисования и затем отображать их либо в виде ограничивающих фигуру линий, либо заполнять их. Собранные воедино операции рисования называются *текущим путем движения пера*. Элемент Canvas поддерживает возможность работы лишь с одним текущим путем.

Для построения связанной фигуры из отдельных сегментов между операциями рисования должны иметься точки соединения. Для этой цели Canvas поддерживает *текущую позицию*. Операции рисования неявно используют ее в качестве начальной точки и обычно переустанавливают текущую позиции в свою конечную точку. Это можно представить себе как перемещение пера по листу бумаги: когда заканчивается рисование каждой отдельной линии, текущей становится позиция, в которой было остановлено движение пера.

Существует возможность создать в текущем пути несколько несвязанных фигур, которые должны отображаться с одними и теми же параметрами рисования. Для отделения фигур используется метод moveTo(); он перемещает текущую позицию в новые координаты без добавления линии, связывающей точки. Когда вызывается этот метод, создается новый *вложенный путь*, или *подпуть*, который в терминах элемента Canvas используется для объединения связанных операций.

Как только требуемый подпуть движения пера сформирован, нарисовать фигуру в виде ограничивающих линий можно методом stroke(), а залить внутреннюю область фигуры – методом fill(); можно также вызвать оба метода.

Из доступных операций рисования можно упомянуть: lineTo() – рисование отрезков прямых линий, rect() – рисование прямоугольников, arc() и arcTo() – рисование дуг, bezierCurveTo() и quadraticCurveTo() – рисование кривых.

Помимо рисования линий и заполнения фигур текущий путь можно использовать как *область отсечки*. Пикселы в пределах этой области будут отображаться, за ее пределами – нет. Область отсечки обладает свойством накапливать изменения – вызов метода clip() для создания области отсечки, пересекающейся с текущей, приводит к созданию новой объединенной области. К сожалению, нет непосредственной возможности обнулить область отсечки – для этой цели следует использовать операции сохранения и восстановления состояния холста (они описаны в этой статье далее).

Если сегменты в любом из вложенных путей не формируют замкнутую фигуру, операции fill() и clip() неявно замыкают их, добавляя виртуальный (невидимый) отрезок прямой линии, соединяющий начальную и конечную точки пути. Чтобы явно добавить такой сегмент и тем самым замкнуть фигуру, следует вызвать метод closePath().

Цвета, градиенты и шаблоны

При заполнении или рисовании границ фигуры существует возможность указать, каким образом должны заполняться линии или ограниченная ими область, для чего используются свойства `fillStyle` и `strokeStyle`. Оба эти свойства могут принимать строку со значением цвета в формате CSS, а также объект `CanvasGradient` или `CanvasPattern`, который описывает градиент или шаблон. Для создания градиента используется метод `createLinearGradient()` или `createRadialGradient()`, для создания шаблона – метод `createPattern()`.

Непрозрачный цвет в нотации CSS задается строкой в формате `"#RRGGBB"`, где *RR*, *GG* и *BB* – это шестнадцатеричные цифры, определяющие интенсивность красной (red), зеленой (green) и синей (blue) составляющих в диапазоне от 00 до FF. Например, ярко-красный цвет описывается строкой `"#FF0000"`. Чтобы определить степень прозрачности цвета, используется строка в формате `"rgba(R, G, B, A)"`. В такой нотации *R*, *G* и *B* определяют интенсивность красной, зеленой и синей составляющих цвета в виде десятичных чисел в диапазоне от 0 до 255, а *A* – альфа-компонент (прозрачность) как число с плавающей точкой в диапазоне от 0,0 (полностью прозрачный цвет) до 1,0 (полностью непрозрачный цвет). Например, полупрозрачный ярко-красный цвет описывается строкой `"rgba(255, 0, 0, 0.5)"`.

Толщина, окончания и сопряжение линий

Элемент `Canvas` обеспечивает различные варианты отображения линий. Толщину линий можно указать с помощью свойства `lineWidth`, окончания линий – с помощью свойства `lineCap`, сопряжения линий – с помощью свойства `lineJoin`.

Система координат и преобразования

По умолчанию начало системы координат холста находится в точке (0, 0), в верхнем левом углу, когда координата X растет в направлении к правой границе, а координата Y – к нижней. Обычно одна элементарная единица измерения в системе координат соответствует одному пикселу.

Однако существует возможность *преобразовать* систему координат, вызывая смещение, масштабирование или вращение системы координат в операциях рисования. Делается это с помощью методов `translate()`, `scale()` и `rotate()`, которые оказывают влияние на *матрицу преобразования* холста. Поскольку система координат может подвергаться подобным преобразованиям, значения координат, которые передаются методам, таким как `lineTo()`, могут не соответствовать количеству пикселов. По этой причине для определения координат в API объекта `Canvas` используются не целые числа, а числа с плавающей точкой.

Преобразования выполняются в обратном порядке их задания. Так, если сначала вызывается метод `scale()`, а затем `translate()`, то система координат сначала будет сдвинута, а затем масштабирована.

Смешение

Нередко одни фигуры рисуются поверх других, причем верхние фигуры скрывают фигуры, которые оказываются ниже. Это поведение обеспечивается элементом `Canvas` по умолчанию. Однако определяя различные значения свойства `globalCompositeOperation`, можно добиться множества интересных эффектов – от объединения операцией XOR (исключающее ИЛИ) до осветления или затемнения области, расположенной под фигурой. Подробнее об этом рассказывается в статье о свойстве `CanvasRenderingContext2D.globalCompositeOperation`, в которой приводятся все его допустимые значения.

Тени

API объекта Canvas включает свойства, которые могут автоматически добавлять тени к любым создаваемым фигурам. Однако к моменту написания этих строк только броузер Safari имел поддержку этого прикладного интерфейса. Цвет тени задается с помощью свойства shadowColor, а ее смещение – с помощью свойств shadowOffsetX и shadowOffsetY. Кроме того, с помощью свойства shadowBlur можно определить степень размытия краев тени.

Сохранение значений графических параметров

Методы save() и restore() позволяют сохранять и восстанавливать параметры объекта CanvasRenderingContext2D. Метод save() помещает параметры на вершину стека, а метод restore() снимает последние сохраненные параметры с вершины стека и делает их текущими.

Сохраняются все свойства объекта CanvasRenderingContext2D (за исключением свойства canvas, которое является константой). Матрица преобразования и область отсечки также сохраняются на стеке, но текущие путь и позиция пера не сохраняются.

См. также

Canvas

CanvasRenderingContext2D.arc()

добавляет дугу в текущий подпуть с заданными центром окружности и радиусом

Синтаксис

```
void arc(float x, float y, float radius,
         float startAngle, endAngle,
         boolean counterclockwise)
```

Аргументы

x, *y*

Координаты центра окружности, частью которой является дуга.

radius

Радиус окружности, на которой лежит дуга.

startAngle, *endAngle*

Углы, которые определяют начальную и конечную точки дуги. Эти углы измеряются в радианах. Крайняя правая точка окружности соответствует углу 0. Увеличение углов идет в направлении часовой стрелки.

counterclockwise

Признак рисования дуги в направлении против часовой стрелки (true) или по часовой стрелке (false).

Описание

Первые пять аргументов этого метода описывают начальную и конечную точки дуги на окружности. Вызов этого метода добавляет в текущий подпуть отрезок прямой линии между текущей позицией пера и начальной точкой дуги. Затем он добавляет в текущий подпуть дугу, лежащую на окружности между начальной и конечной точками. Последний аргумент определяет направление дуги, соединяющей начальную и конечные точки. Этот метод перемещает текущую позицию пера в конечную точку дуги.

См. также

CanvasRenderingContext2D.arcTo(),CanvasRenderingContext2D.beginPath(),CanvasRenderingContext2D.closePath()

CanvasRenderingContext2D.arcTo()

добавляет дугу в текущий подпуть с заданными точками пересечения с касательными и радиусом

Синтаксис

```
void arcTo(float x1, float y1,
           float x2, float y2,
           float radius)
```

Аргументы

x1, *y1*

Координаты точки P1.

x2, *y2*

Координаты точки P2.

radius

Радиус окружности, на которой лежит дуга.

Описание

Метод добавляет в текущий подпуть дугу, но описывает эту дугу несколько иначе, чем метод arc(). Дуга, добавляемая в путь, является частью окружности с заданным радиусом *radius*. Начальная точка дуги соответствует точке пересечения с касательной, соединяющей текущую позицию пера и точку P1, а конечная соответствует точке пересечения с касательной, соединяющей точки P1 и P2. Дуга соединяет начальную и конечную точки в кратчайшем направлении.

В большинстве случаев дуга начинается в текущей позиции и заканчивается в точке P2, но не всегда. Если координаты текущей позиции и начальной точки дуги не совпадают, этот метод соединяет их отрезком прямой линии. Данный метод всегда перемещает текущую позицию пера в конечную точку дуги.

Пример

Нарисовать правый верхний угол прямоугольника с закруглением можно следующим образом:

```
c.moveTo(10,10);          // Переход в левый верхний угол
c.lineTo(90, 10)          // горизонтальная линия до начала закругленного угла
c.arcTo(100, 10, 100, 20, 10); // Закругление
c.lineTo(100, 100);            // Вертикальная линия до правого нижнего угла
```

Ошибки

Этот метод не реализован в Firefox 1.5.

См. также

CanvasRenderingContext2D.arc()

CanvasRenderingContext2D.beginPath()

начинает новый подпуть в элементе Canvas

Синтаксис

```
void beginPath()
```

Описание

Метод `beginPath()` отменяет любое существующее определение пути пера и начинает новый. Текущей позицией пера становится точка с координатами (0,0).

При создании объекта-контекста холста первый раз метод `beginPath()` вызывается неявно.

См. также

`CanvasRenderingContext2D.closePath()`, `CanvasRenderingContext2D.fill()`, `CanvasRenderingContext2D.stroke()`; глава 22

CanvasRenderingContext2D.bezierCurveTo()

добавляет кривую Безье третьего порядка в текущий подпуть пера

Синтаксис

```
void bezierCurveTo(float cpX1, float cpY1,
                   float cpX2, float cpY2,
                   float x, float y)
```

Аргументы

cpX1, *cpY1* Координаты контрольной точки, ассоциированной с начальной точкой кривой (текущей позицией).

cpX2, *cpY2* Координаты контрольной точки, ассоциированной с конечной точкой кривой.

x, *y* Координаты конечной точки кривой.

Описание

Метод `bezierCurveTo()` добавляет кривую Безье третьего порядка в текущий подпуть пера элемента `Canvas`. Начальная точка кривой находится в текущей позиции, а координаты конечной точки определяются аргументами *x* и *y*. Форма кривой Безье определяется контрольными точками (`cpX1, cpY1`) и (`cpX2, cpY2`). По возвращении из метода текущей позицией становится точка (`x, y`).

См. также

`CanvasRenderingContext2D.quadraticCurveTo()`

CanvasRenderingContext2D.clearRect()

стирает содержимое прямоугольной области

Синтаксис

```
void clearRect(float x, float y,
               float width, float height)
```

Аргументы

x, *y*

Координаты верхнего левого угла прямоугольника.

width, *height*

Ширина и высота прямоугольника.

Описание

Метод `clearRect()` стирает содержимое в заданной прямоугольной области, заполняя ее прозрачным цветом.

CanvasRenderingContext2D.clip()

устанавливает область отсечки

Синтаксис

```
void clip()
```

Описание

Этот метод обрезает текущий путь, используя для этого существующую область отсечки, и затем задействует получившийся обрезанный путь как новую область отсечки. Обратите внимание: нет никакого способа увеличить путь отсечки. Если область отсечки требуется лишь на время, сначала следует вызвать метод `save()`, чтобы затем с помощью `restore()` восстановить прежнюю область отсечки. По умолчанию область отсечки совпадает с границами холста.

Этот метод сбрасывает (обнуляет) текущий путь.

CanvasRenderingContext2D.closePath()

закрывает открытый подпуть

Синтаксис

```
void closePath()
```

Описание

Если текущий подпуть пера остается открытым, `closePath()` закроет его, добавив отрезок прямой, соединяющей начальную и конечную точки. Если подпуть уже закрыт, этот метод ничего делать не будет. После того как подпуть закроется, к нему невозможно будет добавить дополнительные прямые или кривые. Чтобы продолжить процесс рисования в текущем пути, необходимо методом `moveTo()` начать создание нового подпути.

Перед вызовом метода `stroke()` или `fill()` вызывать метод `closePath()` совсем необязательно. Путь движения пера будет закрыт неявно перед заливкой фигуры (а также при вызове метода `clip()`).

См. также

`CanvasRenderingContext2D.beginPath()`, `CanvasRenderingContext2D.moveTo()`, `CanvasRenderingContext2D.stroke()`, `CanvasRenderingContext2D.fill()`

CanvasRenderingContext2D.createLinearGradient()

создает линейный цветной градиент

Синтаксис

```
CanvasGradient createLinearGradient(float xStart, float yStart,
                                    float xEnd, float yEnd)
```

Аргументы

xStart, *yStart*

Координаты начальной точки градиента.

xEnd, *yEnd*

Координаты конечной точки градиента.

Возвращаемое значение

Объект `CanvasGradient`, представляющий линейный цветной градиент.

Описание

Этот метод создает и возвращает новый объект `CanvasGradient`, который выполняет линейную интерполяцию цветов между заданными начальной и конечной точками. Обратите внимание: этот метод не определяет цвета градиента. Для этих целей следует использовать метод `addColorStop()` вновь созданного объекта. Чтобы рисовать линии или заполнять фигуры с помощью градиента, необходимо присвоить объект `CanvasGradient` свойству `strokeStyle` или `fillStyle`.

См. также

`CanvasGradient.addColorStop()`, `CanvasRenderingContext2D.createRadialGradient()`

CanvasRenderingContext2D.createPattern()

создает шаблон мозаичного изображения

Синтаксис

```
CanvasPattern createPattern(Image image,
                            String repetitionStyle)
```

Аргументы

image

Изображение, которое будет использоваться для создания мозаики. Обычно в этом аргументе передается объект `Image`, но это может быть и элемент `Canvas`.

repetitionStyle

Определяет, как будет выкладываться мозаика. Ниже перечислены допустимые значения:

Значение	Смысл
`"repeat"`	Изображение выкладывается мозаикой в обоих направлениях. Это значение по умолчанию
`"repeat-x"`	Изображение выкладывается мозаикой только по оси X
`"repeat-y"`	Изображение выкладывается мозаикой только по оси Y
`"no-repeat"`	Изображение мозаики не повторяется, а используется однократно

Возвращаемое значение

Объект CanvasPattern, представляющий шаблон.

Описание

Этот метод создает и возвращает объект CanvasPattern, который представляет шаблон, определяемый повторяющимся изображением. Чтобы рисовать линии или заполнять фигуры с использованием шаблона, необходимо присвоить объект CanvasPattern свойству strokeStyle или fillStyle.

Ошибки

Firefox 1.5 поддерживает только стиль repeat. Остальные значения игнорируются.

См. также

CanvasPattern

CanvasRenderingContext2D.createRadialGradient()

создает радиальный цветной градиент

Синтаксис

```
CanvasGradient createRadialGradient(float xStart, float yStart, float radiusStart,
                                    float xEnd, float yEnd, float radiusEnd)
```

Аргументы

xStart, *yStart*

Координаты центра начальной окружности.

radiusStart

Радиус начальной окружности.

xEnd, *yEnd*

Координаты центра конечной окружности

radiusEnd

Радиус конечной окружности.

Возвращаемое значение

Объект CanvasGradient, представляющий радиальный цветной градиент.

Описание

Этот метод создает и возвращает новый объект CanvasGradient, который выполняет радиальную интерполяцию цветов между двумя заданными окружностями. Обратите внимание: этот метод не определяет цвета градиента. Для этих целей следует использовать метод addColorStop() вновь созданного объекта. Чтобы рисовать линии или заполнять фигуры с помощью градиента, необходимо присвоить объект CanvasGradient свойству strokeStyle или fillStyle.

Радиальные градиенты отображаются с использованием цвета со смещением 0 для первой окружности, со смещением 1 для второй окружности и интерполированными цветами (красная, зеленая и синяя составляющие, а также альфа-компонент) для рисования промежуточных окружностей.

См. также

CanvasGradient.addColorStop(), CanvasRenderingContext2D.createLinearGradient()

CanvasRenderingContext2D.drawImage()

рисует изображение

Синтаксис

```
void drawImage(Image image, float x, float y)

void drawImage(Image image, float x, float y,
               float width, float height)

void drawImage(Image image, integer sourceX, integer sourceY,
               integer sourceWidth, integer sourceHeight,
               float destX, float destY,
               float destWidth, float destHeight)
```

Аргументы

image

Изображение, которое должно быть нарисовано. Это может быть объект `Image`, представляющий тег `<img>`, невидимое изображение, созданное конструктором `Image()`, или объект `Canvas`.

x, *y*

Координаты верхнего левого угла изображения.

width, *height*

Размеры, в которые должно поместиться изображение. С помощью этих параметров можно выполнять масштабирование изображения.

sourceX, *sourceY*

Верхний левый угол выводимого фрагмента изображения. Эти целочисленные аргументы измеряются в пикселах изображения.

destX, *destY*

Координаты холста, куда должен быть помещен верхний левый угол фрагмента изображения.

destWidth, *destHeight*

Размеры, в которые должен поместиться фрагмент изображения.

Описание

Существует три разновидности этого метода. Первая копирует изображение на холст целиком, помещая верхний левый угол в заданные координаты и отображая каждый пиксел изображения на элемент системы координат холста. Вторая разновидность также копирует изображение на холст целиком, но позволяет указать желаемые ширину и высоту изображения в единицах системы координат холста. Третья разновидность является самой универсальной: она позволяет определить прямоугольный фрагмент изображения и скопировать его с произвольным масштабированием в любую точку внутри холста.

Изображения, передаваемые этому методу, должны быть либо объектами `Image`, либо объектами `Canvas`. Объект `Image` может представлять тег `<img>` в документе или быть невидимым изображением, созданным вызовом конструктора `Image()`.

См. также

`Image`

CanvasRenderingContext2D.fill()

выполняет заливку текущего пути

Синтаксис

```
void fill()
```

Описание

Метод `fill()` заполняет текущий путь цветом, градиентом или шаблоном, заданным свойством `fillStyle`. Каждый подпуть в пути заполняется независимо от другого. Любой подпуть, который не был замкнут, заполняется так, как если бы для него неявно был вызван метод `closePath()`. (Обратите внимание: это не означает, что вызов этого метода сделает подпуть замкнутым.)

Чтобы определить, какие точки находятся внутри пути, а какие снаружи, элемент `Canvas` использует правило «non-zero winding rule». Описание этого правила выходит за рамки темы данной книги, но оно обычно имеет существенное значение лишь для сложных самопересекающихся путей.

Заполнение пути не очищает его. Можно сразу вслед за методом `fill()` вызвать метод `stroke()`, не выполняя повторное определение пути.

См. также `CanvasRenderingContext2D.fillRect()`

CanvasRenderingContext2D.fillRect()

заполняет прямоугольник

Синтаксис

```
void fillRect(float x, float y,
              float width, float height)
```

Аргументы

x, *y*

Координаты верхнего левого угла прямоугольника.

width, *height*

Размеры прямоугольника.

Описание

Метод `fillRect()` выполняет заливку заданного прямоугольника цветом, градиентом или шаблоном, который задается свойством `fillStyle`.

Кроме того, существующие реализации `fillRect()` очищают путь, как если бы был вызван метод `beginPath()`. Такое поведение может быть не стандартизовано и на него не следует полагаться.

См. также

`CanvasRenderingContext2D.fill()`, `CanvasRenderingContext2D.rect()`, `CanvasRenderingContext2D.strokeRect()`

CanvasRenderingContext2D.globalCompositeOperation

определяет порядок смешения цветов на холсте

Синтаксис

```
String globalCompositeOperation
```

Описание

Данное свойство определяет, как выводимые на холст цвета должны смешиваться с уже существующими цветами. В следующей таблице перечислены все допустимые значения и их назначение. Под словом «source» в этих значениях понимаются цвета, которые выводятся на холст, а под словом «destination» – цвета, которые уже имеются на холсте. Значение по умолчанию – `"source-over"`.

Значение	Смысл
`"copy"`	Рисуется только новая фигура, все остальное стирается
`"darker"`	Когда две фигуры перекрывают друг друга, результирующий цвет области перекрытия определяется как разность значений цветов
`"destination-atop"`	Существующее содержимое сохраняется только там, где оно перекрывает новую фигуру. Новая фигура рисуется позади существующего содержимого
`"destination-in"`	Существующее содержимое сохраняется, где новая фигура и существующее содержимое перекрываются. Все остальное делается прозрачным
`"destination-out"`	Существующее содержимое сохраняется, где оно не перекрывает новую фигуру. Все остальное делается прозрачным
`"destination-over"`	Новая фигура рисуется позади существующего содержимого
`"lighter"`	Когда две фигуры перекрывают друг друга, результирующий цвет определяется как сумма значений цветов
`"source-atop"`	Новая фигура рисуется только там, где она накладывается на существующее содержимое
`"source-in"`	Новая фигура рисуется только там, где новая фигура и существующее содержимое перекрываются. Все остальное делается прозрачным
`"source-out"`	Новая фигура рисуется там, где она не накладывается на существующее содержимое
`"source-over"`	Новая фигура рисуется поверх существующего содержимого. Это поведение по умолчанию
`"xor"`	Фигуры делаются прозрачными в местах перекрытия, остальные части фигур рисуются обычным образом

Ошибки

Firefox 1.5 не поддерживает значения `"copy"` и `"darker"`.

CanvasRenderingContext2D.lineCap

определяет, как должны рисоваться окончания линий

Синтаксис

```
String lineCap
```

Описание

Свойство `lineCap` определяет, как должны рисоваться окончания линий. Устанавливать это свойство имеет смысл только при рисовании толстых линий. Допустимые значения перечислены в следующей таблице. Значение по умолчанию – `"butt"`.

Значение	Смысл
`"butt"`	Это значение по умолчанию. Оно указывает на то, что окончания линий не должны рисоваться. В этом случае окончание линии выглядит просто как отрезок, перпендикулярный к боковым сторонам линии. Линия не выступает за свои конечные точки
`"round"`	Это значение указывает, что линия должна иметь окончание в виде полукруга с диаметром, равным толщине линии, в результате линия выступает за конечные точки на половину своей толщины
`"square"`	Это значение указывает, что линия должна иметь окончание в виде квадрата. Это значение по своему поведению напоминает значение `"butt"`, но при использовании данного значения линия выступает за конечные точки на половину своей толщины

Ошибки

В Firefox 1.5 некорректно реализована поддержка значения `"butt"`. В этом броузере по своему поведению это значение полностью совпадает со значением `"square"`.

См. также `CanvasRenderingContext2D.lineJoin`

CanvasRenderingContext2D.lineJoin

определяет, как должны рисоваться сопряжения линий в вершинах

Синтаксис

```
String lineJoin
```

Описание

Когда путь включает вершины, где соединяются прямые линии и/или кривые, свойство `lineJoin` определяет, как должны рисоваться эти вершины. Действие этого свойства проявляется только при рисовании толстых линий.

Значение по умолчанию, `"miter"`, указывает, что внешние края двух линий в точке сопряжения должны быть продолжены, пока они не пересекутся. Когда две линии сопрягаются под очень острым углом, область сопряжения может оказаться достаточно длинной. Ограничить максимальную длину такого варианта сопряжения можно с помощью свойства `miterLimit`. Когда длина сопряжения превышает этот предел, сопряжение просто усекается.

Значение `"round"` указывает, что внешние края линий, образующих вершину, должны сопрягаться закрашенной дугой, диаметр которой равен толщине линий.

Значение `"bevel"` указывает, что внешние края линий, образующих вершину, должны сопрягаться закрашенным треугольником.

Ошибки

В Firefox 1.5 некорректно реализовано сопряжение в стиле `"bevel"`, из-за чего сопряжения отображаются как закругления. Кроме того, когда линии рисуются полупрозрачным цветом, некорректно отображаются сопряжения в стиле `"miter"`.

См. также CanvasRenderingContext2D.lineCap, CanvasRenderingContext2D.miterLimit

CanvasRenderingContext2D.lineTo()

добавляет прямую линию в текущий подпуть

Синтаксис

```
void lineTo(float x, float y)
```

Аргументы

x, *y* Координаты конечной точки линии.

Описание

Метод lineTo() добавляет в текущий подпуть прямую линию. Линия начинается в текущей позиции пера и заканчивается в точке с координатами (x, y). Когда этот метод возвращает управление, текущая позиция перемещается в точку (x, y).

См. также

CanvasRenderingContext2D.beginPath(), CanvasRenderingContext2D.moveTo()

CanvasRenderingContext2D.miterLimit

максимальное отношение длины сопряжения линий к толщине

Синтаксис

```
float miterLimit
```

Описание

Когда толстые линии сопрягаются под очень острым углом и при этом в свойстве lineJoin установлено значение "miter", область сопряжения может оказаться достаточно длинной. Слишком длинная область сопряжения может выглядеть достаточно некрасиво. Свойство miterLimit позволяет определить максимальную длину сопряжения. Величина этого свойства выражает отношение длины области сопряжения к толщине линий. Значение по умолчанию – 10, оно означает, что область сопряжения по длине никогда не должна превышать толщину линий более чем в 10 раз. Когда длина сопряжения превышает этот предел, оно просто усекается. Значение этого свойства игнорируется, когда в свойстве lineJoin установлено значение "round" или "bevel".

Ошибки

В Firefox 1.5 поведение этого свойства реализовано некорректно. Если длина сопряжения превышает значение свойства miterLimit, оно преобразуется в сопряжение с закруглением ("round").

См. также CanvasRenderingContext2D.lineJoin

CanvasRenderingContext2D.moveTo()

устанавливает текущую позицию пера и начинает новый подпуть

Синтаксис

```
void moveTo(float x, float y)
```

Аргументы

x, *y* Координаты новой текущей позиции пера.

Описание

Метод `moveTo()` переносит текущую позицию пера в точку (*x*, *y*) и создает новый подпуть с начальной точкой в этих координатах. Если перед этим существовал подпуть, состоящий из единственной точки, этот подпуть удаляется из текущего пути.

См. также `CanvasRenderingContext2D.beginPath()`

CanvasRenderingContext2D.quadraticCurveTo()

добавляет кривую Безье второго порядка в текущий подпуть пера

Синтаксис

```
void bezierCurveTo(float cpX, float cpY,
                   float x, float y)
```

Аргументы

cpX, *cpY* Координаты контрольной точки.

x, *y* Координаты конечной точки кривой.

Описание

Данный метод добавляет кривую Безье второго порядка в текущий подпуть пера. Начальная точка кривой находится в текущей позиции, а координаты конечной точки определяются аргументами *x* и *y*. Форма кривой Безье, соединяющей эти две точки, определяется контрольной точкой (`cpX, cpY`). По возвращении из метода текущей позицией становится точка (`x,y`).

Ошибки

В Firefox 1.5 этот метод реализован некорректно.

См. также `CanvasRenderingContext2D.bezierCurveTo()`

CanvasRenderingContext2D.rect()

добавляет подпуть, описывающий прямоугольник, в текущий путь

Синтаксис

```
void rect(float x, float y,
          float width, float height)
```

Аргументы

x, *y* Координаты верхнего левого угла прямоугольника.

width, *height* Размеры прямоугольника.

Описание

Этот метод добавляет прямоугольник в путь пера. Прямоугольник представляет собой отдельный подпуть, который никак не связан ни с одним из имеющихся подпутей. По возвращении из метода текущей позицией становится точка (0,0).

См. также `CanvasRenderingContext2D.fillRect()`, `CanvasRenderingContext2D.strokeRect()`

CanvasRenderingContext2D.restore()

устанавливает параметры холста в ранее сохраненные значения

Синтаксис

```
void restore()
```

Описание

Метод снимает с вершины стека значения параметров холста и записывает их в свойства объекта CanvasRenderingContext2D, восстанавливая область отсечки и матрицу преобразования. Дополнительные сведения см. в статье о методе save().

Ошибки

В Firefox 1.5 сохранение и восстановление свойства strokeStyle выполняется некорректно.

См. также CanvasRenderingContext2D.save()

CanvasRenderingContext2D.rotate()

выполняет поворот системы координат холста

Синтаксис

```
void rotate(float angle)
```

Аргументы

angle

Угол поворота в радианах. Положительные значения соответствуют повороту по часовой стрелке, отрицательные – против часовой стрелки.

Описание

Данный метод изменяет порядок отображения координат холста на систему координат элемента <canvas> таким образом, что любые фигуры, нарисованные после вызова этого метода, выглядят повернутыми на указанный угол. Этот метод не выполняет вращение самого элемента <canvas>. Обратите внимание: угол задается в радианах. Чтобы преобразовать градусы в радианы, нужно умножить величину угла на константу Math.PI и разделить на число 180.

См. также CanvasRenderingContext2D.scale(), CanvasRenderingContext2D.translate()

CanvasRenderingContext2D.save()

сохраняет копию текущих параметров холста

Синтаксис

```
void save()
```

Описание

Метод save() помещает копию текущих параметров холста на вершину стека сохраняемых параметров. Это позволяет внести временные изменения в какие-либо параметры и затем восстановить предыдущие значения вызовом метода restore().

В перечень сохраняемых параметров входят все свойства объекта CanvasRenderingContext2D (за исключением доступного только для чтения свойства canvas), а также матрица преобразования, которая является результатом вызова методов rotate(), scale() и translate(). Кроме того, в стеке сохраняется и область отсечки, созданная методом clip(). Однако следует заметить, что текущие путь и позиция пера не входят в данный перечень и этим методом не сохраняются.

Ошибки

В Firefox 1.5 значение свойства strokeStyle не сохраняется и не восстанавливается.

См. также CanvasRenderingContext2D.restore()

CanvasRenderingContext2D.scale()

выполняет масштабирование системы координат холста

Синтаксис

```
void scale(float sx, float sy)
```

Аргументы

sx, *sy* Масштабные множители по горизонтали и по вертикали.

Описание

Метод scale() добавляет преобразование масштаба в текущую матрицу преобразования холста. Масштабирование выполняется отдельно по горизонтали и по вертикали. Например, если передать методу значения 2,0 и 0,5, все последующие фигуры будут иметь в два раза большую ширину и в два раза меньшую высоту по сравнению с тем, как они если бы они были нарисованы до вызова метода scale(). Отрицательные значения аргумента *sx* вызывают зеркальное отражение координат относительно оси Y, а отрицательные значения аргумента *sy* вызывают зеркальное отражение координат относительно оси X.

См. также

CanvasRenderingContext2D.rotate(), CanvasRenderingContext2D.translate()

CanvasRenderingContext2D.stroke()

выполняет рисование текущего пути

Синтаксис

```
void stroke()
```

Описание

Метод stroke() выполняет рисование линий, составляющих текущий путь. Путь определяет геометрию линии, которая должна быть воспроизведена, но визуальное представление линии зависит от значений свойств strokeStyle, lineWidth, lineCap, lineJoin и miterLimit.

Под термином *stroke* (*чертить*) понимается вычерчивание линий пером или кистью. Это означает «нарисовать контур». В противовес методу stroke(), метод fill() выполняет заливку внутренней области пути без рисования ее контура.

См. также

CanvasRenderingContext2D.fill(), CanvasRenderingContext2D.lineCap, CanvasRenderingContext2D.lineJoin, CanvasRenderingContext2D.strokeRect()

CanvasRenderingContext2D.strokeRect()

выполняет рисование прямоугольника

Синтаксис

```
void strokeRect(float x, float y,
                float width, float height)
```

Аргументы

x, *y*

Координаты верхнего левого угла прямоугольника.

width, *height*

Размеры прямоугольника.

Описание

Этот метод рисует контур (не выполняя заливку внутренней области) прямоугольника с заданными координатами и размерами. Цвет и толщина линий определяются значениями свойств strokeStyle и lineWidth. Стиль оформления сопряжений в углах прямоугольника определяется значением свойства lineJoin.

Существующие реализации strokeRect() очищают путь, как если бы был вызван метод beginPath(). Такое поведение может быть не стандартизовано и на него не следует полагаться.

См. также

CanvasRenderingContext2D.fillRect(), CanvasRenderingContext2D.lineJoin, CanvasRenderingContext2D.rect(), CanvasRenderingContext2D.stroke()

CanvasRenderingContext2D.translate()

выполняет сдвиг пользовательской системы координат холста

Синтаксис

```
void translate(float dx, float dy)
```

Аргументы

dx, *dy*

Величина смещения по осям X и Y.

Описание

Метод translate() добавляет горизонтальное и вертикальное смещения в матрицу преобразования холста. Значения аргументов *dx* и *dy* добавляются к координатам всех точек, которые затем будут добавлять в путь.

См. также

CanvasRenderingContext2D.rotate(), CanvasRenderingContext2D.scale()

CDATASection

DOM Level 1 XML

узел CDATA XML-документа **Node→CharacterData→Text→CDATASection**

Описание

Этот редко используемый интерфейс представляет раздел CDATA в XML-документе. Программисты, работающие с HTML-документами, никогда не встретят узлов данного типа, и этот интерфейс им не потребуется.

CDATASection – это подынтерфейс интерфейса Text; он не определяет никаких собственных свойств или методов. Текстовое содержимое раздела CDATA доступно через свойство nodeValue, унаследованное от Node, или через свойство data, унаследованное от CharacterData. Несмотря на то, что узлы CDATASection часто рассматриваются так же, как узлы Text, обратите внимание, что Node.normalize() не выполняет слияния соседних разделов CDATA. Создается CDATASection методом Document.createCDATASection().

См. также

CharacterData, Text

CharacterData

DOM Level 1 Core

общая функциональность узлов Text и Comment **Node→CharacterData**

Подынтерфейсы

Comment, Text

Свойства

`String data`

Текст, содержащийся в узле.

`readonly unsigned long length`

Количество символов, содержащихся в узле.

Методы

`appendData()`

Дописывает указанную строку к тексту данного узла.

`deleteData()`

Удаляет из данного узла текст заданной длины, начиная с символа с указанным смещением.

`insertData()`

Вставляет строку в текст узла, начиная с символа с заданным смещением.

`replaceData()`

Заменяет указанное количество символов заданной строкой, начиная с указанного текстового смещения.

`substringData()`

Возвращает копию текста заданной длины, начиная с указанного символа.

Описание

CharacterData – это надынтерфейс для узлов Text и Comment. Документы никогда не содержат узлов CharacterData; они содержат только узлы Text и Comment. Однако посколь-

ку функции, выполняемые узлами этих типов, аналогичны, надинтерфейс CharacterData определен в этой реализации, чтобы узлы Text и Comment могли ее наследовать.

См. также Comment, Text

CharacterData.appendData() DOM Level 1 Core

добавляет строку к узлу Text или Comment

Синтаксис

```
void appendData(String arg)
    throws DOMException;
```

Аргументы

arg

Строка, добавляемая к узлу Text или Comment.

Исключения

Этот метод генерирует исключение DOMException со значением code — NO_MODIFICATION_ALLOWED_ERR, если он вызывается для узла, доступного только для чтения.

Описание

Этот метод дописывает строку *arg* в конец свойства *data* для данного узла.

CharacterData.deleteData() DOM Level 1 Core

удаляет символы из узла Text или Comment

Синтаксис

```
void deleteData(unsigned long смещение,
                unsigned long количество)
    throws DOMException;
```

Аргументы

смещение

Позиция первого удаляемого символа.

количество

Количество удаляемых символов.

Исключения

Этот метод может генерировать исключение DOMException с одним из следующих значений code:

INDEX_SIZE_ERR

Если аргумент *смещение* или *количество* отрицателен или если смещение больше длины узла Text или Comment.

NO_MODIFICATION_ALLOWED_ERR

Узел доступен только для чтения и не может модифицироваться.

Описание

Этот метод удаляет заданное количество символов из узла Text или Comment, начиная с символа в позиции *смещение*. Если *смещение* плюс *количество* больше, чем число символов в узле Text или Comment, удаляются все символы, начиная с символа в позиции *смещение* до конца строки.

CharacterData.insertData() DOM Level 1 Core

вставляет строку в узел Text или Comment

Синтаксис

```
void insertData(unsigned long смещение,
                String arg)
    throws DOMException;
```

Аргументы

смещение

Позиция символа внутри узла Text или Comment, куда должна вставляться строка.

arg

Вставляемая строка.

Исключения

Этот метод может генерировать исключение DOMException с одним из перечисленных далее значений code в следующих ситуациях:

INDEX_SIZE_ERR

Значение аргумента *смещение* отрицательно или больше длины узла Text или Comment.

NO_MODIFICATION_ALLOWED_ERR

Узел доступен только для чтения и не может быть изменен.

Описание

Этот метод вставляет указанную строку *arg* в текст узла Text или Comment в указанной позиции *смещение*.

CharacterData.replaceData() DOM Level 1 Core

заменяет символы узла Text или Comment на строку

Синтаксис

```
void replaceData(unsigned long смещение,
                 unsigned long количество,
                 String arg)
    throws DOMException;
```

Аргументы

смещение

Позиция символа в узле Text или Comment, с которой должна начаться замена.

количество

Количество заменяемых символов.

arg

Строка, заменяющая символы, заданные аргументами *смещение* и *количество*.

Исключения

Этот метод может генерировать исключение DOMException с перечисленными далее значениями code при следующих обстоятельствах:

INDEX_SIZE_ERR

Значение аргумента *смещение* отрицательно или больше, чем длина узла Text или Comment, либо значение аргумента *количество* отрицательно.

NO_MODIFICATION_ALLOWED_ERR

Узел доступен только для чтения и не может быть изменен.

Описание

Этот метод заменяет указанное *количество* символов, начиная с позиции *смещение*, содержимым строки *arg*. Если сумма *смещения* и *количества* больше длины узла Text или Comment, заменяются все символы, начиная с символа в позиции *смещение*.

CharacterData.substringData() DOM Level 1 Core

извлекает подстроку из узла Text или Comment

Синтаксис

```
String substringData(unsigned long смещение,
                     unsigned long количество)
    throws DOMException;
```

Аргументы

смещение

Позиция первого возвращаемого символа.

количество

Число символов в возвращаемой подстроке.

Возвращаемое значение

Строка, состоящая из указанного *количества* символов узла Text или Comment, начиная с позиции *смещение*.

Исключения

Метод может генерировать исключение DOMException с такими значениями code:

INDEX_SIZE_ERR

Значение аргумента *смещение* отрицательно или больше, чем длина узла Text или Comment, либо значение аргумента *количество* отрицательно.

DOMSTRING_SIZE_ERR

Указанный диапазон слишком велик, чтобы помещаться в строку для реализации JavaScript в данном броузере.

Описание

Этот метод извлекает подстроку, образованную заданным *количеством* символов, начиная с позиции *смещение*, из текста узла Text или Comment. Метод полезен, только когда размер содержащегося в узле текста больше максимального числа символов, которые могут быть помещены в строку для реализации JavaScript в данном броузере. В этом случае JavaScript-программа не может непосредственно использовать свойство data узла Text или Comment, а должна вместо этого работать с подстроками меньшего размера из текста узла. На практике такая ситуация маловероятна.

Checkbox

См. статью об объекте Input

Comment

DOM Level 1 Core

HTML- или XML-комментарий **Node→CharacterData→Comment**

Описание

Узел Comment представляет комментарий в HTML- или XML-документе. Содержимое комментария (т. е. текст между <!-- и -->) доступно через свойство data, унаследованное от интерфейса CharacterData, или через свойство nodeValue, унаследованное от интерфейса Node. Этим содержимым можно манипулировать различными методами, унаследованными от CharacterData. Создать объект Comment можно методом Document.createComment().

См. также

CharacterData

CSS2Properties

DOM Level 2 CSS

набор CSS-атрибутов и их значений **Object→CSS2Properties**

Свойства

String cssText

Текстовое представление набора атрибутов стилей и их значений. Текст в этом свойстве отформатирован так же, как и в таблице CSS-стилей, минус селектор элемента и фигурные скобки, окружающие атрибуты и значения. При попытке записать в это свойство недопустимое значение возбуждается исключение DOMException со значением SYNTAX_ERR в свойстве code. Попытка установить значение свойства cssText, когда объект CSS2Properties доступен только для чтения, приводит к возбуждению исключения DOMException со значением NO_MODIFICATION_ALLOWED_ERR в свойстве code.

Помимо свойства cssText объект CSS2Properties обладает по одному свойству для каждого CSS-атрибута, определяемого спецификацией CSS2. Имена этих свойств в основном соответствуют именам CSS-атрибутов с небольшими изменениями, необходимыми для того, чтобы избежать синтаксических ошибок в JavaScript. Атрибуты, имена которых состоят из нескольких слов и содержат дефисы, такие как "font-family", в JavaScript записываются без дефисов, а каждое слово, кроме первого, пишется с прописной буквы (например: fontFamily). Кроме того, атрибут "float" конфликтует с зарезервированным словом float, поэтому он транслируется в свойство cssFloat.

Имена свойств объекта CSS2Properties, соответствующие атрибутам, определяемым спецификацией CSS2, перечислены в следующей таблице. Обратите внимание: некоторые броузеры поддерживают не все CSS-атрибуты, потому реализация некоторых из нижеперечисленных свойств может отсутствовать. Так как свойства непосредственно соответствуют CSS-атрибутам, для всех этих свойств отсутствуют самостоятельные описания. Смысл и допустимые значения CSS-атрибутов описаны в литературе по CSS, например в книге издательства O'Reilly «Cascading Style Sheets: The Definitive Guide»[1] Эрика Майера (Eric A. Meyer). Все эти свойства представляют собой строки. Присваивание значений этим свойствам может привести к тем же исключениям и по тем же причинам, что и попытка записи в свойство cssText.

azimuth	background	backgroundAttachment	backgroundColor
backgroundImage	backgroundPosition	backgroundRepeat	border
borderBottom	borderBottomColor	borderBottomStyle	borderBottomWidth
borderCollapse	borderColor	borderLeft	borderLeftColor
borderLeftStyle	borderLeftWidth	borderRight	borderRightColor
borderRightStyle	borderRightWidth	borderSpacing	borderStyle
borderTop	borderTopColor	borderTopStyle	borderTopWidth
borderWidth	bottom	captionSide	clear
clip	color	content	counterIncrement
counterReset	cssFloat	cue	cueAfter
cueBefore	cursor	direction	display
elevation	emptyCells	font	fontFamily
fontSize	fontSizeAdjust	fontStretch	fontStyle
fontVariant	fontWeight	height	left
letterSpacing	lineHeight	listStyle	listStyleImage
listStylePosition	listStyleType	margin	marginBottom
marginLeft	marginRight	marginTop	markerOffset
marks	maxHeight	maxWidth	minHeight
minWidth	orphans	outline	outlineColor
outlineStyle	outlineWidth	overflow	padding
paddingBottom	paddingLeft	paddingRight	paddingTop
page	pageBreakAfter	pageBreakBefore	pageBreakInside
pause	pauseAfter	pauseBefore	pitch
pitchRange	playDuring	position	quotes
richness	right	size	speak
speakHeader	speakNumeral	speakPunctuation	speechRate
stress	tableLayout	textAlign	textDecoration

[1] Эрик Мейер «CSS – каскадные таблицы стилей. Подробное руководство», 3-е издание. – Пер. с англ. – СПб.: Символ-Плюс, 2008.

textIndent	textShadow	textTransform	top
unicodeBidi	verticalAlign	visibility	voiceFamily
volume	whiteSpace	widows	width
wordSpacing	zIndex		

Описание

Объект CSS2Properties представляет набор атрибутов CSS-стилей и их значения. Он определяет по одному свойству для каждого CSS-атрибута, определенного в спецификации CSS2. Свойство style элемента HTMLElement представляет собой объект CSS2Properties, доступный для чтения/записи, так же как и свойство style объекта CSSRule. Однако метод Window.getComputedStyle() возвращает объект CSS2Properties, доступный только для чтения.

См. также CSSRule, HTMLElement, Window.getComputedStyle(); глава 16

CSSRule

DOM Level 2 CSS, IE 5

правило в таблице стилей CSS **Object→CSSRule**

Свойства

String selectorText

Содержит текст селектора, определяющий элемент документа, к которому применяется данное стилевое правило. Установка этого свойства вызывает возбуждение исключения DOMException со значением NO_MODIFICATION_ALLOWED_ERR в свойстве code, если правило доступно только для чтения, или со значением SYNTAX_ERR, если новое значение не соответствует синтаксису правил CSS.

readonly CSS2Properties style

Значения стиля, применяемые к элементу, определяемому свойством selectorText. Обратите внимание: несмотря на то что само свойство style доступно только для чтения, объект CSS2Properties, на который оно ссылается, доступен как для чтения, так и для записи.

Описание

Объект CSSRule является представлением правила в таблице CSS-стилей: он дает информацию о стилях, которые должны применяться к определенному набору элементов документа. Свойство selectorText – это строковое представление селектора элемента для данного правила, а свойство style – ссылка на объект CSS2Properties, который представляет набор атрибутов стилей, применяемых к выбранным элементам.

Спецификация DOM Level 2 для модуля CSS фактически определяет более сложную иерархию интерфейсов CSSRule для представления различных типов правил, которые могут появляться в CSSStyleSheet. Свойства, описанные здесь, на самом деле определяются DOM-интерфейсом CSSStyleRule. Правила стилей являются наиболее общими и наиболее важными типами правил в таблицах стилей, а перечисленные здесь свойства – единственные, которые могут использоваться независимо от типа броузера. IE недостаточно полно поддерживает спецификацию DOM Level 2 (по крайней мере, до версии IE 7), но реализует объект CSSRule, поддерживающий два указанных свойства.

См. также CSS2Properties, CSSStyleSheet

CSSStyleSheet

DOM Level 2 CSS, IE 4

таблица стилей CSS **Object→CSSStyleSheet**

Свойства

`readonly CSSRule[] cssRules`

Доступный только для чтения объект, похожий на массив, который хранит объекты `CSSRule`, составляющие таблицу стилей. В IE вместо него используется свойство `rules`. В DOM-совместимых реализациях этот массив включает объекты, представляющие все правила в таблице стилей, включая правила с префиксом @, такие как директива `@import`. Правила этого типа реализуют интерфейсы, отличные от описанных в статье о `CSSRule`. Данные типы объектов правил плохо поддерживаются разными типами броузеров, по этой причине их описания не были включены в книгу. Таким образом, прежде чем использовать свойства элементов массива, необходимо убедиться, что они определяют свойства `CSSRule`.

`boolean disabled`

Значение `true` означает, что таблица стилей неактивна и не будет применяться к документу. Значение `false` – таблица стилей активна и будет применяться к документу.

`readonly String href`

Строка URL-адреса таблицы стилей, которая связана с документом, или `null`, если таблица стилей встроена в документ.

`readonly StyleSheet parentStyleSheet`

Таблица стилей, которая включает в себя данную таблицу, или `null`, если данная таблица включена непосредственно в документ.

`readonly CSSRule[] rules`

Эквивалент массива `cssRule[]` стандарта DOM в IE.

`readonly String title`

Заголовок таблицы стилей, если указан. Заголовок может определяться атрибутом `title` элемента `<style>` или `<link>`, который ссылается на эту таблицу стилей.

`readonly String type`

Тип данной таблицы стилей в виде MIME-типа. Таблицы стилей CSS имеют тип `text/css`.

Методы

`addRule()`

Характерный для IE метод добавления CSS-правила в таблицу стилей.

`deleteRule()`

Метод стандарта DOM, который удаляет правило в указанной позиции.

`insertRule()`

Метод стандарта DOM, который вставляет новое правило в указанную позицию.

`removeRule()`

Характерный для IE метод удаления правила из таблицы стилей.

Описание

Этот интерфейс представляет таблицу стилей CSS. Он обладает свойствами и методами, позволяющими деактивировать таблицу стилей, читать, вставлять и удалять правила. В IE реализован несколько иной прикладной интерфейс, нежели описывается в стандарте DOM. В IE вместо массива `cssRules[]` используется массив `rules[]`, а вместо стандартных методов `insertRule()` и `deleteRule()` — методы `addRule()` и `removeRule()`.

Объекты `CSSStyleSheet`, которые применяются к документу, являются элементами массива `styleSheets[]` объекта `Document`. Кроме того, стандарт DOM требует (хотя реализация этого требования еще не получила широкого распространения), чтобы любой элемент `<style>` или `<link>` либо узел `ProcessingInsrtuction`, определяющий таблицы стилей или ссылающийся на них, обеспечивал доступ к объекту `CSSStyleSheet` через свойство `sheet`.

См. также `CSSRule`, свойство `styleSheets[]` объекта `Document`; глава 16

CSSStyleSheet.addRule() IE 4

характерный для IE метод добавления правила в таблицу стилей

Синтаксис

```
void addRule(String селектор,
             String стиль,
             integer индекс)
```

Аргументы

селектор

CSS-селектор для данного правила.

стиль

Стили, которые должны применяться к элементу, определяемому аргументом *селектор*. Эта строка стиля представляет собой список пар атрибут-значение, разделенных точкой с запятой. В строке отсутствуют открывающая и закрывающая фигурные скобки.

индекс

Позиция в массиве `rules`, куда вставляется это правило. Если этот необязательный аргумент отсутствует, новое правило добавляется в конец массива.

Описание

Данный метод вставляет (или добавляет в конец) в массив `rules` новое CSS-правило в позицию *индекс*. Этот метод представляет собой альтернативу стандартному методу `insertRule()`, реализованную в IE. Обратите внимание: методы `addRule()` и `insertRule()` отличаются набором входных аргументов.

CSSStyleSheet.deleteRule() DOM Level 2 CSS

удаляет правило из таблицы стилей

Синтаксис

```
void deleteRule(unsigned long индекс)
    throws DOMException;
```

Аргументы

индекс

Индекс удаляемого правила в массиве cssRules.

Исключения

Метод генерирует исключение DOMException с кодом INDEX_SIZE_ERR, если *индекс* отрицателен либо больше или равен cssRules.length. Если таблица стилей доступна только для чтения, он генерирует исключение DOMException с кодом NO_MODIFICATION_ALLOWED_ERR.

Описание

Метод удаляет правило в позиции, указанной аргументом *индекс*, из массива cssRules. Это метод стандарта DOM; альтернативный метод CSSStyleSheet.removeRule(), реализованный в IE, описывается в соответствующей статье.

CSSStyleSheet.insertRule() DOM Level 2 CSS

вставляет правило в таблицу стилей

Синтаксис

```
unsigned long insertRule(String правило,
                         unsigned long индекс)
    throws DOMException;
```

Аргументы

правило

Полное, подходящее для синтаксического разбора текстовое представление правила, которое должно быть добавлено в таблицу стилей. Для стилевых правил оно включает как селектор элемента, так и информацию о стиле.

индекс

Позиция в массиве cssRules, куда должно быть вставлено или добавлено правило.

Возвращаемое значение

Значение аргумента *индекс*.

Исключения

Этот метод генерирует исключение DOMException с одним из перечисленных ниже значений code в следующих ситуациях:

HIERARCHY_REQUEST_ERR

Синтаксис CSS не допускает появления указанного правила в данном месте.

INDEX_SIZE_ERR

Значение аргумента *индекс* отрицательно или больше, чем cssRules.length.

NO_MODIFICATION_ALLOWED_ERR

Таблица стилей доступна только для чтения.

SYNTAX_ERR

Указанный текст правила содержит синтаксическую ошибку.

Описание

Метод вставляет (или добавляет) новое CSS-правило в позиции, указанной аргументом *индекс* в массиве cssRules данной таблицы стилей. Это метод стандарта DOM; альтернативный метод CSSStyleSheet.addRule(), реализованный в IE, описывается в соответствующей статье.

CSSStyleSheet.removeRule() IE 4

характерный для IE метод удаления правила из таблицы стилей

Синтаксис

```
void removeRule(integer индекс)
```

Аргументы

индекс

Индекс удаляемого правила в массиве rules[]. Если этот необязательный аргумент опущен, удаляется первое правило в массиве.

Описание

Этот метод удаляет CSS-правило, находящееся в позиции *индекс* в массиве rules[]. Данный метод представляет собой альтернативу стандартному методу deleteRule(), реализованную в IE.

Document DOM Level 1 Core

HTML- или XML-документ **Node→Document**

Подынтерфейсы

HTMLDocument

Свойства

readonly Window defaultView

Объект Window (или «представление» в терминологии DOM) в веб-броузере, в котором отображается документ.

readonly DocumentType doctype

Для XML-документов с объявлением <!DOCTYPE> задает узел DocumentType, представляющий DTD документа. Для HTML- и XML-документов без объявления <!DOCTYPE> это свойство равно null.

readonly Element documentElement

Ссылка на корневой элемент документа. Для HTML-документов это свойство всегда является объектом Element, представляющим тег <html>. Этот корневой элемент также доступен через массив childNodes[], унаследованный от Node. См. также описание свойства body объекта HTMLDocument.

readonly DOMImplementation implementation

Объект DOMImplementation, представляющий реализацию, создавшую этот документ.

readonly CSSStyleSheet[] styleSheets

Коллекция объектов, представляющих все таблицы стилей, встроенные в документ, или на которые есть ссылки из него. В HTML-документах они включают таблицы стилей, определенные с помощью тегов <link> и <style>.

Методы

`addEventListener()`

Добавляет функцию-обработчик события в набор обработчиков событий документа. Это метод стандарта DOM, он поддерживается всеми современными броузерами за исключением IE.

`attachEvent()`

Добавляет функцию-обработчик события в набор обработчиков событий документа. Это метод, реализованный в IE и представляющий альтернативу методу `addEventListener()`.

`createAttribute()`

Создает новый узел `Attr` с указанным именем.

`createAttributeNS()`

Создает новый узел `Attr` с указанным именем и пространством имен.

`createCDATASection()`

Создает новый узел `CDATASection`, содержащий указанный текст.

`createComment()`

Создает новый узел `Comment`, содержащий указанную строку.

`createDocumentFragment()`

Создает новый пустой узел `DocumentFragment`.

`createElement()`

Создает новый узел `Element` с указанным именем тега.

`createElementNS()`

Создает новый узел `Element` с указанными именем и пространством имен.

`createEvent()`

Создает новый искусственный объект `Event` указанного типа.

`createExpression()`

Создает новый объект `XPathExpression`, который представляет скомпилированный XPath-запрос. Альтернатива, реализованная в IE, представлена методом `Node.selectNodes()`.

`createProcessingInstruction()`

Создает новый узел `ProcessingInstruction` с заданными целью и строкой данных.

`createRange()`

Создает новый объект `Range`. Этот метод формально является частью интерфейса `DocumentRange`; он обеспечивается объектом `Document` только в реализациях, поддерживающих модуль `Range`.

`createTextNode()`

Создает новый узел `Text`, представляющий указанный текст.

`detachEvent()`

Удаляет функцию-обработчик события из документа. Этот метод представляет собой альтернативу стандартному методу `removeEventListener()`, реализованную в IE.

`dispatchEvent()`

Посылает искусственное событие этому документу.

`evaluate()`

Исполняет XPath-запрос к данному документу. Альтернатива, реализованная в IE, представлена методом `Node.selectNodes()`.

`getElementById()`

Возвращает элемент-потомок данного документа, имеющий указанное значение своего атрибута `id`, или `null`, если такой элемент в документе отсутствует.

`getElementsByTagName()`

Возвращает массив (формально – `NodeList`) всех узлов `Element` в данном документе, имена которых совпадают с указанным именем тега. Узлы `Element` расположены в результирующем массиве в том же порядке, что и в исходном тексте документа.

`getElementsByTagNameNS()`

Возвращает массив всех узлов элемента, имеющих указанные имя тега и пространство имен.

`importNode()`

Делает копию узла из некоторого другого документа, подходящую для вставки в этот документ.

`loadXML()`

Выполняет разбор строки на языке XML и запоминает результат в заданном элементе документа.

`removeEventListener()`

Удаляет функцию-обработчик события из набора обработчиков событий документа. Этот стандартный метод DOM реализован во всех современных броузерах, за исключением IE.

Описание

Интерфейс `Document` – это корневой узел дерева документа. `Document` может иметь несколько дочерних узлов, но только один из них может быть узлом `Element` – это корневой элемент документа. К корневому элементу проще всего обращаться через свойство `documentElement`. Свойства `doctype` и `implementation` предоставляют доступ к объекту `DocumentType` (если он есть) и к объекту `DOMImplementation` данного документа.

Большинство методов, определенных в интерфейсе `Document`, – это «методы-фабрики», используемые для создания различных типов узлов, которые могут быть вставлены в документ. К исключениям можно отнести методы `getElementById()` и `getElementsByTagName()`, особенно удобные для поиска определенного элемента или набора связанных элементов внутри дерева документа. Еще одно исключение составляют методы регистрации обработчиков событий, такие как `addEventHandler()`. Эти методы, связанные с механизмом обработки событий, также определяются интерфейсом `Element` и подробно описываются в справочной статье `Element`.

Обычно доступ к объекту `Document` можно получить через свойство `document` объекта `Window`. Кроме того, объект `Document` доступен через свойство `contentDocument` объектов `Frame` и `IFrame`, а также через свойство `ownerDocument` любого объекта `Node`, который имеется в документе.

При работе с XML-документами (включая XHTML-документы) создать новый объект `Document` можно с помощью метода `createDocment()` объекта `DOMImplementation`:

```
document.implementation.createDocument(namespaceURL, rootTagName, null);
```

В IE можно воспользоваться следующим способом:

```
new ActiveXObject("MSXML2.DOMDocument");
```

Реализацию кроссплатформенной вспомогательной функции, предназначенной для создания объекта Document, можно найти в примере 21.1. Кроме того, можно загрузить XML-файл из сети и преобразовать его в объект Document. (См. описание свойства responseXML объекта XMLHttpRequest.) Дополнительно существует возможность преобразовать XML-строку в объект Document методом DOMParser.parseFromString() или реализованным в IE методом Document.loadXML(). (В примере 21.4 приводится кросс-платформенная вспомогательная функция, которая использует эти методы для разбора строк на языке XML.)

Описания свойств, характерных для HTML, приводятся в справочной статье об интерфейсе HTMLDocument.

См. также

DOMImplementation, DOMParser, HTMLDocument, Window, XMLHttpRequest; глава 15

Document.addEventListener()

См. описание метода Element.addEventListener()

Document.attachEvent()

См. описание метода Element.attachEvent()

Document.createAttribute() DOM Level 1 Core

создает новый узел Attr

Синтаксис

```
Attr createAttribute(String имя)
    throws DOMException;
```

Аргументы

имя

Имя создаваемого атрибута.

Возвращаемое значение

Вновь созданный узел Attr со свойством nodeName, равным *имени*.

Исключения

Этот метод генерирует исключение DOMException с кодом, равным INVALID_CHARACTER_ERR, если *имя* содержит недопустимые символы.

См. также

Attr, Element.setAttribute(), Element.setAttributeNode()

Document.createAttributeNS() DOM Level 2 Core

создает новый узел Attr с указанными именем и пространством имен

Синтаксис

```
Attr createAttributeNS(String URIпространстваИмен,
                       String уточненноеИмя)
    throws DOMException;
```

Аргументы

URIпространстваИмен

Уникальный идентификатор пространства имен для узла `Attr` или `null` в случае отсутствия пространства имен.

уточненноеИмя

Уточненное имя атрибута, которое должно включать префикс пространства имен, двоеточие и локальное имя.

Возвращаемое значение

Вновь созданный узел `Attr` с указанными именем и пространством имен.

Исключения

Этот метод может генерировать исключение `DOMException` с одним из перечисленных далее кодов в следующих ситуациях:

`INVALID_CHARACTER_ERR`

Аргумент *уточненноеИмя* содержит недопустимый символ.

`NAMESPACE_ERR`

Аргумент *уточненноеИмя* имеет неверный формат или имеется несоответствие между аргументами *уточненноеИмя* и *URIпространстваИмен*.

`NOT_SUPPORTED_ERR`

Реализация не поддерживает XML-документов, следовательно, в ней нет этого метода.

Описание

Метод `createAttributeNS()` идентичен методу `createAttribute()`, за исключением того, что созданный узел `Attr` имеет имя и пространство имен, а не просто имя. Применять этот метод имеет смысл только в XML-документах, использующих пространства имен.

Document.createCDATASection() DOM Level 1 Core

создает новый узел CDATASection

Синтаксис

```
CDATASection createCDATASection(String данные)
    throws DOMException;
```

Аргументы

данные

Текст в создаваемом разделе `CDATA`.

Возвращаемое значение

Вновь созданный узел CDATASection, содержимым которого являются указанные *данные*.

Исключения

Для HTML-документов этот метод генерирует исключение DOMException с кодом, равным NOT_SUPPORTED_ERR, т. к. в HTML-документах наличие узлов CDATASection не допускается.

Document.createComment() DOM Level 1 Core

создает новый узел Comment

Синтаксис

```
Comment createComment(String данные);
```

Аргументы

данные

Текст создаваемого узла Comment.

Возвращаемое значение

Вновь созданный узел Comment, текстом которого являются указанные *данные*.

Document.createDocumentFragment() DOM Level 1 Core

создает новый пустой узел DocumentFragment

Синтаксис

```
DocumentFragment createDocumentFragment();
```

Возвращаемое значение

Вновь созданный узел DocumentFragment, не содержащий дочерних узлов.

Document.createElement() DOM Level 1 Core

создает новый узел Element

Синтаксис

```
Element createElement(String имяТега)
    throws DOMException;
```

Аргументы

имяТега

Имя тега создаваемого элемента. HTML-теги безразличны к регистру, тогда как в именах XML-тегов регистр букв имеет значение.

Возвращаемое значение

Вновь созданный узел Element с указанным именем тега.

Исключения

Этот метод генерирует исключение DOMException с кодом INVALID_CHARACTER_ERR, если *имяТега* содержит недопустимый символ.

Document.createElementNS() DOM Level 2 Core

создает новый узел Element с использованием пространства имен

Синтаксис

```
Element createElementNS(String URIпространстваИмен,
                        String уточненноеИмя)
    throws DOMException;
```

Аргументы

URIпространстваИмен

Уникальный идентификатор пространства имен для нового элемента или `null` в случае отсутствия пространства имен.

уточненноеИмя

Уточненное имя нового узла `Element`. Должно включать префикс пространства имен, двоеточие и локальное имя.

Возвращаемое значение

Вновь созданный узел `Element` с указанными именем тега и пространством имен.

Исключения

Этот метод может генерировать исключение `DOMException` с одним из перечисленных ниже значений `code` в следующих ситуациях:

`INVALID_CHARACTER_ERR`

Аргумент *уточненноеИмя* содержит недопустимый символ.

`NAMESPACE_ERR`

Формат аргумента *уточненноеИмя* неверен или имеется несоответствие между аргументами *уточненноеИмя* и *URIпространстваИмен*.

`NOT_SUPPORTED_ERR`

Реализация не поддерживает XML-документы, следовательно, в ней нет этого метода.

Описание

Метод `createElementNS()` идентичен методу `createElement()`, за исключением того, что у созданного узла `Element` не просто есть имя, он именован и определен в пространстве имен. Применять этот метод имеет смысл только в XML-документах, использующих пространства имен.

Document.createEvent() DOM Level 2 Events

создает объект Event

Синтаксис

```
Event createEvent(String типСобытий)
    throws DOMException;
```

Аргументы

типСобытий

Имя модуля событий, для которого требуется создать объект `Event`. Список допустимых типов событий приводится в подразделе «Описание».

Возвращаемое значение

Вновь созданный объект Event указанного типа.

Исключения

Этот метод генерирует исключение DOMException с кодом NOT_SUPPORTED_ERR, если реализация не поддерживает события запрашиваемого типа.

Описание

Метод создает новое событие, имеющее тип, указанный в аргументе *типСобытий*. Обратите внимание: значение этого аргумента должно быть не именем создаваемого интерфейса событий, а именем DOM-модуля, определяющего этот интерфейс. В следующей таблице приведены допустимые значения аргумента *типСобытий* и интерфейсы событий, создаваемые каждым из этих значений.

Аргумент eventType	Интерфейс событий	Метод инициализации
HTMLEvents	Event	initEvent()
MouseEvents	MouseEvent	initMouseEvent()
UIEvents	UIEvent	initUIEvent()

Создав объект Event с помощью этого метода, необходимо инициализировать объект, вызвав указанный в таблице метод инициализации. Подробности о методе инициализации приводится в справочной статье соответствующего интерфейса событий.

Этот метод фактически определяется не интерфейсом Document, а DOM-интерфейсом DocumentEvent. Если реализация поддерживает модуль Events, то объект Document всегда реализует интерфейс DocumentEvent и поддерживает этот метод.

См. также Event, MouseEvent, UIEvent

Document.createExpression() Firefox 1.5, Safari 2.01, Opera 9

создает выражение XPath для последующего использования

Синтаксис

```
XPathExpression createExpression(String текстВыражения,
                                 Function отображениеURLПространстваИмен)
    throws XPathException
```

Аргументы

текстВыражения

Строка, представляющая компилируемое выражение XPath.

отображениеURLПространстваИмен

Функция, которая выполняет отображение префикса пространства имен на полный URL-адрес пространства имен или null, если выполнять отображение не требуется.

Возвращаемое значение

Объект XPathExpression.

Исключения

Этот метод возбуждает исключение, если *текстВыражения* содержит синтаксическую ошибку или использует пространство имен, которое не распознается функцией *отображениеURLПространстваИмен*.

Описание

Данный метод принимает строку, представляющую XPath-выражение, и преобразует ее в скомпилированное представление `XPathExpression`. Помимо выражения метод принимает функцию в виде `function(prefix)`, которая выполняет преобразование строки префикса пространства имен в строку с полным URL-адресом этого пространства имен.

Этот интерфейс не поддерживается в Internet Explorer. Вариант, реализованный в IE, вы найдете в справочной статье о методе `Node.selectNodes()`.

См. также `Document.evaluate()`, `Node.selectNodes()`, `XPathExpression`, `XPathResult`

Document.createProcessingInstruction() DOM Level 1 Core

создает узел ProcessingInstruction

Синтаксис

```
ProcessingInstruction createProcessingInstruction(String цель,
                                                   String данные)
    throws DOMException;
```

Аргументы

цель Цель исполняемой инструкции.

данные Текст исполняемой инструкции.

Возвращаемое значение

Вновь созданный узел `ProcessingInstruction`.

Исключения

Этот метод может генерировать исключение `DOMException` с одним из перечисленных ниже значений `code` в следующих ситуациях:

`INVALID_CHARACTER_ERR`

Указанная цель содержит недопустимый символ.

`NOT_SUPPORTED_ERR`

Это HTML-документ, поэтому исполняемые инструкции не поддерживаются.

Document.createRange() DOM Level 2 Range

создает объект Range

Синтаксис

```
Range createRange();
```

Возвращаемое значение

Вновь созданный объект `Range`, в котором обе граничные точки установлены на начало документа.

Описание

Этот метод создает объект Range, который может использоваться для представления области данного документа или области объекта DocumentFragment, связанного с этим документом.

Обратите внимание: этот метод фактически определен не в интерфейсе Document, а в интерфейсе DocumentRange. Если реализация поддерживает модуль Range, то объект Document всегда реализует DocumentRange и определяет этот метод.

Document.createTextNode() DOM Level 1 Core

создает новый узел Text

Синтаксис

```
Text createTextNode(String данные);
```

Аргументы

данные

Содержимое узла Text.

Возвращаемое значение

Вновь созданный узел Text, представляющий указанную строку данных.

Document.detachEvent()

См. описание метода Element.detachEvent()

Document.dispatchEvent()

См. описание метода Element.dispatchEvent()

Document.evaluate() Firefox 1.5, Safari 2.01, Opera 9

вычисляет выражение XPath

Синтаксис

```
XPathResult evaluate(String текстВыражения,
                     Node контекстныйУзел,
                     Function отображениеURLПространстваИмен,
                     short типРезультата,
                     XPathResult результат)
    throws DOMException, XPathException
```

Аргументы

текстВыражения

Строка, представляющая вычисляемое выражение XPath.

контекстныйУзел

Узел документа, для которого вычисляется выражение.

отображениеURLПространстваИмен

Функция, которая выполняет отображение префикса пространства имен на полный URL-адрес пространства имен или `null`, если выполнять отображение не требуется.

типРезультата

Определяет ожидаемый тип объекта, к которому будет приведен результат. Возможные значения аргумента *типРезультата* – это константы, определяемые объектом `XPathResult`.

результат

Существующий объект `XPathResult` или `null`, если требуется создать новый объект.

Возвращаемое значение

Объект `XPathResult`, представляющий результат вычисления выражения для заданного контекстного узла.

Исключения

Этот метод может возбуждать исключение, если текст выражения содержит синтаксическую ошибку, если результат выражения не может быть приведен к требуемому типу результата, если выражение содержит пространства имен, которые не распознаются функцией *отображениеURLПространстваИмен*, и если контекстный узел имеет неверный тип или не связан с данным документом.

Описание

Данный метод вычисляет указанное XPath-выражение для заданного контекстного узла и возвращает объект `XPathResult`, выполняя преобразование объекта к требуемому типу результата. Если выражение предполагается вычислять несколько раз, предпочтительнее сначала скомпилировать выражение вызовом метода `Document.createExpression()`, а затем вызывать метод `evaluate()` полученного объекта `XPathExpression`.

Этот интерфейс не поддерживается в Internet Explorer. Вариант, реализованный в IE, вы найдете в справочных статьях о методах `Node.selectNodes()` и `Node.selectSingleNode()`.

См. также

`Document.createExpression()`, `Node.selectNodes()`, `Node.selectSingleNode()`, `XPathExpression`, `XPathResult`

Document.getElementById() DOM Level 2 Core

находит элемент с заданным уникальным идентификатором (ID)

Синтаксис

```
Element getElementById(String IdЭлемента);
```

Аргументы

IdЭлемента

Значение атрибута `id` требуемого элемента.

Возвращаемое значение

Узел `Element`, представляющий элемент документа с указанным атрибутом `id`, или `null`, если такой элемент не найден.

Описание

Метод ищет в документе узел Element с атрибутом id, значение которого равно *IdЭлемента*, и возвращает этот элемент. Если такой элемент не найден, он возвращает null. Значение атрибута id предполагается уникальным в пределах документа, а если этот метод находит более одного элемента с указанным значением *IdЭлемента*, то он может выбрать любой из них или вернуть null. Этот метод важен и часто используется, т. к. обеспечивает простой способ получения объекта Element, представляющего определенный элемент в документе. Обратите внимание: имя этого метода оканчивается суффиксом «Id», а не на «ID», будьте аккуратны в его написании.

В HTML-документах этот метод ищет атрибут с именем id. Для поиска HTML-элементов по значению атрибута name следует использовать метод HTMLDocument.getElementsByName().

В XML-документах он ищет любой атрибут, имеющий тип id, независимо от того, какое имя у этого атрибута. Если типы XML-атрибутов неизвестны (например, если XML-анализатор не смог найти DTD документа), этот метод всегда возвращает null. В клиентском языке JavaScript этот метод для работы с XML-документами обычно не используется. Фактически метод getElementById() изначально был определен как член интерфейса HTMLDocument, но в DOM Level 2 был перенесен в интерфейс Document.

См. также

Document.getElementsByTagName(), Element.getElementsByTagName(), HTMLDocument.getElementsByName()

Document.getElementsByTagName() DOM Level 1 Core

возвращает все узлы Element с заданным именем тега

Синтаксис

```
Node[] getElementsByTagName(String имяТега);
```

Аргументы

имяТега

Имя тега для узлов Element, которые нужно получить, или строка маски «*» для получения всех узлов Element в документе независимо от имени тега. Для HTML-документов имена тегов сравниваются без учета регистра (до версии 6 броузер IE не поддерживал синтаксис с маской «*»).

Возвращаемое значение

Доступный только для чтения массив (формально — NodeList) всех узлов Element в дереве документа с указанным именем тега. Узлы Element возвращаются в том же порядке, в котором они расположены в исходном тексте документа.

Описание

Этот метод возвращает объект NodeList (его можно рассматривать как массив, доступный только для чтения), содержащий все узлы Element из документа, имеющие указанное имя тега, в том порядке, в котором они представлены и в исходном тексте документа. Объект NodeList «живой», т. е. его содержимое по необходимости автоматически обновляется, если элементы с указанным именем тега добавляются или удаляются из документа.

HTML-документы нечувствительны к регистру, поэтому можно задать *имяТега* в любом регистре. В то же время XML-документы чувствительны к регистру, и *имяТега* должно соответствовать только тегам, имеющим в исходном тексте документа то же имя и в точности тот же регистр.

Обратите внимание: интерфейс `Element` определяет метод с тем же именем, который выполняет поиск по поддереву документа. Кроме того, интерфейс `HTMLDocument` определяет метод `getElementsByName()`, выполняющий поиск элементов по значению их атрибутов `name`, а не по именам тегов.

Пример

Поиск и перебор в цикле всех тегов `<h1>` в документе осуществляется так:

```
var headings = document.getElementsByTagName("h1");
for(var i = 0; i < headings.length; i++) { // Цикл по всем полученным тегам
    var h = headings[i];
    // Теперь делаем что-нибудь с элементом <h1> в переменной h
}
```

См. также

`Document.getElementById()`, `Element.getElementsByTagName()`, `HTMLDocument.getElementsByName()`

Document.getElementsByTagNameNS() DOM Level 2 Core

возвращает все узлы Element с заданными именем тега и пространством имен

Синтаксис

```
Node[] getElementsByTagNameNS(String URIпространстваИмен,
                              String локальноеИмя);
```

Аргументы

URIпространстваИмен

Уникальный идентификатор пространства имен для требуемых элементов или «*» для соответствия любому пространству имен.

локальноеИмя

Локальное имя требуемых элементов или «*» для соответствия любому локальному имени.

Возвращаемое значение

Доступный только для чтения массив (формально – `NodeList`) всех узлов `Element` в дереве документа, имеющих указанные пространство имен и локальное имя.

Описание

Этот метод работает точно так же, как `getElementsByTagName()`, за исключением того, что ищет элементы по пространству имен и имени, а не только по имени. Его применение имеет смысл только в XML-документах, использующих пространства имен.

Document.importNode() DOM Level 2 Core

копирует узел из одного документа для использования в другом документе

Синтаксис

```
Node importNode(Node импортируемыйУзел,
                boolean глубина)
    throws DOMException;
```

Аргументы

импортируемыйУзел

Узел, который должен импортироваться.

глубина

Если `true`, рекурсивно копирует также всех потомков импортируемого узла.

Возвращаемое значение

Копия импортируемого узла (и, возможно, всех его потомков), в котором свойство `ownerDocument` установлено равным данному документу.

Исключения

Этот метод генерирует исключение `DOMException` с кодом `NOT_SUPPORTED_ERR`, если аргумент *импортируемыйУзел* является узлом `Document` или `DocumentType`, т. к. эти типы узлов не могут импортироваться.

Описание

Этот метод получает узел, определенный в другом документе, и возвращает копию узла, подходящую для вставки в данный документ. Если аргумент *глубина* равен `true`, копируются также все потомки узла. Исходный узел и его потомки никак не модифицируются. В полученной копии свойство `ownerDocument` устанавливается равным данному документу, а `parentNode` — `null`, поскольку копия пока не вставлена в документ. Функции-подписчики событий, зарегистрированные в исходном узле или дереве, не копируются.

Когда импортируется узел `Element`, с ним импортируются только атрибуты, явно заданные в исходном документе. Когда импортируется узел `Attr`, его свойство `specified` автоматически устанавливается равным `true`.

См. также

`Node.cloneNode()`

Document.loadXML() Internet Explorer

заполняет данный документ содержимым строки на языке XML

Синтаксис

```
void loadXML(String текст)
```

Аргументы

текст Текст на языке XML, подвергаемый разбору.

Описание

Этот характерный для IE метод выполняет синтаксический разбор строки, содержащей текст на языке XML, и строит DOM-дерево узлов в текущем объекте `Document`. Любые узлы, существовавшие в документе ранее, уничтожаются.

Метод отсутствует в объектах Document, представляющих HTML-документы. Обычно перед вызовом loadXML() создается новый пустой объект Document, призванный хранить результат преобразования XML-текста:

```
var doc = new ActiveXObject("MSXML2.DOMDocument");
doc.loadXML(markup);
```

Альтернативный вариант для броузеров, отличающихся от IE, вы найдете в справочной статье о методе DOMParser.parseFromString().

См. также DOMParser.parseFromString()

Document.removeEventListener()

См. описание метода Element.removeEventListener()

DocumentFragment

DOM Level 1 Core

смежные узлы и их поддеревья **Node→DocumentFragment**

Описание

Интерфейс DocumentFragment представляет часть (фрагмент) документа. Если говорить конкретно, то он представляет список смежных узлов документа и всех их потомков, но без общего родительского узла. Узлы DocumentFragment никогда не являются частью дерева документа, а унаследованное свойство parentNode в них всегда равно null. Однако особенность узлов DocumentFragment делает их очень полезными: когда поступает запрос на вставку DocumentFragment в дерево документа, вставляется не сам узел DocumentFragment, а все его дочерние узлы. Благодаря этому интерфейс DocumentFragment хорош в качестве временного хранилища для узлов, которые требуется вставить в документ все сразу. Интерфейс DocumentFragment также очень полезен для реализации операций вырезания, копирования и вставки, особенно в сочетании с интерфейсом Range.

Создать новый пустой узел DocumentFragment можно с помощью метода Document.createDocumentFragment(), а получить узел DocumentFragment, содержащий фрагмент существующего документа, – с помощью метода Range.extractContents() или Range.cloneContents().

См. также Range

DocumentType

DOM Level 1 XML

DTD XML-документа **Node→DocumentType**

Свойства

readonly String internalSubset *[DOM Level 2]*

Внутреннее подмножество DTD (т. е. часть DTD, присутствующая в самом документе, а не во внешнем файле). Ограничивающие квадратные скобки внутреннего подмножества не являются частью возвращаемого значения. Если внутреннее подмножество не существует, это свойство равно null.

readonly String name

Имя типа документа. Это идентификатор, расположенный непосредственно за объявлением <!DOCTYPE> в начале XML-документа, и он совпадает с именем тега корневого элемента документа.

`readonly String publicId` *[DOM Level 2]*

Открытый идентификатор внешнего подмножества DTD или `null`, если идентификатор не указан.

`readonly String systemId` *[DOM Level 2]*

Системный идентификатор внешнего подмножества DTD или `null`, если идентификатор не указан.

Описание

Этот редко используемый интерфейс представляет DTD XML-документа. Программистам, работающим исключительно с HTML-документами, этот интерфейс никогда не потребуется.

Так как DTD не является частью содержимого документа, узлы `DocumentType` никогда не присутствуют в дереве документа. Если в XML-документе имеется DTD, узел `DocumentType` для этого определения типа докумета доступен через свойство `doctype` узла `Document`.

Хотя стандарт W3C DOM включает API для обращения к сущностям и нотациям языка XML, определенным в DTD, этот прикладной интерфейс здесь не описывается. Обычно веб-броузеры не выполняют анализ DTD загружаемого документа и клиентский JavaScript-код не имеет доступа к этим сущностям и нотациям. С точки зрения программирования на клиентском JavaScript этот интерфейс представляет только содержимое тега `<!DOCTYPE>`, но не содержимое DTD-файла, на который ссылается этот тег.

Узлы `DocumentType` неизменяемы и никак не могут модифицироваться.

См. также

`Document`, `DOMImplementation.createDocument()`, `DOMImplementation.createDocumentType()`

DOMException

DOM Level 1 Core

сигнализирует об исключениях или ошибках в базовых объектах DOM

Object→DOMException

Константы

Следующие константы определяют допустимые значения для свойства `code` объекта `DOMException`. Обратите внимание: эти константы являются статическими свойствами `DOMException`, а не свойствами отдельных объектов исключений.

`unsigned short INDEX_SIZE_ERR = 1`

Ошибка границ диапазона массива или индекса в строке.

`unsigned short DOMSTRING_SIZE_ERR = 2`

Возвращаемый текст слишком велик для строки в данной реализации JavaScript.

`unsigned short HIERARCHY_REQUEST_ERR = 3`

Была предпринята попытка поместить узел в недопустимое место в иерархическом дереве документа.

`unsigned short WRONG_DOCUMENT_ERR = 4`

Попытка использовать узел в документе, отличном от документа, создавшего узел.

`unsigned short INVALID_CHARACTER_ERR = 5`

Используется недопустимый символ (например, в имени элемента).

`unsigned short NO_DATA_ALLOWED_ERR = 6`

Сейчас не используется.

`unsigned short NO_MODIFICATION_ALLOWED_ERR = 7`

Была предпринята попытка модифицировать узел, доступный только для чтения и не допускающий модификации.

`unsigned short NOT_FOUND_ERR = 8`

Узел не найден там, где ожидалось.

`unsigned short NOT_SUPPORTED_ERR = 9`

Метод или свойство не поддерживается в текущей реализации DOM.

`unsigned short INUSE_ATTRIBUTE_ERR = 10`

Была предпринята попытка связать узел `Attr` с узлом `Element`, когда узел `Attr` уже связан с другим узлом `Element`.

`unsigned short INVALID_STATE_ERR = 11` ***[DOM Level 2]***

Попытка использовать объект, который еще (или уже) не находится в состоянии, допускающем подобное использование.

`unsigned short SYNTAX_ERR = 12` ***[DOM Level 2]***

Указанная строка содержит синтаксическую ошибку. Обычно используется со спецификациями CSS-свойств.

`unsigned short INVALID_MODIFICATION_ERR = 13` ***[DOM Level 2]***

Попытка модифицировать тип объекта `CSSRule` или `CSSValue`.

`unsigned short NAMESPACE_ERR = 14` ***[DOM Level 2]***

Ошибка, связанная с пространствами имен элемента или атрибута.

`unsigned short INVALID_ACCESS_ERR = 15` ***[DOM Level 2]***

Попытка доступа к объекту способом, который не поддерживается в данной реализации.

Свойства

`unsigned short code`

Код ошибки, предоставляющий некоторые сведения о причине исключения. Допустимые значения (и их смысл) для этого свойства определяются только что перечисленными константами.

Описание

Объект `DOMException` создается, когда DOM-метод или DOM-свойство используется некорректно или в неверном контексте. Значение свойства `code` обозначает общий тип возникшего исключения. Обратите внимание, что исключение `DOMException` может генерироваться при чтении или установке свойства объекта, а также при вызове метода объекта.

Описание свойств и методов объекта в этом справочнике включает список типов исключений, которые они могут генерировать. Однако обратите внимание, что некоторые часто возникающие исключения в этих списках отсутствуют. Исключение `DOMException` с кодом `NO_MODIFICATION_ALLOWED_ERR` генерируется при каждой попытке моди-

фицировать узел, доступный только для чтения. Следовательно, большинство методов и доступных для чтения и записи свойств интерфейса Node (и его подынтерфейсов) могут генерировать это исключение. Так как узлы, доступные только для чтения, присутствуют только в XML-документах, а не в HTML-документах, и т. к. этот тип DOMException столь универсален для методов и доступных только для чтения свойств объектов Node, исключение NO_MODIFICATION_ALLOWED_ERR не вошло в описания этих методов и свойств.

Аналогично многие DOM-методы и DOM-свойства, возвращающие строки, могут генерировать исключение DOMException с кодом DOMSTRING_SIZE_ERR, обозначающим, что возвращаемый текст слишком велик для представления в виде строкового значения в текущей реализации JavaScript. Несмотря на то, что этот тип исключения теоретически может генерироваться многими свойствами и методами, на практике он встречается очень редко и опускается в описаниях этих методов и свойств.

Обратите внимание: исключение DOMException сигнализирует не обо всех исключениях в DOM. Исключения, затрагивающие DOM-модуль Range, приводят к исключению RangeException.

См. также

RangeException

DOMImplementation

DOM Level 1 Core

методы, не зависящие от конкретного документа **Object→DOMImplementation**

Методы

createDocument()

Создает новый объект Document с корневым элементом (свойство documentElement возвращаемого объекта Document) указанного типа.

createDocumentType()

Создает новый узел DocumentType.

hasFeature()

Проверяет, поддерживает ли текущая реализация указанную версию указанного модуля.

Описание

Интерфейс DOMImplementation представляет собой структуру, предназначенную для размещения методов, не относящихся к какому-либо конкретному объекту Document, а являющихся «глобальными» для реализации DOM. Ссылку на объект DOMImplementation можно получить через свойство implementation любого объекта Document.

DOMImplementation.createDocument()

DOM Level 2 Core

создает новый документ и указанный корневой элемент

Синтаксис

```
Document createDocument(String URIпространстваИмен,
                        String уточненноеИмя,
                        DocumentType типДокумента)
    throws DOMException;
```

Аргументы

URIпространстваИмен

Уникальный идентификатор пространства имен создаваемого корневого элемента или null в случае отсутствия пространства имен.

уточненноеИмя

Имя создаваемого корневого элемента для этого документа. Если значение *URIпространстваИмен* не равно null, это имя должно включать префикс – имя пространства имен и двоеточие.

типДокумента

Объект DocumentType для только что созданного документа или null, если он не требуется.

Возвращаемое значение

Объект Document со свойством documentElement, установленным равным корневому узлу Element указанного типа.

Исключения

Этот метод может генерировать исключение DOMException с перечисленными ниже значениями code в следующих ситуациях:

INVALID_CHARACTER_ERR

Аргумент *уточненноеИмя* содержит недопустимый символ.

NAMESPACE_ERR

Формат аргумента *уточненноеИмя* неверен или есть несоответствие между аргументами *уточненноеИмя* и *URIпространстваИмен*.

NOT_SUPPORTED_ERR

Текущая реализация не поддерживает XML-документы и не реализует этот метод.

WRONG_DOCUMENT_ERR

Указанный тип документа уже используется для другого документа или был создан с помощью другого объекта DOMImplementation.

Описание

Этот метод создает новый объект XML-документа и указанный корневой объект documentElement для этого документа. Если аргумент *типДокумента* не равен null, свойство ownerDocument этого объекта DocumentType устанавливается равным только что созданному документу.

Этот метод используется для создания XML-документов и может не поддерживаться реализациями, ориентированными только на HTML.

См. также DOMImplementation.createDocumentType()

DOMImplementation.createDocumentType() DOM Level 2 Core

создает новый узел DocumentType

Синтаксис

```
DocumentType createDocumentType(String уточненноеИмя,
                                String publicId,
                                String systemId)
    throws DOMException;
```

Аргументы

уточненноеИмя

Имя типа документа. Если используются пространства имен XML, это может быть уточненное имя, содержащее префикс пространства имен, и локальное имя, разделенные двоеточием.

publicId

Открытый идентификатор типа документа или `null`.

systemId

Системный идентификатор типа документа или `null`. Этот аргумент обычно задает локальное имя DTD-файла.

Возвращаемое значение

Новый объект `DocumentType` со свойством `ownerDocument`, равным `null`.

Исключения

Этот метод может генерировать исключение `DOMException` с перечисленными далее значениями `code` в следующих ситуациях:

`INVALID_CHARACTER_ERR`

Аргумент *уточненноеИмя* содержит недопустимый символ.

`NAMESPACE_ERR`

Формат аргумента *уточненноеИмя* неверен.

`NOT_SUPPORTED_ERR`

Текущая реализация не поддерживает XML-документы и не реализует этот метод.

Описание

Этот метод создает новый узел `DocumentType` и задает только внешнее подмножество типа документа. Стандарт DOM до Level 2 не предлагает какого-либо способа задания внутреннего подмножества, а в возвращаемом узле `DocumentType` не определено никаких узлов `Entity` или `Notation`. Этот метод полезен только для XML-документов.

DOMImplementation.hasFeature() DOM Level 1 Core

определяет, поддерживает ли реализация определенный модуль

Синтаксис

```
boolean hasFeature(String модуль,
                   String версия);
```

Аргументы

модуль

Имя модуля (feature), поддержка которого проверяется. Допустимые имена модулей стандарта DOM Level 2 перечислены в приведенной ниже таблице. Имена модулей чувствительны к регистру.

версия

Номер версии проверяемого модуля либо `null` или пустая строка `""`, если достаточно поддержки любой версии модуля. В спецификации DOM Level 2 поддерживаются номера версий 1.0 и 2.0.

Возвращаемое значение

Значение `true`, если реализация полностью поддерживает указанную версию указанного модуля, или `false` – в противном случае. При отсутствии номера версии метод возвращает `true`, если реализация полностью поддерживает любую версию указанного модуля.

Описание

Стандарт W3C DOM является модульным, и от реализаций не требуется реализовывать все модули или все функциональные возможности, предлагаемые стандартом. Этот метод проверяет, поддерживает ли реализация DOM указанный модуль. Информация о доступности в каждой статье этого справочника по DOM включает имя модуля. Обратите внимание, что хотя Internet Explorer 5 и 5.5 частично поддерживают спецификацию DOM Level 1, этот важный метод до IE 6 не поддерживался.

Полный набор имен модулей, которые могут выступать в качестве аргумента *модуль*, представлен в следующей таблице.

Модуль	Описание
`Core`	Поддерживаются интерфейсы `Node`, `Element`, `Document`, `Text` и другие фундаментальные интерфейсы, обязательные для всех реализаций DOM. Все совместимые реализации должны поддерживать этот модуль
`HTML`	Реализованы интерфейсы `HTMLElement`, `HTMLDocument` и другие интерфейсы, специфические для HTML
`XML`	Реализованы узлы `Entity`, `EntityReference`, `ProcessingInstruction`, `Notation` и другие типы узлов, полезные только для XML-документов
`StyleSheets`	Реализованы простые интерфейсы, описывающие обобщенные таблицы стилей
`CSS`	Реализованы интерфейсы, специфические для таблиц стилей CSS.
`CSS2`	Реализован интерфейс `CSS2Properties`
`Events`	Реализованы интерфейсы базовой обработки событий
`UIEvents`	Реализованы интерфейсы для событий пользовательского интерфейса
`MouseEvents`	Реализованы интерфейсы для событий мыши
`HTMLEvents`	Реализованы интерфейсы для HTML-событий
`MutationEvents`	Реализованы интерфейсы для событий изменения документа
`Range`	Реализованы интерфейсы для манипулирования областями документа
`Traversal`	Реализованы интерфейсы для расширенного обхода документа
`Views`	Реализованы интерфейсы для представлений документа

Пример

Этот метод может использоваться следующим образом:

```
// Проверяет, поддерживает ли броузер DOM Level 2 Range API
if (document.implementation &&
    document.implementation.hasFeature &&
    document.implementation.hasFeature("Range", "2.0")) {
    // Если да, используем его здесь...
}
```

```
else {
    // Если нет, выполняем обработку, при которой не требуется объект Range
}
```

См. также Node.isSupported()

DOMParser

Firefox 1.0, Safari 2.01, Opera 7.60

выполняет разбор XML-текста для создания объекта Document **Object→DOMParser**

Конструктор

```
new DOMParser()
```

Методы

parseFromString()

Разбирает строку с XML-текстом и возвращает объект Document.

Описание

Объект DOMParser выполняет синтаксический разбор XML-текста и возвращает объект Document XML-документа. Чтобы воспользоваться объектом DOMParser, сначала нужно создать экземпляр объекта с помощью конструктора, а затем вызвать метод parseFromString():

```
var doc = (new DOMParser( )).parseFromString(текст);
```

Internet Explorer не поддерживает объект DOMParser. Вместо этого для синтаксического разбора XML-документов он поддерживает метод Document.loadXML(). Обратите внимание: парсинг XML-документов можно также выполнять с помощью объекта XMLHttpRequest. См. описание свойства responseXML объекта XMLHttpRequest.

См. также Document.loadXML(), XMLHttpRequest; глава 21

DOMParser.parseFromString()

выполняет разбор XML-текста

Синтаксис

```
Document parseFromString(String текст,
                         String типСодержимого)
```

Аргументы

текст

Строка XML-текста.

типСодержимого

Тип содержимого строки *текст*. Это может быть строка "text/xml", "application/xml" или "application/xhtml+xml". Обратите внимание: тип text/html не поддерживается.

Возвращаемое значение

Объект Document, который хранит разобранное представление строки *текст*. Описание альтернативы, реализованной в IE, вы найдете в статье о методе Document.loadXML().

Element

DOM Level 1 Core

HTML- или XML-элемент

Object→Element

Подынтерфейсы

`HTMLElement`

Свойства

`readonly String tagName`

Имя тега элемента. Например, для HTML-элемента `<p>` это строка `"P"`. Для HTML-элементов имя тега возвращается в верхнем регистре независимо от его регистра в исходном тексте документа. XML-документы чувствительны к регистру, и имя тега возвращается в точности в том виде, в каком оно написано в исходном тексте документа. Это свойство имеет то же значение, что и свойство `nodeName` интерфейса `Node`.

Методы

`addEventListener()`

Добавляет функцию-обработчик события в набор обработчиков событий документа. Это DOM метод, который поддерживается всеми современными броузерами за исключением IE.

`attachEvent()`

Добавляет функцию-обработчик событий в набор обработчиков событий документа. Это метод, реализованный в IE и представляющий альтернативу методу `addEventListener()`.

`detachEvent()`

Удаляет функцию-обработчик событий из документа. Этот метод представляет собой альтернативу стандартному методу `removeEventListener()`, реализованную в IE.

`dispatchEvent()`

Посылает искусственное событие этому узлу.

`getAttribute()`

Возвращает значение атрибута с указанным именем в виде строки.

`getAttributeNS()`

Возвращает строковое значение атрибута, заданного локальным именем и URI пространства имен. Применение метода имеет смысл только для XML-документов, использующих пространства имен.

`getAttributeNode()`

Возвращает значение атрибута с указанным именем в виде узла `Attr`.

`getAttributeNodeNS()`

Возвращает значение `Attr` для атрибута, заданного локальным именем и URI пространства имен. Применение метода имеет смысл только для XML-документов, использующих пространства имен.

`getElementsByTagName()`

Возвращает массив (формально `NodeList`) всех узлов `Element`, являющихся потомками данного элемента и имеющих указанное имя тега, в том порядке, в котором они расположены в документе.

`getElementsByTagNameNS()`

То же самое, что и `getElementsByTagName()`, кроме того, что имя тега элемента задается локальным именем и URI пространства имен. Использование метода имеет смысл только для XML-документов, использующих пространства имен.

`hasAttribute()`

Возвращает `true`, если этот элемент имеет атрибут с указанным именем, или `false` в противном случае. Обратите внимание: этот метод возвращает `true`, если указанный атрибут явно задан в исходном тексте документа или если DTD документа задает для этого атрибута значение по умолчанию.

`hasAttributeNS()`

То же самое, что и `hasAttribute()`, за исключением того, что атрибут задается комбинацией локального имени и URI пространства имен. Использование метода имеет смысл только для XML-документов, использующих пространства имен.

`removeAttribute()`

Удаляет из элемента атрибут с указанным именем. Однако следует заметить, что этот метод удаляет только атрибуты, явно заданные для этого элемента в исходном тексте документа. Если в DTD задается значение, предлагаемое для этого атрибута по умолчанию, это значение становится новым значением атрибута.

`removeAttributeNode()`

Удаляет указанный узел `Attr` из списка атрибутов этого элемента. Обратите внимание, что этот метод работает только в отношении атрибутов, явно указанных для этого элемента в исходном тексте документа. Если DTD задает значение, предлагаемое для удаляемого атрибута по умолчанию, создается новый узел `Attr` для значения атрибута, предлагаемого по умолчанию.

`removeAttributeNS()`

То же самое, что и `removeAttribute()`, за исключением того, что удаляемый атрибут задается комбинацией локального имени и URI пространства имен. Применение метода имеет смысл только для XML-документов, использующих пространства имен.

`removeEventListener()`

Удаляет функцию-обработчик событий из набора обработчиков событий документа. Этот стандартный DOM-метод реализован во всех современных броузерах, за исключением IE.

`setAttribute()`

Устанавливает атрибут с указанным именем, равным указанному значению. Если атрибут с таким именем еще не существует, в элемент добавляется новый атрибут.

`setAttributeNode()`

Добавляет указанный узел `Attr` в список атрибутов этого элемента. Если атрибут с тем же именем уже существует, его значение заменяется.

`setAttributeNodeNS()`

То же самое, что и `setAttributeNode()`, но этот метод подходит для узлов, возвращаемых методом `Document.createAttributeNS()`. Применение метода имеет смысл только для XML-документов, использующих пространства имен.

setAttributeNS()

То же самое, что и setAttribute(), за исключением того, что устанавливаемый атрибут задается комбинацией локального имени и URI пространства имен. Применение метода имеет смысл только для XML-документов, использующих пространства имен.

Описание

Интерфейс Element представляет HTML- или XML-элементы (теги). Свойство tagName задает имя элемента. Свойство documentElement объекта Document содержит ссылку на корневой элемент этого документа. Свойство body объекта HTMLDocument похоже на него – оно содержит ссылку на элемент <body> документа. Чтобы отыскать элемент HTML-документа с определенным именем, можно воспользоваться методом Document.getElementById() (при условии, что искомый элемент имеет уникальное значение атрибута id). Чтобы отыскать элемент по имени тега, можно использовать метод getElementsByTagName(), который присутствует и в объекте Element, и в объекте Document. В HTML-документах можно воспользоваться похожим методом HTMLDocument.getElementsByName() для поиска элементов по значению атрибута name. Наконец, чтобы вставить в документ новый объект Element, можно задействовать метод Document.createElement().

Метод addEventListener() (и его альтернатива attachEvent() в IE) предоставляет возможность регистрировать функции-обработчики событий заданных типов в заданных элементах. Более подробная информация приводится в главе 17. Формально методы addEventListener(), removeEventListener() и dispatchEvent() определены в интерфейсе EventTarget в спецификации DOM Level 2 Events. Но поскольку все объекты Element реализуют интерфейс EventTarget, описание этих методов приводится здесь.

Другие разнообразные методы этого интерфейса предоставляют доступ к атрибутам элементов. В HTML-документах (и во многих XML-документах) все атрибуты имеют простые строковые значения, и для любых манипуляций с атрибутами можно использовать простые методы getAttribute() и setAttribute().

При работе с XML-документами, которые могут содержать ссылки на сущности как часть значений атрибутов, вам придется иметь дело с объектами Attr и их поддеревом узлов. Можно получать и устанавливать объект Attr для атрибута с помощью методов getAttributeNode() и setAttributeNode() или выполнять цикл по узлам Attr в массиве attributes[] интерфейса Node. Для работы с XML-документом, использующим пространства имен XML, предназначены методы, имена которых имеют окончание «NS».

В спецификации DOM Level 1 метод normalize() был частью интерфейса Element. В спецификации Level 2 метод normalize() стал частью интерфейса Node. Все узлы Element наследуют этот метод и по-прежнему могут его использовать.

См. также HTMLElement, Node; главы 15 и 17

Element.addEventListener() DOM Level 2 Events

регистрирует обработчик события

Синтаксис

```
void addEventListener(String тип,
                      Function обработчик,
                      boolean фазаЗахвата);
```

Аргументы

тип

Тип события, для обработки которого будет вызываться функция-обработчик. Например, `"load"`, `"click"` или `"mousedown"`.

обработчик

Функция-обработчик события, которая вызывается в момент попадания элементу события заданного типа. При вызове функции-обработчику передается объект `Event`, а кроме того, она вызывается как метод элемента, в котором эта функция зарегистрирована.

фазаЗахвата

Если `true`, заданный *обработчик* будет вызываться только во время фазы захвата в процессе распространения события. Чаще используется значение `false`, означающее, что *обработчик* будет вызываться не в фазе захвата, а только если данный узел является целевым элементом для события или событие «всплывает» к данном узлу от целевого элемента.

Описание

Этот метод добавляет указанную функцию-обработчик события в набор зарегистрированных обработчиков для данного узла и данного *типа* события. Если аргумент *фазаЗахвата* имеет значение `true`, функция регистрируется как перехватывающий обработчик события. Если аргумент *фазаЗахвата* имеет значение `false`, функция регистрируется как обычный обработчик события.

Метод `addEventListener()` может вызываться многократно для регистрации нескольких обработчиков одного и того же типа события для одного и того же узла. Однако следует отметить, что стандарт DOM никак не определяет порядок вызова этих обработчиков.

Если произойдет повторная попытка регистрации одной и той же функции-обработчика для того же самого узла и того же *типа* события с тем же значением аргумента *фазаЗахвата*, эта повторная попытка будет просто проигнорирована. Если регистрация нового обработчика произойдет в процессе обработки события, то этот новый обработчик для данного события вызываться не будет.

Когда создается копия узла с помощью метода `Node.cloneNode()` или `Document.importNode()`, обработчики событий, зарегистрированные в исходном элементе, не копируются.

Этот метод также определен и работает аналогичным образом в объектах `Document` и `Window`.

См. также

`Event`; глава 17

Element.attachEvent() IE 4

регистрирует обработчик события

Синтаксис

```
void attachEvent(String тип,
                 Function обработчик);
```

Аргументы

тип

Тип события, для обработки которого будет вызываться указанный *обработчик*. Тип события должен начинаться с префикса «on», например: `"onload"`, `"onclick"` или `"onmousedown"`.

обработчик

Функция, которая будет вызываться для обработки событий указанного типа при передаче их данному элементу. Функции-обработчику не передается никаких аргументов, однако она может обратиться к объекту `Event` через свойство `event` объекта `Window`.

Описание

Этот метод используется для регистрации обработчиков событий в IE. Он предназначен для тех же целей, что и стандартный метод `addEventListener()` (который не поддерживается в IE), но имеет следующие существенные отличия:

- Поскольку модель событий в IE не поддерживает перехват событий, методы `attachEvent()` и `detachEvent()` принимают всего два аргумента: тип события и функцию-обработчик.
- Имена обработчиков событий, передаваемых методам IE, должны включать префикс «on». Например, при регистрации обработчика событий методом `attachEvent()` следует использовать строку `"onclick"`, тогда как при регистрации методом `addEventListener()` – строку `"click"`.
- Функции, зарегистрированные методом `attachEvent()`, не получают при вызове объект `Event` в виде аргумента. Вместо этого они должны обращаться к свойству `event` объекта `Window`.
- Функции, зарегистрированные методом `attachEvent()`, вызываются как глобальные функции, а не как методы элемента документа, в котором возникло событие. То есть когда вызываются обработчики, зарегистрированные методом `attachEvent()`, ключевое слово `this` ссылается на объект `Window`, а не на элемент, получивший событие.
- Метод `attachEvent()` позволяет несколько раз зарегистрировать один и тот же обработчик. Когда возникает событие, обработчик будет вызван столько раз, сколько раз он был зарегистрирован.

Этот метод определяется и работает аналогичным образом в объектах `Document` и `Window`.

См. также `Element.addEventListener()`, `Event`; глава 17

Element.detachEvent() IE 4

удаляет обработчик события

Синтаксис

```
void detachEvent(String тип,
                 Function обработчик)
```

Аргументы

тип

Тип события, обработчик которого будет удален. Тип события должен начинаться с префикса «on», например `"onload"`, `"onclick"` или `"onmousedown"`.

обработчик

Функция-обработчик, которая должна быть удалена.

Описание

Данный метод отменяет регистрацию обработчика событий, выполненную методом `attachEvent()`. Это аналог стандартного метода `removeEventListener()`, реализованный в IE. Чтобы удалить обработчик события из элемента, достаточно просто вызвать метод `detachEvent()` с теми же аргументами, с которыми вызывался метод `attachEvent()`.

Этот метод определяется и работает аналогичным образом в объектах `Document` и `Window`.

Element.dispatchEvent() DOM Level 2 Events

передает искусственное событие в данный узел

Синтаксис

```
boolean dispatchEvent(Event событие)
    throws EventException;
```

Аргументы

событие Объект `Event`, представляющий событие.

Возвращаемое значение

Значение `false`, если метод `preventDefault()` объекта *событие* вызывается всякий раз в процессе распространения события, `true` – в противном случае.

Исключения

Этот метод возбуждает исключение, если объект *событие* не инициализирован, а также в случае, если его свойство `type` содержит значение `null` или пустую строку.

Описание

Данный метод отправляет искусственное событие, созданное методом `Document.createEvent()` и инициализированное методом инициализации, определяемым интерфейсом `Event` или одним из его подынтерфейсов. Узел, в котором вызывается этот метод, становится получателем события, но событие сначала распространяется вниз по дереву документа для прохождения фазы захвата, а затем, если свойство `bubbles` объекта события имеет значение `true`, «всплывает» вверх по дереву документа после того, как обработано узлом-получателем.

См. также `Document.createEvent()`, `Event.initEvent()`, `MouseEvent.initMouseEvent()`

Element.getAttribute() DOM Level 1 Core

возвращает строковое значение атрибута с указанным именем

Синтаксис

```
String getAttribute(String имя);
```

Аргументы

имя Имя атрибута, значение которого должно быть возвращено.

Возвращаемое значение

Строковое значение атрибута с указанным именем. Если в документе не указано значение атрибута, возвращаемым значением будет пустая строка. Однако некоторые реализации в этом случае возвращают значение null.

Описание

Метод getAttribute() возвращает значение указанного атрибута для элемента. Обратите внимание, что объекты, представляющие HTML-элементы, определяют JavaScript-свойства, соответствующие стандартным HTML-атрибутам, поэтому надобность в этом методе возникает только при необходимости обратиться к нестандартным атрибутам.

В XML-документах значения атрибутов недоступны непосредственно как свойства элемента, и к ним надо обращаться путем вызова метода. Для XML-документов, в которых используются пространства имен, может потребоваться метод getAttributeNS() или getAttributeNodeNS().

Пример

Следующий фрагмент иллюстрирует два различных способа получения значения атрибута для HTML-элемента <img>:

```
// Получаем все изображения в документе
var images = document.body.getElementsByTagName("img");
// Получаем атрибут src первого изображения
var src0 = images[0].getAttribute("src");
// Получаем атрибут src второго изображения путем чтения свойства
var src1 = images[1].src;
```

См. также Element.getAttributeNode(), Element.getAttributeNS(), Node

Element.getAttributeNode() DOM Level 1 Core

возвращает узел Attr для атрибута с указанным именем

Синтаксис

```
Attr getAttributeNode(String имя);
```

Аргументы

имя Имя требуемого атрибута.

Возвращаемое значение

Узел Attr, который представляет значение атрибута с указанным именем, или null, если в этом элементе нет такого атрибута.

Описание

Метод getAttributeNode() возвращает узел Attr, представляющий значение атрибута с указанным именем. Обратите внимание: этот узел Attr может быть получен также через свойство attributes, унаследованное от интерфейса Node.

См. также Element.getAttribute(), Element.getAttributeNodeNS()

Element.getAttributeNodeNS() DOM Level 2 Core

возвращает узел Attr для атрибута с пространством имен

Синтаксис

```
Attr getAttributeNodeNS(String URIпространстваИмен,
                        String локальноеИмя);
```

Аргументы

URIпространстваИмен

URI, уникально идентифицирующий пространство имен этого атрибута, или `null`, если пространства имен нет.

локальноеИмя

Идентификатор, задающий имя атрибута в его пространстве имен.

Возвращаемое значение

Узел `Attr`, который представляет значение указанного атрибута, или `null`, если в этом элементе нет такого атрибута.

Описание

Этот метод работает так же, как `getAttributeNode()`, за исключением того, что атрибут задается комбинацией URI пространства имен и локального имени, определенного в данном пространстве имен. Применение этого метода имеет смысл только для XML-документов, использующих пространства имен.

См. также `Element.getAttributeNode()`, `Element.getAttributeNS()`

Element.getAttributeNS() DOM Level 2 Core

возвращает значение атрибута, использующего пространства имен

Синтаксис

```
String getAttributeNS(String URIпространстваИмен,
                      String локальноеИмя);
```

Аргументы

URIпространстваИмен

URI, уникально идентифицирующий пространство имен этого атрибута, или `null`, если пространства имен нет.

локальноеИмя

Идентификатор, задающий имя атрибута в его пространстве имен.

Возвращаемое значение

Строковое значение атрибута с указанным именем. Если в документе искомый атрибут не определен, возвращаемым значением будет пустая строка, но некоторые реализации в этом случае возвращают значение `null`.

Описание

Этот метод работает точно так же, как метод `getAttribute()`, кроме того, что атрибут задается комбинацией URI пространства имен и локального имени, определенного

в данном пространстве имен. Применение этого метода имеет смысл только для XML-документов, использующих пространства имен.

См. также `Element.getAttribute()`, `Element.getAttributeNodeNS()`

Element.getElementsByTagName() DOM Level 1 Core

находит элементы-потомки с указанным именем тега

Синтаксис

```
Element[] getElementsByTagName(String имя);
```

Аргументы

имя

Имя тега для требуемых элементов или значение «*», обозначающее, что должны возвращаться все элементы-потомки независимо от имени тега.

Возвращаемое значение

Массив (формально – `NodeList`) объектов `Element`, являющихся потомками этого элемента и имеющих указанное имя тега.

Описание

Этот метод обходит всех потомков этого элемента и возвращает массив (фактически – объект `NodeList`) узлов `Element`, представляющих все элементы документа с указанным именем тега. Элементы в возвращаемом массиве расположены в том же порядке, в котором они присутствуют в исходном документе.

Обратите внимание: в интерфейсе `Document` имеется также метод `getElementsByTagName()`, который работает аналогично данному, но обходит весь документ, а не только потомков одного элемента. Не путайте этот метод с `HTMLDocument.getElementsByName()`, который выполняет поиск элементов на основе значения их атрибута `name`, а не имени тега.

Пример

Все теги `<div>` в документе позволяет найти следующий код:

```
var divisions = document.body.getElementsByTagName("div");
```

А этот код может найти все теги `<p>` внутри первого тега `<div>`:

```
var paragraphs = divisions[0].getElementsByTagname("p");
```

См. также

`Document.getElementById()`, `Document.getElementsByTagName()`, `HTMLDocument.getElementsByName()`

Element.getElementsByTagNameNS() DOM Level 2 Core

находит элементы-потомки с указанными именем и пространством имен

Синтаксис

```
Node[] getElementsByTagNameNS(String URIпространстваИмен,
                              String локальноеИмя);
```

Аргументы

URIпространстваИмен

URI-значение, уникально идентифицирующее пространство имен элемента.

локальноеИмя

Идентификатор, задающий имя элемента внутри пространства имен.

Возвращаемое значение

Доступный только для чтения массив (формально – `NodeList`) объектов `Element`, являющихся потомками этого элемента и имеющих указанные имя и пространство имен.

Описание

Этот метод работает так же, как `getElementsByTagName()`, но имя тега требуемого элемента задается в виде комбинации URI пространства имен и локального имени, определенного внутри пространства имен. Применение этого метода имеет смысл только при работе с XML-документами, использующими пространства имен.

См. также `Document.getElementsByTagNameNS()`, `Element.getElementsByTagName()`

Element.hasAttribute() DOM Level 2 Core

определяет, имеется ли в этом элементе указанный атрибут

Синтаксис

```
boolean hasAttribute(String имя);
```

Аргументы

имя Имя требуемого атрибута.

Возвращаемое значение

Значение `true`, если этот элемент имеет заданное значение или значение, предлагаемое по умолчанию для указанного атрибута, и `false` – в противном случае.

Описание

Этот метод определяет, имеется ли в элементе атрибут с указанным именем, но не возвращает значение этого атрибута. Обратите внимание: `hasAttribute()` возвращает `true`, если указанный атрибут явно задан в документе, а также если тип документа имеет значение, предлагаемое по умолчанию для этого атрибута.

См. также `Element.getAttribute()`, `Element.setAttribute()`

Element.hasAttributeNS() DOM Level 2 Core

определяет, имеется ли в этом элементе указанный атрибут

Синтаксис

```
boolean hasAttributeNS(String URIпространстваИмен,
                       String локальноеИмя);
```

Аргументы

URIпространстваИмен

Уникальный идентификатор пространства имен для атрибута или `null` в случае отсутствия пространства имен.

локальноеИмя

Имя атрибута внутри указанного пространства имен.

Возвращаемое значение

Значение `true`, если в этом элементе есть явно заданное значение или значение, предлагаемое по умолчанию для указанного атрибута, в противном случае – `false`.

Описание

Этот метод работает точно так же, как `hasAttribute()`, за исключением того, что проверяемый атрибут задается пространством имен и именем. Применение этого метода имеет смысл только в XML-документах, использующих пространства имен.

См. также

`Element.getAttributeNS()`, `Element.hasAttribute()`, `Element.setAttributeNS()`

Element.removeAttribute() DOM Level 1 Core

удаляет атрибут с указанным именем из элемента

Синтаксис

```
void removeAttribute(String имя);
```

Аргументы

имя Имя удаляемого атрибута.

Исключения

Этот метод может генерировать исключение `DOMException` с кодом `NO_MODIFICATION_ALLOWED_ERR`, если элемент доступен только для чтения и не допускает удаления своих атрибутов.

Описание

Метод `removeAttribute()` удаляет атрибут с указанным именем из элемента. Если для указанного атрибута типом документа задается значение по умолчанию, последующие вызовы `getAttribute()` будут возвращать это значение. Попытки удалить несуществующие атрибуты или атрибуты, которые не указаны, но имеют значения по умолчанию, игнорируются.

См. также

`Element.getAttribute()`, `Element.setAttribute()`, `Node`

Element.removeAttributeNode() DOM Level 1 Core

удаляет узел Attr из элемента

Синтаксис

```
Attr removeAttributeNode(Attr старыйАтрибут)
    throws DOMException;
```

Аргументы

старыйАтрибут

Узел Attr, который должен быть удален из элемента.

Возвращаемое значение

Удаленный узел Attr.

Исключения

Этот метод может генерировать исключение DOMException со следующими значениями code:

NO_MODIFICATION_ALLOWED_ERR

Элемент доступен только для чтения и не допускает удаления атрибутов.

NOT_FOUND_ERR

Аргумент *старыйАтрибут* не является атрибутом элемента.

Описание

Этот метод удаляет (и возвращает) узел Attr из набора атрибутов элемента. Если для удаляемого атрибута в DTD задано значение по умолчанию, добавляется новый узел Attr, представляющий это значение. Вместо этого метода часто проще применить метод removeAttribute().

См. также Attr, Element.removeAttribute()

Element.removeAttributeNS() DOM Level 2 Core

удаляет атрибут, заданный именем и пространством имен

Синтаксис

```
void removeAttributeNS(String URIпространстваИмен,
                       String локальноеИмя);
```

Аргументы

URIпространстваИмен

Уникальный идентификатор для пространства имен атрибута или null в случае отсутствия пространства имен.

локальноеИмя

Имя атрибута в указанном пространстве имен.

Исключения

Этот метод может генерировать исключение DOMException с кодом NO_MODIFICATION_ALLOWED_ERR, если элемент доступен только для чтения и не допускает удаления своих атрибутов.

Описание

Метод removeAttributeNS() работает точно так же, как removeAttribute(), за исключением того, что удаляемый атрибут задается именем и пространством имен, а не просто именем. Применение этого метода имеет смысл только для XML-документов, использующих пространства имен.

См. также

Element.getAttributeNS(), Element.removeAttribute(), Element.setAttributeNS()

Element.removeEventListener() DOM Level 2 Events

удаляет обработчик события

Синтаксис

```
void removeEventListener(String тип,
                         Function обработчик,
                         Boolean фазаЗахвата);
```

Аргументы

тип

Тип события, обработчик которого удаляется.

обработчик

Функция-обработчик события, которая должна быть удалена.

фазаЗахвата

Значение true, если удаляется перехватывающий обработчик события, и false, если удаляется обычный обработчик события.

Описание

Метод удаляет указанную функцию-обработчик события. Аргументы *тип* и *фазаЗахвата* должны иметь те же значения, что и при вызове метода addEventListener(). Если обработчик, соответствующий указанным значениям аргументов, не найден, этот метод ничего делать не будет.

После того как функция-обработчик события будет удалена этим методом, она больше не сможет вызываться для обработки событий указанного *типа* в данном узле. Это верно даже для случая, когда обработчик события удаляется из другого обработчика события того же типа в том же узле.

Этот метод также определен и работает аналогичным образом в объектах Document и Window.

Element.setAttribute() DOM Level 1 Core

создает или изменяет значение атрибута элемента

Синтаксис

```
void setAttribute(String имя,
                  String значение)
    throws DOMException;
```

Аргументы

имя Имя создаваемого или модифицируемого атрибута.

значение Строковое значение атрибута.

Исключения

Этот метод может генерировать исключение DOMException со следующими значениями code:

INVALID_CHARACTER_ERR

В аргументе имя есть символ, недопустимый в именах HTML- и XML-атрибутов.

NO_MODIFICATION_ALLOWED_ERR

Элемент доступен только для чтения и не допускает модификации своих атрибутов.

Описание

Метод устанавливает указанный атрибут равным заданному значению. Если атрибута с таким именем еще нет, создается новый атрибут. Обратите внимание: объекты HTMLElement в HTML-документе определяют JavaScript-свойства, соответствующие всем стандартным HTML-атрибутам. По этой причине данный метод обычно используется лишь для доступа к нестандартным атрибутам.

Пример

```
// Устанавливает атрибут TARGET всех ссылок в документе
var links = document.body.getElementsByTagName("a");
for(var i = 0; i < links.length; i++) {
    links[i].setAttribute("target", "newwindow");
    // или гораздо проще: links[i].target = "newwindow";
}
```

См. также

Element.getAttribute(), Element.removeAttribute(), Element.setAttributeNode()

Element.setAttributeNode() DOM Level 1 Core

добавляет к элементу новый узел Attr

Синтаксис

```
Attr setAttributeNode(Attr новыйАтрибут)
    throws DOMException;
```

Аргументы

новыйАтрибут

Узел Attr, представляющий атрибут, который должен быть добавлен или значение которого должно быть изменено.

Возвращаемое значение

Узел Attr, который был заменен аргументом *новыйАтрибут*, или null, если никакой атрибут не был заменен.

Исключения

Этот метод может генерировать исключение DOMException с одним из перечисленных ниже значений code:

INUSE_ATTRIBUTE_ERR

Аргумент *новыйАтрибут* уже входит в набор атрибутов какого-либо другого узла Element.

NO_MODIFICATION_ALLOWED_ERR

Узел Element доступен только для чтения и его атрибуты нельзя модифицировать.

WRONG_DOCUMENT_ERR

Значение свойства ownerDocument нового атрибута отличается у элемента, для которого оно устанавливается.

Описание

Метод добавляет новый узел Attr к набору атрибутов узла Element. Если атрибут с тем же именем для элемента уже существует, он заменяется аргументом *новыйАтрибут*, а замененный узел Attr становится возвращаемым значением. Если такого атрибута еще нет, этот метод определяет новый атрибут для элемента.

Обычно вместо setAttributeNode() проще использовать метод setAttribute().

См. также

Attr, Document.createAttribute(), Element.setAttribute()

Element.setAttributeNodeNS() DOM Level 2 Core

добавляет к элементу новый узел Attr с пространством имен

Синтаксис

```
Attr setAttributeNodeNS(Attr новыйАтрибут)
    throws DOMException;
```

Аргументы

новыйАтрибут

Узел Attr, представляющий атрибут, который должен быть добавлен или значение которого должно быть изменено.

Возвращаемое значение

Узел Attr, который был заменен аргументом *новыйАтрибут*, или null, если никакой атрибут не был заменен.

Исключения

Этот метод генерирует исключения по тем же причинам, что и setAttributeNode(). Он может также генерировать исключение DOMException с кодом NOT_SUPPORTED_ERR, сигнализируя о том, что метод не реализован, потому что текущая реализация не поддерживает XML-документы и пространства имен.

Описание

Этот метод работает точно так же, как setAttributeNode(), за исключением того, что он предназначен для использования с узлами Attr, представляющими атрибуты, задаваемые пространством имен и именем.

Применение этого метода имеет смысл только при работе с XML-документами, использующими пространства имен. Он может быть не реализован (т. е. генерировать исключение NOT_SUPPORTED_ERR) в броузерах, не поддерживающих XML-документы.

См. также

Attr, Document.createAttributeNS(), Element.setAttributeNS(), Element.setAttributeNode()

Element.setAttributeNS() DOM Level 2 Core

создает или изменяет атрибут с пространством имен

Синтаксис

```
void setAttributeNS(String URIпространстваИмен,
                    String уточненноеИмя,
                    String значение)
    throws DOMException;
```

Аргументы

URIпространстваИмен

URI, уникально идентифицирующий пространство имен атрибута, который должен быть установлен или создан, или `null` в случае отсутствия пространства имен.

уточненноеИмя

Имя атрибута, заданное в виде префикса пространства имен, за которым следуют двоеточие и имя внутри этого пространства имен.

значение

Новое значение атрибута.

Исключения

Этот метод может генерировать исключение `DOMException` со следующими значениями `code`:

`INVALID_CHARACTER_ERR`

Аргумент *уточненноеИмя* содержит символ, недопустимый в именах HTML- или XML-атрибутов.

`NAMESPACE_ERR`

Формат аргумента *уточненноеИмя* неверен или имеется несоответствие между префиксом пространства имен в аргументе *уточненноеИмя* и аргументом *URIпространстваИмен*.

`NO_MODIFICATION_ALLOWED_ERR`

Элемент доступен только для чтения и не допускает модификации своих атрибутов.

`NOT_SUPPORTED_ERR`

Реализация DOM не поддерживает XML-документы.

Описание

Этот метод похож на `setAttribute()`, за исключением того, что создаваемый или устанавливаемый атрибут задается в виде URI пространства имен и уточненного имени, состоящего из префикса пространства имен, двоеточия и локального имени внутри пространства имен.

Применение этого метода имеет смысл только при работе с XML-документами, использующими пространства имен. Он может быть не реализован (т. е. генерировать исключение `NOT_SUPPORTED_ERR`) в броузерах, не поддерживающих XML-документы.

См. также `Element.setAttribute()`, `Element.setAttributeNode()`

Event

DOM Level 2 Events, IE

информация о событии **Object→Event**

Подынтерфейсы

UIEvent

Стандартные Свойства

Следующие свойства определяются стандартом DOM Level 2 Events. Информацию о дополнительных свойствах некоторых типов событий вы найдете в справочных статьях об объектах KeyEvent, MouseEvent и UIEvent:

`readonly boolean bubbles`

Значение `true`, если тип события поддерживает «всплытие» (и если не вызван метод `stopPropagation()`), и `false` – в противном случае.

`readonly boolean cancelable`

Значение `true`, если действие, предлагаемое по умолчанию и связанное с событием, может быть отменено с помощью метода `preventDefault()`, и `false` – в противном случае.

`readonly Object currentTarget`

Объект `Element`, `Document` или `Window`, обрабатывающий событие в данный момент. Во время фазы перехвата и всплытия значение свойства отличается от `target`.

`readonly unsigned short eventPhase`

Текущая стадия распространения события. Значение свойства – одна из трех перечисленных далее констант, которые представляют фазы захвата, нормальной обработки события и всплытия:

Константа фазы события	Значение
`Event.CAPTURING_PHASE`	1
`Event.AT_TARGET`	2
`Event.BUBBLING_PHASE`	3

`readonly Object target`

Целевой узел события, т. е. объект `Element`, `Document` или `Window`, в котором было сгенерировано событие.

`readonly Date timeStamp`

Дата и время, когда произошло событие (или, формально, когда был создан объект `Event`). Реализации не обязаны предоставлять допустимое значение времени в этом поле, таким образом, метод `getTime()` данного объекта `Date` должен возвращать 0. (См. описание объекта `Date` в третьей части книги.)

`readonly String type`

Имя события, представляемого данным объектом `Event`. Это имя, с которым был зарегистрирован обработчик события, или имя свойства обработчика события без префикса «on», например `"click"`, `"load"` или `"submit"`.

Свойства IE

Internet Explorer не поддерживает (по крайней мере, вплоть до IE 7) модель событий стандарта DOM, а объект `Event` в IE определяет совершенно иной набор свойств.

Модель событий IE не предоставляет иерархию наследования для событий различных типов, поэтому все перечисленные далее свойства в равной степени относятся к событиям любого типа:

`boolean altKey`

Указывает, удерживалась ли нажатой клавиша Alt в момент события.

`integer button`

Для событий от мыши свойство `button` указывает, какая или какие кнопки мыши были нажаты. Это значение представляет собой битовую маску: если установлен бит, соответствующий числу 1, значит, была нажата левая кнопка мыши, соответствующий числу 2, – была нажата правая кнопка мыши, соответствующий числу 4, – была нажата средняя кнопка мыши (на трехкнопочных моделях).

`boolean cancelBubble`

Чтобы в обработчике события остановить дальнейшее распространение события вверх по дереву вмещающих элементов, в это свойство нужно записать значение `true`.

`integer clientX, clientY`

Координаты относительно страницы веб-броузера, в которой возникло событие.

`boolean ctrlKey`

Указывает, удерживалась ли нажатой клавиша Ctrl в момент события.

`Element fromElement`

Для событий `mouseover` и `mouseout` свойство `fromElement` содержит ссылку на объект, с которого двигался указатель мыши.

`integer keyCode`

Для событий нажатия клавиш это свойство содержит код Unicode символа, сгенерированный нажатой клавишей. Для событий `keydown` и `keyup` содержит виртуальный код нажатой клавиши. Виртуальные коды могут зависеть от используемой раскладки клавиатуры.

`integer offsetX, offsetY`

Координаты, в которых возникло событие, в координатной системе элемента-источника события (см. описание свойства `srcElement`).

`boolean returnValue`

Если установлено, значение этого свойства имеет приоритет перед значением, фактически возвращаемым обработчиком события. Данное свойство следует установить в `false`, чтобы предотвратить исполнение действия, предлагаемого по умолчанию элементом-источником, в котором возникло событие.

`integer screenX, screenY`

Координаты относительно экрана, в которых возникло событие.

`boolean shiftKey`

Указывает, удерживалась ли нажатой клавиша Shift в момент события.

`Object srcElement`

Ссылка на объект `Window`, `Document` или `Element`, ставший источником события.

`Element toElement`

Для событий `mouseover` и `mouseout` содержит ссылку на объект, в пределы которого был перемещен указатель мыши.

`String type`

Тип события. Значением этого свойства является имя события без префикса «on». Например, когда вызывается обработчик события `onclick()`, свойство `type` объекта `Event` будет содержать значение `"click"`.

`integer x, y`

Координаты X и Y относительно документа или самого внутреннего элемента-контейнера, который был динамически позиционирован средствами CSS.

Стандартные Методы

Следующие методы определяются спецификацией DOM Lebel 2 Events. В модели событий IE объект `Event` не имеет методов:

`initEvent()`

Инициализирует свойства только что созданного объекта `Event`.

`preventDefault()`

Сообщает броузеру о том, чтобы он не выполнял действие, предлагаемое по умолчанию для этого события, если таковое существует. Если событие не относится к отменяемому типу, этот метод не выполняет никаких действий.

`stopPropagation()`

Прекращает дальнейшее распространение события через стадии перехвата, обработки и всплытия. После вызова этого метода вызываются остальные обработчики того же события в том же узле, но событие не передается каким-либо другим узлам.

Описание

Свойства объекта `Event` содержат информацию о событии, например, ссылку на элемент, в котором возникло событие. С помощью методов объекта `Event` можно управлять распространением события. Стандарт DOM Level 2 Events определяет стандартную модель событий, реализованную всеми современными броузерами, за исключением Internet Explorer, который определяет собственную несовместимую модель. В этой справочной статье перечислены свойства как стандартного объекта `Event`, так и объекта `Event` в IE. Дополнительную информацию об этих двух моделях событий можно найти в главе 17. Однако следует отметить, что в стандартной модели событий функция-обработчик получает объект `Event` в виде аргумента, тогда как в модели событий IE он хранится в свойстве `event` объекта `Window`.

В стандартной модели событий различные подынтерфейсы интерфейса `Event` определяют дополнительные свойства, предоставляющие сведения об определенных типах событий. В модели событий IE существует только один тип объекта `Event`, и он используется для представления событий всех типов.

См. также `KeyEvent`, `MouseEvent`, `UIEvent`; глава 17

Event.initEvent() DOM Level 2 Events

инициализирует свойства нового события

Синтаксис

```
void initEvent(String eventTypeArg,
               boolean canBubbleArg,
               boolean cancelableArg);
```

Аргументы

eventTypeArg

Тип события. Это может быть один из предопределенных типов событий, таких как `"load"` или `"submit"`, либо специальный тип, выбранный программистом. Однако имена, начинающиеся с префикса «DOM», зарезервированы.

canBubbleArg

Возможность всплытия события.

cancelableArg

Возможность отмены события с помощью метода `preventDefault()`.

Описание

Метод инициализирует свойства `type`, `bubbles` и `cancelable` искусственного объекта `Event`, созданного методом `Document.createEvent()`. Метод может вызываться для вновь созданных объектов `Event` только до их диспетчеризации с использованием метода `EventTarget.dispatchEvent()`.

См. также

`Document.createEvent()`, `MouseEvent.initMouseEvent()`, `UIEvent.initUIEvent()`

Event.preventDefault() DOM Level 2 Events

отменяет действие, предлагаемое для события по умолчанию

Синтаксис

```
void preventDefault();
```

Описание

Метод сообщает броузеру, чтобы он не выполнял предлагаемое по умолчанию действие для этого события (если оно есть). Например, если свойство `type` равно `"submit"`, любой обработчик события, вызванный во время любой стадии обработки события, может отменить передачу данных формы, вызвав этот метод. Обратите внимание: если свойство `cancelable` объекта `Event` равно `false`, то действие, предлагаемое по умолчанию, либо отсутствует, либо не может быть отменено. В любой из этих ситуаций вызов данного метода не будет иметь никакого эффекта.

Event.stopPropagation() DOM Level 2 Events

останавливает дальнейшее распространение события

Синтаксис

```
void stopPropagation();
```

Описание

Метод останавливает распространение события и отменяет его передачу другим узлам документа. Он может вызываться в ходе любой стадии распространения события. Обратите внимание: этот метод не отменяет вызов любых других обработчиков событий того же узла документа, но предотвращает передачу событий любым другим узлам.

ExternalInterface

объект ActionScript во Flash 8

двунаправленный интерфейс взаимодействия с Flash

Статические свойства

`available`

Указывает, возможно ли взаимодействие между Flash и JavaScript. Содержит значение `false`, если броузер запрещает такое взаимодействие в соответствии с политикой безопасности.

Статические функции

`addCallback()`

Экспортирует ActionScript-метод так, чтобы его можно было вызывать из JavaScript-сценария.

`call()`

Вызывает JavaScript-функцию из ActionScript-сценария.

Описание

`ExternalInterface` – это ActionSript-объект, который определяется в модуле расширения Flash 8 и более поздних версиях. Он задает две статические функции для использования ActionScript-кодом в Flash-роликах. Эти функции позволяют организовать взаимодействие между JavaScript-сценарием в веб-броузере и ActionCript-сценарием в Flash-ролике.

См. также

`FlashPlayer`; глава 23

ExternalInterface.addCallback()

функция ActionScript во Flash 8

экспортирует метод ActionScript для вызова из JavaScript

Синтаксис

```
boolean ExternalInterface.addCallback(String имя,
                                      Object экземпляр,
                                      Function функция)
```

Аргументы

имя

Имя, под которым будет экспортироваться функция. Вызов функции с этим именем приводит к вызову ActionScript-функции *функция* как метода объекта *экземпляр*.

экземпляр

ActionScript-объект, в контексте которого будет вызвана *функция*, или `null`. При вызове *функции* этот аргумент станет значением ключевого слова `this`.

функция

ActionScript-функция, которая вызывается при обращении к JavaScript-функции *имя*.

Возвращаемое значение

Значение `true` в случае успеха и `false` в случае неудачи.

Описание

Данная функция используется ActionScript-сценарием в Flash-ролике, чтобы обеспечить возможность JavaScript-сценарию вызвать в веб-броузере ActionScript-функцию. Вызов `addCallback()` определяет JavaScript-функцию верхнего уровня с именем *имя*, которая при обращении к ней вызывает ActionScript-функцию *функция* в качестве метода ActionScript-объекта *экземпляр*.

Аргументы JavaScript-функции преобразуются и передаются *функции*, а возвращаемое значение *функции* преобразуется и становится возвращаемым значением JavaScript-функции. Аргументы и возвращаемое значение могут быть значениями элементарных типов: числами, строками, логическими значениями, а также объектами и массивами, содержащими значения элементарных типов. Однако из клиентского JavaScript-кода невозможно передать ActionScript-функции такие объекты, как `Window` или `Document`. Точно так же невозможно вернуть JavaScript-сценарию специфичный для Flash ActionScript-объект, такой как `MovieClip`.

См. также `FlashPlayer`; глава 23

ExternalInterface.call()

функция ActionScript во Flash 8

вызывает функцию JavaScript из ActionScript

Синтаксис

```
Object ExternalInterface.call(String имя,
                              Object аргументы...)
```

Аргументы

имя

Имя вызываемой JavaScript-функции.

аргументы...

Ноль и более аргументов, которые преобразуются и передаются JavaScript-функции.

Возвращаемое значение

Возвращаемое значение JavaScript-функции, преобразованное в ActionScript-значение.

Описание

Эта статическая функция используется ActionScript-сценарием в Flash-ролике для вызова JavaScript-функции, определенной в веб-броузере, куда встраивается Flash-ролик. Подробнее порядок преобразования входных аргументов и возвращаемых значений между ActionScript- и JavaScript-кодом описывается в справочной статье о методе `ExternalInterface.addCallback()`.

FileUpload

См. статью об объекте Input

FlashPlayer Flash 2.0

модуль расширения для воспроизведения Flash-роликов

Методы

`GetVariable()`

Возвращает значение переменной, определенной во Flash-ролике.

`GotoFrame()`

Выполняет переход к заданному кадру в ролике.

`IsPlaying()`

Проверяет, выполняется ли воспроизведение ролика.

`LoadMovie()`

Загружает дополнительный Flash-ролик и отображает его в заданном слое или уровне текущего ролика.

`Pan()`

Перемещает область просмотра ролика.

`PercentLoaded()`

Определяет, какая доля ролика уже загружена.

`Play()`

Начинает воспроизведение ролика.

`Rewind()`

Выполняет перемотку ролика на первый кадр.

`SetVariable()`

Устанавливает значение переменной, определенной в ролике.

`SetZoomRect()`

Устанавливает размеры и положение области просмотра Flash-ролика.

`StopPlay()`

Останавливает воспроизведение ролика.

`TotalFrames()`

Возвращает размер ролика в виде количества кадров.

`Zoom()`

Изменяет размер области просмотра.

Описание

Объект `FlashPlayer` представляет встроенный в веб-страницу Flash-ролик и является экземпляром подключаемого Flash-модуля, который призван воспроизводить ролик. Получить ссылку на объект `FlashPlayer` можно, например, методом `Document.getElementById()`, используя тег `<embed>` или `<object>`, с помощью которого Flash-ролик встраивается в веб-страницу.

После того как ссылка на объект `FlashPlayer` получена, можно с помощью различных JavaScript-методов управлять воспроизведением ролика, а также устанавливать и получать значения переменных. Обратите внимание: имена всех методов `FlashPlayer` начинаются с прописной буквы, что не соответствует соглашениям об именовании, принятым в JavaScript.

См. также

Глава 23.

FlashPlayer.GetVariable() Flash 4

возвращает значение, определенное во Flash-ролике

Синтаксис

```
String GetVariable(String имяПеременной)
```

Аргументы

имяПеременной

Имя переменной, определенной во Flash-ролике.

Возвращаемое значение

Значение именованной переменной в виде строки или `null`, если переменной с таким именем не существует.

FlashPlayer.GotoFrame() Flash 2

выполняет переход к заданному кадру в ролике

Синтаксис

```
void GotoFrame(integer номерКадра)
```

Аргументы

номерКадра

Номер кадра, переход к которому выполняется.

Описание

Эта функция выполняет переход к кадру с заданным номером или к последнему доступному кадру, если заданный кадр еще не загружен. Чтобы избежать неопределенности, перед выполнением перехода можно методом `PercentLoaded()` определить, какая доля ролика уже загружена.

FlashPlayer.IsPlaying() Flash 2

проверяет, выполняется ли воспроизведение ролика

Синтаксис

```
boolean IsPlaying()
```

Возвращаемое значение

Значение `true`, если ролик воспроизводится, и `false` – в противном случае.

FlashPlayer.LoadMovie() Flash 3

загружает дополнительный ролик

Синтаксис

```
void LoadMovie(integer слой,
               String url)
```

Аргументы

слой Слой, или уровень, в текущем ролике, где должен отображаться вновь загруженный ролик.

url URL загружаемого ролика.

Описание

Этот метод загружает дополнительный ролик с URL-адреса, заданного аргументом *url*, и отображает его в заданном *слое* внутри текущего ролика.

FlashPlayer.Pan() Flash 2

перемещает область просмотра ролика

Синтаксис

```
void Pan(integer dx, integer dy,
         integer режим)
```

Аргументы

dx, *dy*

Величина перемещения по горизонтали и вертикали.

режим

Этот аргумент определяет, как должны интерпретироваться аргументы *dx* и *dy*. Если этот аргумент имеет значение 0, остальные аргументы интерпретируются как количество пикселов. Если этот аргумент имеет значение 1, остальные аргументы интерпретируются как проценты.

Описание

Flash-плеер определяет область просмотра, внутри которой будет отображаться Flash-ролик. Как правило, размер области просмотра совпадает с размером ролика, но такое соответствие может быть нарушено методом `SetZoomRect()` или `Zoom()`: эти методы могут изменять размеры области просмотра, в результате чего в ней может уместиться лишь фрагмент ролика.

Когда ролик не умещается полностью в области просмотра, методом `Pan()` можно перемещать область просмотра, чтобы показывать различные фрагменты изображения. Однако этот метод не позволяет смещать область просмотра за границы ролика.

См. также `FlashPlayer.SetZoomRect()`, `FlashPlayer.Zoom()`

FlashPlayer.PercentLoaded() Flash 2

определяет, какая доля ролика уже загружена

Синтаксис

```
integer PercentLoaded()
```

Возвращаемое значение

Целое число в диапазоне от 0 до 100, представляющее примерную долю ролика в процентах, которая уже загружена в проигрыватель.

FlashPlayer.Play() Flash 2

начинает воспроизведение ролика

Синтаксис

```
void Play()
```

Описание

Начинает воспроизведение ролика.

FlashPlayer.Rewind() Flash 2

выполняет перемотку ролика на первый кадр

Синтаксис

```
void Rewind()
```

Описание

Метод выполняет перемотку ролика на первый кадр.

FlashPlayer.SetVariable() Flash 4

устанавливает значение переменной, определенной в Flash-ролике

Синтаксис

```
void SetVariable(String имя, String значение)
```

Аргументы

имя Имя переменной, значение которой изменяется.

значение Новое значение указанной переменной. Это значение должно быть строкой.

Описание

Метод устанавливает новое значение переменной, определенной в ролике.

FlashPlayer.SetZoomRect() Flash 2

устанавливает размер и положение области просмотра ролика

Синтаксис

```
void SetZoomRect(integer left, integer top,
                 integer right, integer bottom)
```

Аргументы

left, *top*

Координаты в твипах верхнего левого угла области просмотра.

right, *bottom*

Координаты в твипах нижнего правого угла области просмотра.

Описание

Метод определяет координаты и размеры области просмотра ролика, т. е. задает вложенный прямоугольник внутри области Flash-плеера, где будет выводиться ролик.

Размеры Flash-роликов измеряются в единицах, известных как *твипы*. 20 твипов составляют один пункт, а 1440 твипов – один дюйм.

См. также `FlashPlayer.Pan()`, `FlashPlayer.Zoom()`

FlashPlayer.StopPlay() Flash 2

останавливает воспроизведение ролика

Синтаксис

```
void StopPlay()
```

Описание

Останавливает воспроизведение ролика.

FlashPlayer.TotalFrames() Flash 2

возвращает длину ролика в кадрах

Синтаксис

```
integer TotalFrames()
```

Описание

Метод возвращает длину ролика в кадрах.

FlashPlayer.Zoom() Flash 2

увеличивает или уменьшает масштаб

Синтаксис

```
void Zoom(integer процент)
```

Аргументы

процент Процент изменения масштаба области просмотра или 0 для восстановления ее полного размера.

Описание

Метод масштабирует область просмотра в соответствии с аргументом *процент*. Значения от 1 до 99 уменьшают размер области просмотра, что увеличивает размер объектов в ролике. Значения больше 100 увеличивают область просмотра (однако область просмотра не может превысить размеров ролика), что уменьшает размер объектов в ролике. Значение 0 – это отдельный случай: когда в виде аргумента передается значение 0, это приводит к восстановлению полного размера области просмотра и в результате весь ролик становится видимым.

Form DOM Level 2 HTML

тег <form> в HTML-документе **Node→Element→HTMLElement→Form**

Свойства

```
readonly HTMLCollection elements
```

Массив (`HTMLCollection`) всех элементов в форме. Подробности см. в описании объекта `Form.element[]`.

readonly long length

Количество элементов в форме. Эквивалентно свойству elements.length.

Помимо этих свойств объект Form определяет следующие свойства, которые соответствуют непосредственно HTML-атрибутам:

Свойство	Атрибут	Описание
String acceptCharset	acceptcharset	Наборы символов, которые могут приниматься сервером
String action	action	URL-адрес обработчика формы
String enctype	enctype	Способ кодирования данных формы
String method	method	HTTP-метод, используемый для передачи формы
String name	name	Имя формы
String target	target	Фрейм или окно, где должны отображаться результаты отправки данных формы

Методы

reset() Сбрасывает все элементы ввода формы к их значениям, предлагаемым по умолчанию.

submit() Передает данные из формы.

Обработчик событий

onreset

Вызывается непосредственно перед сбросом элементов формы.

onsubmit

Вызывается непосредственно перед передачей данных формы. Этот обработчик позволяет выполнить проверку значений формы перед их передачей на сервер.

Синтаксис HTML

Объект Form создается с помощью стандартного HTML-тега <form>. Форма содержит любые элементы ввода, создаваемые тегами <input>, <select>, <textarea> и другими:

```
<form
[ name="имя_формы" ]      // Имя формы, используемое JavaScript-сценарием
[ target="имя_окна" ]     // Имя окна для откликов
[ action="url" ]          // URL-адрес, куда отправляются данные формы
[ method=("get"|"post") ] // Метод передачи данных формы
[ enctype="кодировка" ]   // Способ кодирования данных формы
[ onreset="обработчик" ]  // Обработчик, вызываемый при сбросе формы
[ onsubmit="обработчик" ] // Обработчик, вызываемый при передаче данных формы
>
// Здесь находятся текст и элементы ввода формы
</form>
```

Описание

Объект Form представляет элемент <form> в HTML-документе. Свойство elements — это объект HTMLCollection, который дает удобный доступ ко всем элементам в форме. Методы submit() и reset() позволяют программным способом отправлять данные формы или сбрасывать все элементы формы в значения, предлагаемые по умолчанию.

Каждая форма в документе представлена элементом массива Document.forms[]. Именованные формы также представлены свойством *имя_формы* соответствующего объекта Document, где *имя_формы* – это имя, заданное в атрибуте name тега <form>.

Элементы формы (кнопки, поля ввода, переключатели и т. д.) собраны в массиве Form.elements[]. К именованным элементам, как и к именованным формам, можно обращаться непосредственно по имени – имя элемента выступает в качестве имени свойства объекта Form. Другими словами, обратиться к элементу Input с именем phone в форме с именем questionnaire можно посредством следующего JavaScript-выражения:

```
document.questionnaire.phone
```

См. также Input, Select, Textarea; глава 18

Form.elements[] DOM Level 2 HTML

элементы ввода формы

Синтаксис

```
readonly HTMLCollection elements
```

Описание

Элемент elements[] – это подобный массиву объект HTMLCollection, содержащий элементы форм (такие как объекты Input, Select и Textarea). Элементы массива расположены в массиве в том же порядке, что и в исходном коде HTML-формы. Каждый элемент массива имеет свойство type, строковое значение которого определяет тип элемента.

Порядок использования

Если элементу массива elements[] присвоено имя с помощью атрибута name="*имя*" его HTML-тега <input>, это имя становится свойством объекта *form*, и это свойство ссылается на элемент. То есть можно ссылаться на объекты ввода по имени, а не по номеру:

```
form.имя
```

См. также Input, HTMLCollection, Select, Textarea

Form.onreset DOM Level 0

обработчик, вызываемый при сбросе формы

Синтаксис

```
Function onreset
```

Описание

Свойство onreset объекта Form задает функцию обработки события, вызываемую, когда пользователь щелкает на кнопке Reset формы или когда вызывается метод Form.reset(). Если обработчик onreset возвращает значение false, элементы формы не сбрасываются. Другой способ регистрации обработчиков событий описывается в справочной статье о методе Element.addEventListener().

См. также Element.addEventListener(), Form.onsubmit, Form.reset(); глава 17

Form.onsubmit DOM Level 0

обработчик, вызываемый при передаче данных формы

Синтаксис

```
Function onsubmit
```

Описание

Свойство `onsubmit` объекта `Form` задает функцию обработки события, вызываемую, когда пользователь передает данные из формы на сервер щелчком на кнопке Submit формы. Обратите внимание: этот обработчик события не вызывается при обращении к методу `Form.submit()`.

Если обработчик события `onsubmit` возвращает значение `false`, данные формы не передаются. Если обработчик возвращает любое другое значение или не возвращает ничего, данные формы передаются обычным образом. Так как обработчик `onsubmit` может отменить передачу данных формы, он является идеальным средством проверки этих данных.

Другой способ регистрации обработчиков событий описывается в справочной статье о методе `Element.addEventListener()`.

См. также `Element.addEventListener()`, `Form.onreset`, `Form.submit()`; глава 17

Form.reset() DOM Level 2 HTML

сбрасывает элементы формы

Синтаксис

```
void reset();
```

Описание

Метод `reset()` сбрасывает все элементы формы в значения, предлагаемые по умолчанию, в точности так же, как если бы пользователь щелкнул на кнопке Reset, за исключением того, что обработчик события `onreset()` формы не вызывается.

См. также `Form.onreset`, `Form.submit()`

Form.submit() DOM Level 2 HTML

отправляет данные формы на веб-сервер

Синтаксис

```
void submit();
```

Описание

Метод `submit()` передает значения элементов формы на сервер. Он отправляет данные точно так же, как если бы пользователь щелкнул на кнопке Submit, за исключением того, что обработчик события `onsubmit` формы не вызывается.

См. также `Form.onsubmit`, `Form.reset()`

Frame

DOM Level 2 HTML

тег <frame> в HTML-документе **Node→Element→HTMLElement→Frame**

Свойства

Как будет описано далее, HTML-фреймы могут быть доступны как объекты `Frame` или `Window`. Когда доступ к фреймам производится как к объектам `Frame`, они наследуют свойства `HTMLElement` и определяют следующие дополнительные свойства:

`Document contentDocument`

Документ, в котором находится содержимое фрейма.

`String src`

URL-адрес, откуда было загружено содержимое фрейма. Установка этого свойства вызывает загрузку во фрейм нового документа. Данное свойство является простым отражением атрибута `src` тега `<frame>` – это не объект `Location`, каковым является свойство `Window.location`.

Помимо этих объект `Frame` определяет следующие свойства, которые непосредственно соответствуют HTML-атрибутам:

Свойство	Атрибут	Описание
`String frameBorder`	`frameborder`	Содержит строку `"0"` для фреймов без рамки
`String longDesc`	`longdesc`	URL-адрес описания фрейма
`String marginHeight`	`marginheight`	Верхнее и нижнее поля фрейма
`String marginWidth`	`marginwidth`	Левое и правое поля фрейма
`String name`	`name`	Имя фрейма для выполнения поиска средствами DOM Level 0 и указания в атрибуте `target` форм
`boolean noResize`	`noresize`	Если содержит `true`, пользователь не сможет изменить размер фрейма
`String scrolling`	`scrolling`	Политика прокрутки фрейма: `"auto"`, `"yes"` или `"no"`

Описание

Фреймы имеют двойственную природу и в клиентском JavaScript-коде могут быть представлены как объектом `Window`, так и объектом `Frame`. В традиционной модели DOM Level 0 каждый тег `<frame>` рассматривается как независимое окно и представляется объектом `Window`, ссылку на который можно получить по его имени или обратившись к элементу массива `frames[]` объемлющего окна:

```
// Получить фрейм в виде объекта Window
var win1 = top.frames[0];    // Из массива frames[] по числовому индексу
var win2 = top.frames['f1']; // Из массива frames[] по имени
var win3 = top.f1;           // Из свойства родительского окна
```

Когда ссылка на фрейм приобретается таким образом, получается объект `Window` и свойства, перечисленные в предыдущем разделе «Свойства», оказываются недоступны. Вместо них следует использовать свойства объекта `Window`, такие как `document` для доступа к документу фрейма и `location` для доступа к URL-адресу документа.

В модели DOM Level 2 ссылки на элементы `<frame>` можно получить по атрибуту `id` или `name` тега, как и в случае любого другого элемента документа:

```
// Получить фрейм в виде объекта Frame
var frame1 = top.document.getElementById('f1');             // по id
var frame2 = top.document.getElementsByTagName('frame')[1]; // по имени тега
```

Когда ссылка на фрейм приобретается с использованием DOM-методов, как только что было продемонстрировано, получается не объект Window, а объект Frame, в котором доступны перечисленные ранее свойства. В этом случае для доступа к содержимому документа следует использовать свойство contentDocument, а для получения URL-адреса данного документа или загрузки нового – свойство src. Чтобы из объекта Frame получить объект Window, можно воспользоваться свойством f.contentDocument.defaultView.

Элементы <iframe> очень похожи на элементы <frame>. Они описываются в справочной статье об объекте IFrame.

Примечательно, что к документам, содержащим несколько фреймов, применяется все та же политика общего происхождения (см. раздел 13.8.2). Броузеры не позволяют обращаться к содержимому фреймов, происхождение которых отличается от происхождения документа, содержащего сценарий. Это истинно независимо от того, каким объектом представлен фрейм, Window или Frame.

См. также IFrame, Window; глава 14

Hidden

См. статью об объекте Input

History

JavaScript 1.0

журнал посещений броузера **Object→History**

Синтаксис

window.history
history

Свойства

length

Это числовое свойство задает количество URL-адресов в журнале (истории) посещений броузера. Знание размера этого списка не особенно полезно, поскольку нет способа определить индекс текущего отображаемого документа в этом списке.

Методы

back() Переход назад к ранее открывавшемуся URL-адресу.

forward() Переход вперед к ранее открывавшемуся URL-адресу.

go() Переход к ранее открывавшемуся URL-адресу.

Описание

Объект History изначально предусматривался для представления истории посещений окна. Однако по соображениям безопасности объект History больше не позволяет получать из сценариев доступ к хранящимся в нем URL-адресам. Единственное, что им доступно, – это функциональность объекта History в виде методов back(), forward() и go().

Пример

Следующая строка выполняет то же действие, что и щелчок на кнопке Назад броузера:

```
history.back();
```

Следующий код выполяет то же действие, что и два щелчка на кнопке Назад:

```
history.go(-2);
```

См. также Свойство `history` объекта `Window`, `Location`

History.back() JavaScript 1.0

возврат к предыдущему URL-адресу

Синтаксис

```
history.back()
```

Описание

В результате вызова метода `back()` окно или фрейм, которому принадлежит объект `History`, заново открывает URL-адрес (если он есть), открытый непосредственно перед текущим. Вызов этого метода имеет тот же эффект, что и щелчок на кнопке Назад в броузере. Он также эквивалентен инструкции:

```
history.go(-1);
```

History.forward() JavaScript 1.0

переход к следующему URL-адресу

Синтаксис

```
history.forward()
```

Описание

В результате вызова метода `forward()` окно или фрейм, которому принадлежит объект `History`, заново открывает URL-адрес (если он есть), открытый непосредственно после текущего. Вызов этого метода имеет тот же эффект, что и щелчок на кнопке Вперед в броузере. Он также эквивалентен инструкции:

```
history.go(1);
```

Обратите внимание: если пользователь не перемещался назад по списку истории просмотра при помощи кнопки Назад или меню Журнал, а JavaScript-код не вызывал методы `History.back()` или `History.go()`, то метод `forward()` не действует, т. к. броузер уже находится в конце своего списка URL-адресов и нет адреса, к которому можно было бы перейти.

History.go() JavaScript 1.0

повторно открывает URL-адрес

Синтаксис

```
history.go(относительная_позиция)
history.go(целевая_строка)
```

Аргументы

относительная_позиция

Относительная позиция в журнале ранее посещавшихся URL-адресов, которую следует открыть.

целевая_строка

URL-адрес (или его фрагмент), который должен быть открыт.

Описание

Первая форма метода `History.go()` принимает целый аргумент и заставляет броузер открыть URL-адрес, который находится на указанном расстоянии в списке истории, хранящемся в объекте `History`. Положительные аргументы перемещают броузер вперед по списку, а отрицательные – назад. Другими словами, вызов `history.go(-1)` эквивалентен вызову `history.back()` и действует так же, как щелчок на кнопке Назад. Аналогично `history.go(3)` заново открывает тот же URL-адрес, который был бы открыт при трехкратном вызове `history.forward()`.

Вторая форма метода `History.go()` принимает строковый аргумент и приводит к тому, что броузер повторно открывает первый (т. е. открытый последним) URL-адрес, содержащий указанную строку. Эта форма обращения к методу плохо документирована и в разных броузерах может работать по-разному. Например в документации Microsoft указывается, что аргумент *целевая_строка* должен точно совпадать с ранее посещавшимся URL-адресом, тогда как в старой документации Netscape (создателем объекта `History` является компания Netscape) говорится, что аргумент может содержать лишь фрагмент ранее посещавшегося URL-адреса.

HTMLCollection

DOM Level 2 HTML

массив HTML-элементов, доступных по позиции или по имени **Object→HTMLCollection**

Свойства

`readonly unsigned long length`

Количество элементов в коллекции.

Методы

`item()`

Возвращает элемент коллекции, расположенный в указанной позиции. Можно не вызывать этот метод явно, а задать номер в квадратных скобках (как для массива).

`namedItem()`

Возвращает элемент из коллекции, имеющий указанное значение атрибута `id` или `name`, либо `null`, если такого элемента нет. Можно также поместить имя элемента в квадратные скобки (как для массива), а не вызывать этот метод явно.

Описание

Объект `HTMLCollection` – это коллекция HTML-элементов с методами, позволяющими извлекать элементы либо по их позиции в документе, либо по их атрибуту `id` или `name`. JavaScript-объекты `HTMLCollection` ведут себя как доступные только для чтения массивы, и для индексирования объекта `HTMLCollection` по номеру или по имени можно использовать нотацию JavaScript с квадратными скобками, а не вызывать методы `item()` и `namedItem()`.

Многие свойства объекта HTMLDocument – это объекты HTMLCollection, предоставляющие удобный доступ к таким элементам документа, как формы, изображения и ссылки. Свойства Form.elements и Select.options – это объекты HTMLCollection. Кроме того, объект HTMLCollection предоставляет удобное средство обхода строк HTML-таблиц или ячеек таблиц внутри строки таблицы.

Объекты HTMLCollection доступны только для чтения – им нельзя присваивать новые элементы даже при использовании нотации JavaScript-массивов. Они «живые», т. е. когда исходный документ изменяется, эти изменения сразу же становятся видимыми через объекты HTMLCollection.

Объекты HTMLCollection напоминают объекты NodeList, но могут индексироваться как именами, так и числовыми значениями.

Пример

```
var c = document.forms;       // Это HTML-коллекция элементов формы
var firstform = c[0];         // Может использоваться как обычный массив
var lastform = c[c.length-1]; // Свойство length содержит количество элементов
var address = c["address"];   // Может использоваться как ассоциативный массив
var address = c.address;      // JavaScript допускает и такую нотацию
```

См. также HTMLDocument, NodeList

HTMLCollection.item() DOM Level 2 HTML

возвращает элемент по его позиции в массиве

Синтаксис

```
Node item(unsigned long индекс);
```

Аргументы

индекс

Позиция возвращаемого элемента. Элементы в HTMLCollection расположены в том же порядке, что и в исходном тексте документа.

Возвращаемое значение

Элемент в позиции *индекс* или null, если индекс меньше нуля либо больше или равен значению свойства length.

Описание

Метод item() возвращает элемент с указанным номером из HTMLCollection. В JavaScript проще рассматривать HTMLCollection как массив и обращаться к нему с помощью нотации массива.

Пример

```
var c = document.images; // Это - HTMLCollection
var img0 = c.item(0);    // Можно использовать метод item() так
var img1 = c[1];         // Но эта нотация проще и используется чаще
```

См. также NodeList.item()

HTMLCollection.namedItem() DOM Level 2 HTML

возвращает элемент по его имени

Синтаксис

```
Node namedItem(String имя);
```

Аргументы

имя Имя возвращаемого элемента.

Возвращаемое значение

Элемент с указанным значением атрибута `id` или `name`, либо `null`, если ни один из элементов `HTMLCollection` не имеет такого имени.

Описание

Этот метод отыскивает и возвращает элемент из `HTMLCollection`, имеющий указанное имя. Если какой-либо элемент имеет атрибут `id`, значение которого равно указанному имени, то возвращается этот элемент. Если такой элемент не найден, возвращается элемент, атрибут `name` которого равен указанному значению. Если такого элемента не существует, `namedItem()` возвращает `null`.

Обратите внимание: атрибут `id` может быть задан для любого HTML-элемента, но только определенные HTML-элементы, такие как формы, элементы формы, изображения и якоря, могут иметь атрибут `name`.

В JavaScript проще рассматривать объект `HTMLCollection` как ассоциативный массив и задавать имя между квадратными скобками, используя нотацию массива.

Пример

```
var forms = document.forms;                // HTML-коллекция форм
var address = forms.namedItem("address"); // Найти <form name="address">
var payment = forms["payment"]             // Проще: находит тег
                                           // <form name="payment">
var login = forms.login; // Тоже работает: находит тег <form name="login">
```

HTMLDocument DOM Level 0

корень дерева HTML-документа **Node→Document→HTMLDocument**

Свойства

`Element[] all` ***[IE4]***

Это нестандартное свойство представляет объект, подобный массиву, с помощью которого можно получить доступ к любому элементу в документе. Массив `all[]` впервые появился в IE 4, и хотя он был заменен такими методами, как `Document.getElementById()` и `Document.getElementsByTagName()`, он все еще используется в распространяемых сценариях. Подробнее об этом массиве см. в справочной статье об элементе `HTMLDocument.all[]`.

`readonly HTMLCollection anchors`

Массив (`HTMLCollection`) всех якорных элементов в документе.

`readonly HTMLCollection applets`

Массив (`HTMLCollection`) всех апплетов в документе.

`HTMLElement body`

Вспомогательное свойство, которое ссылается на объект `HTMLBodyElement`, представляющий тег `<body>` этого документа. Для документов, определяющих наборы фреймов, это свойство ссылается на самый внешний тег `<frameset>`.

`String cookie`

Позволяет получать и устанавливать свойство `cookie` для этого документа. Подробности см. в справочной статье о свойстве `HTMLDocument.cookie`.

`readonly String domain`

Имя домена сервера, откуда был загружен документ, или `null`, если он отсутствует. В отдельных случаях это свойство может использоваться для ослабления требований политики общего происхождения. Подробности см. в справочной статье о свойстве `HTMLDocument.domain`.

`readonly HTMLCollection forms`

Массив (`HTMLCollection`) всех объектов `Form` в документе.

`readonly HTMLCollection images`

Массив (`HTMLCollection`) всех объектов `Image` в документе. Обратите внимание: для совместимости с DOM Level 0 изображения, определенные с помощью тега `<object>`, не включаются в эту коллекцию.

`readonly HTMLCollection lastModified`

Содержит дату и время последнего изменения документа. Эти значения поставляются в HTTP-заголовке `Last-Modified`, который может передаваться веб-сервером по требованию.

`readonly HTMLCollection links`

Массив (`HTMLCollection`) всех объектов `Link` в документе.

`readonly String referrer`

URL-адрес, ссылающийся на данный документ, или `null`, если документ был открыт не через гиперссылку. Это свойство позволяет получить доступ к HTTP-заголовку `referer`. Обратите внимание на правописание: имя заголовка включает три символа «r», а имя JavaScript-свойства – четыре.

`String title`

Содержимое тега `<title>` данного документа.

`readonly String URL`

URL-адрес документа. Это значение часто совпадает со значением свойства `location.href` объекта `Window`, содержащего документ. Однако в случае перенаправления свойство `URL` будет хранить фактический URL-адрес документа, тогда как `location.href` – запрошенный URL-адрес.

Методы

`close()`

Закрывает поток вывода документа, открытый методом `open()`, заставляя вывести любые буферизованные данные.

`getElementsByName()`

Возвращает массив узлов (`NodeList`) всех элементов в документе, значение атрибута `name` которых равно указанному.

`open()`

Открывает поток вывода, в который может быть записано содержимое нового документа. Обратите внимание: этот метод удаляет любое текущее содержимое документа.

`write()`

Дописывает строку HTML-текста в открытый документ.

`writeln()`

Дописывает строку HTML-текста и символ перевода строки в открытый документ.

Описание

Этот интерфейс расширяет интерфейс `Document` и определяет специфические для HTML свойства и методы. Значительная часть свойств представляют собой объекты `HTMLCollection` (по сути массивы, доступные только для чтения, которые могут индексироваться числами и именами), хранящие ссылки на якоря, формы, гиперссылки и другие важные элементы документа, доступные из сценариев. Эти свойства-коллекции появились с введением модели DOM Level 0. Они были заменены методом `Document.getElementsByTagName()`, но благодаря удобству продолжают широко использоваться.

Особо следует отметить метод `write()`, который позволяет сценариям вставлять в документ динамически сгенерированное содержимое в процессе загрузки и анализа документа.

Обратите внимание: в модели DOM Level 1 интерфейс `HTMLDocument` определяет метод с именем `getElementById()`. В DOM Level 2 этот метод был перемещен в интерфейс `Document` и теперь наследуется интерфейсом `HTMLDocument`, а не определяется в нем. Подробности см. в справочной статье о методе `Document.getElementById()`.

См. также `Document`, `Document.getElementById()`, `Document.getElementsByTagName()`

HTMLDocument.all[] IE4

все HTML-элементы в документе

Синтаксис

document.all[i]
document.all[*имя*]
document.all.tags(*имяТега*)

Описание

Объект `all[]` – это универсальный объект, подобный массиву. Он предоставляет доступ ко всем HTML-элементам в документе. Впервые массив `all[]` появился в IE 4, после чего был заимствован некоторыми другими броузерами. Ему на смену пришли стандартные методы `getElementById()` и `getElementsByTagName()` интерфейса `Document`, а также стандартный метод `getElementsByName()` интерфейса `HTMLDocument`. Но несмотря на это, массив `all[]` продолжает использоваться в существующих сценариях.

Элементы расположены в массиве `all[]` в том же порядке, в котором они находятся в исходном коде документа. Если порядок следования элементов точно известен, можно получать доступ к этим элементам по числовым индексам. Однако чаще всего

массив all[] служит для доступа к элементам по значениям атрибутов id или name. Если в документе существует более одного элемента с искомым именем, тогда при использовании такого имени свойство all[] вернет массив элементов с этим именем.

Метод all.tags() принимает имя тега и возвращает массив HTML-элементов указанного типа.

См. также Document.getElementById(), Document.getElementsByTagName(), HTMLElement

HTMLDocument.close() DOM Level 0

закрывает открытый документ и отображает его

Синтаксис

```
void close();
```

Описание

Этот метод закрывает поток вывода документа, открытый методом open(), и заставляет вывести все буферизованные данные. После вывода динамического содержимого документа методом write() не следует забывать вызывать метод close(), чтобы обеспечить вывод всего содержимого. После вызова close() вызывать метод write() не нужно, т. к. он неявно вызывает метод open(), стирающий существующий документ и создающий новый.

См. также HTMLDocument.open(), HTMLDocument.write()

HTMLDocument.cookie DOM Level 0

cookie-файлы документа

Синтаксис

```
String cookie
```

Описание

Свойство cookie является строковым, оно позволяет читать, создавать, изменять и удалять cookie-файлы документа. *Cookie*– это небольшая порция именованных данных, сохраняемых веб-броузером. Cookie-файлы дают возможность броузеру «запоминать» данные, введенные на одной странице, и воспроизводить их на другой странице или восстанавливать заданные пользователем параметры при следующих посещениях страницы. Данные, содержащиеся в cookie, в случае необходимости автоматически передаются между веб-броузером и веб-сервером, что позволяет серверному сценарию читать и изменять хранящиеся в нем значения. Клиентский JavaScript-сценарий также может читать и изменять содержимое cookie-файлов с помощью этого свойства.

Свойство HTMLDocument.cookie ведет себя не так, как обычное свойство, доступное для чтения и записи. Хотя прочитать и записать значение в HTMLDocument.cookie можно, значение, прочитанное из этого свойства, в общем случае не будет совпадать с записанным. За дополнительной информацией по использованию этого особенно сложного свойства обращайтесь к главе 19.

Порядок использования

Cookies предназначены для сохранения небольших объемов данных. Поскольку они задумывались совсем не как общецелевой механизм обмена данными или программирования, при их использовании следует проявлять умеренность. Обратите внимание: веб-броузеры не обязаны хранить более 20 cookies на каждый веб-сервер, а также cookie, состоящие из пар *имя-значение*, длиной более 4 Кбайт.

См. также Глава 19

HTMLDocument.domain DOM Level 0

домен безопасности документа

Синтаксис

```
String domain
```

Описание

В соответствии со стандартом DOM Level 2 для модуля `HTML` свойство `domain` — это обычная строка, доступная только для чтения и содержащая имя хоста веб-сервера, с которого был загружен документ.

Данное свойство имеет еще одно важное предназначение (хотя и не стандартизованное). Политика общего происхождения (см. раздел 13.8.2) не позволяет JavaScript-сценариям, находящимся в одном документе, читать содержимое другого документа (например, документа, отображаемого в теге `<iframe>`), если оба документа имеют разное происхождение (т. е. были получены с разных веб-серверов). На крупных веб-сайтах, которые располагаются на нескольких веб-серверах, это может вызывать определенные проблемы. Например, сценарию, полученному с хоста *www.oreilly.com*, может потребоваться прочитать содержимое документа, полученного с хоста *search.oreilly.com*.

Свойство `domain` помогает решать подобные проблемы. Это свойство можно изменить, правда, диапазон возможных изменений весьма ограничен: в него можно записать только суффикс самого домена. Например, сценарий, загруженный с сервера *search.oreilly.com*, может записать в свойство `domain` суффикс «oreilly.com». Если исполняющийся в другом окне сценарий, полученный с *www.oreilly.com*, и данный сценарий установят значение свойства `domain` равным `"oreilly.com"`, то каждый из них сможет читать содержимое документа другого сценария даже при том, что они получены с разных серверов. Обратите внимание: сценарий, полученный с сервера *search.oreilly.com*, не сможет записать в свойство `domain` строку `"search.oreilly"` или `".com"`.

См. также Раздел 13.8.2 «Политика общего происхождения»

HTMLDocument.getElementsByName() DOM Level 2 HTML

отыскивает элементы с указанным значением атрибута name

Синтаксис

```
Element[] getElementsByName(String имяЭлемента);
```

Аргументы

имяЭлемента

Желаемое значение атрибута name.

Возвращаемое значение

Массив (фактически – NodeList) объектов Element, имеющих атрибут name, значение которого равно указанному значению. Если такие элементы не обнаружатся, возвращаемый массив NodeList будет пуст и иметь нулевую длину.

Описание

Этот метод отыскивает в дереве HTML-документа узлы Element, имеющие атрибут name, значение которого равно указанному значению, и возвращает объект NodeList (его можно рассматривать как массив), содержащий все найденные элементы. Если элементы не найдены, возвращается объект NodeList со свойством length, равным 0.

Не путайте этот метод ни с методом Document.getElementById(), который находит единственный узел Element на основе уникального значения атрибута id, ни с методом Document.getElementsByTagName(), возвращающим NodeList элементов с указанным именем тега.

См. также Document.getElementById(), Document.getElementsByTagName()

HTMLDocument.open() DOM Level 0

начинает новый документ, стирая текущий

Синтаксис

```
void open();
```

Описание

Этот метод стирает текущий HTML-документ и начинает новый, в который можно писать с помощью методов write() и writeln(). После вызова метода open() для открытия нового документа и метода write() для задания содержимого документа необходимо вызвать метод close() для закрытия документа и вывода его содержимого.

Этот метод не должен вызываться сценарием или обработчиком событий, являющимся частью переписываемого документа, т. к. сам сценарий или обработчик также будет переписан.

Пример

```
var w = window.open("");          // Открывает новое окно
var d = w.document;               // Получает его объект HTMLDocument
d.open();                         // Открывает документ для записи
d.write("<h1>Hello world</h1>"); // Выводит в документ некоторый HTML-код
d.close();                        // Завершает и отображает документ
```

См. также HTMLDocument.close(), HTMLDocument.write()

HTMLDocument.write() DOM Level 0

дописывает HTML-текст в открытый документ

Синтаксис

```
void write(String текст);
```

Аргументы

текст

Дописываемый к документу HTML-текст.

Описание

Этот метод дописывает указанный *текст* к документу. В соответствии со стандартом DOM этот метод принимает только один строковый аргумент. Однако на практике метод `write()` может принимать произвольное число аргументов. Эти аргументы преобразуются в строки и по порядку добавляются в документ.

Обычно метод `Document.write()` используется одним из двух способов. Во-первых, метод может вызываться в текущем документе из сценария, находящегося внутри тега `<script>`, в процессе синтаксического разбора документа. В этом случае метод `write()` выведет HTML-текст так, как если бы он непосредственно присутствовал в файле, в месте, где находится программный код, вызвавший этот метод.

Во-вторых, метод `Document.write()` может использоваться для динамического создания нового документа в окне, фрейме или плавающем фрейме, отличном от того, в котором исполняется вызывающий сценарий. Если целевой документ уже был открыт, метод `write()` допишет содержимое аргументов в конец документа. Если документ не был открыт, `write()` сотрет существующий и откроет новый (пустой) документ, к которому добавит содержимое своих аргументов.

После открытия документа с помощью метода `Document.write()` можно добавлять в конец документа произвольные объемы информации. Когда новые документы полностью создаются таким способом, они должны быть закрыты методом `Document.close()`. Обратите внимание: несмотря на то что метод `open()` можно и не вызывать, метод `close()` нужно вызывать всегда.

Результат работы метода `Document.write()` может не сразу появиться в документе, если броузер будет буферизовать анализируемый текст и отображать его по частям. Вызов метода `Document.close()` – это единственный способ явно «вытолкнуть» все данные из буфера и отобразить их.

См. также `HTMLDocument.close()`, `HTMLDocument.open()`

HTMLDocument.writeln() DOM Level 0

дописывает HTML-текст и символ перевода строки в открытый документ

Синтаксис

```
void writeln(String текст);
```

Аргументы

text

HTML-текст, дописываемый к документу.

Описание

Этот метод похож на `HTMLDocument.write()`, за исключением того, что за дописываемым текстом следует символ перевода строки, что может быть удобным, например при формировании содержимого тега `<pre>`.

См. также `HTMLDocument.write()`

HTMLElement

DOM Level 2 HTML

элемент HTML-документа **Node→Element→HTMLElement**

Свойства

Каждый элемент HTML-документа обладает свойствами, соответствующими атрибутам HTML-элемента. Здесь перечислены свойства, поддерживаемые всеми HTML-тегами. Другие свойства, характерные для отдельных типов HTML-тегов, перечислены в таблице подраздела «Описание». Объекты `HTMLElement` наследуют большое число стандартных свойств от интерфейсов `Node` и `Element` и дополнительно реализуют некоторые нестандартные свойства, описываемые далее:

`String className`

Значение атрибута `class` элемента, задающее ноль или более имен CSS-классов, разделенных символом пробела. Обратите внимание: это свойство не получило имя «class», т. к. это имя в JavaScript зарезервировано.

`CSS2Properties currentStyle`

Это свойство, реализованное только в IE, является представлением каскадного набора всех CSS-свойств, применяемых к элементу. Данное свойство представляет собой альтернативу методу `Window.getComputedStyle()`.

`String dir`

Значение атрибута `dir` элемента, задающее направление текста в документе.

`String id`

Значение атрибута `id`. Никакие два элемента в одном документе не должны иметь одинаковые значения атрибута `id`.

`String innerHTML`

Доступная для чтения и записи строка, которая определяет HTML-текст, содержащийся внутри элемента, за исключением открывающего и закрывающего тегов самого элемента. Операция чтения этого свойства возвращает содержимое элемента в виде строки HTML-текста. Операция записи замещает содержимое элемента представлением HTML-текста после его синтаксического разбора. Данное свойство нельзя изменить в процессе загрузки документа (для этих целей существует метод `HTMLDocument.write()`). Это нестандартное свойство изначально появилось в IE 4. Ныне оно реализовано во всех современных броузерах.

`String lang`

Значение атрибута `lang`, задающее код языка для содержимого элемента.

`int offsetHeight, offsetWidth`

Высота и ширина элемента и всего его содержимого в пикселах, включая отступы и рамки, но без учета полей. Это нестандартные, но широко поддерживаемые свойства.

`int offsetLeft, offsetTop`

Координаты X и Y верхнего левого угла CSS-рамки элемента относительно контейнерного элемента `offsetParent`. Это нестандартные, но широко поддерживаемые свойства.

`Element offsetParent`

Указывает на контейнерный элемент, определяющий систему координат, относительно которой измеряются свойства `offsetLeft` и `offsetTop`. Для большинства эле-

ментов свойство offsetParent ссылается на вмещающий их объект Document. Однако если контейнерный элемент имеет динамическое позиционирование, ссылка на него становится значением свойства offsetParent динамически позиционируемого элемента. В некоторых броузерах ячейки таблицы позиционируются относительно элемента строки, в который они помещаются, а не относительно документа. Пример использования этого свойства переносимым образом вы найдете в главе 16. Это нестандартное, но широко поддерживаемое свойство.

`int scrollHeight, scrollWidth`

Общая высота и ширина элемента в пикселах. Когда элемент имеет полосы прокрутки (например, потому что был установлен CSS-атрибут overflow), значения этих свойств отличаются от значений свойств offsetHeight и offsetWidth, которые просто содержат размеры видимой части элемента. Это нестандартные, но широко поддерживаемые свойства.

`int scrollLeft, scrollTop`

Число пикселов, на которое элемент был прокручен за левую или верхнюю границу. Обычно эти свойства полезны только для элементов с полосами прокрутки, у которых, например, CSS-атрибут overflow имеет значение auto. Эти свойства определены также в теге <body> или <html> документа (в зависимости от броузера), задавая общую величину прокрутки всего документа. Обратите внимание: эти свойства не определяют величину прокрутки в теге <iframe>. Это нестандартные, но широко поддерживаемые свойства.

`CSS2Properties style`

Значение атрибута style, задающее встроенные CSS-стили для элемента. Обратите внимание: значение этого свойства не является строкой. Подробности см. в справочной статье об объекте CSS2Properties.

`String title`

Значение атрибута title, задающее описательный текст для вывода в качестве всплывающей подсказки элемента.

Методы

Объекты HTMLElement наследуют стандартные методы интерфейсов Node и Element. Определенные типы элементов реализуют методы, характерные для отдельных типов HTML-тегов. Эти теги перечислены в таблице подраздела «Описание», а также упоминаются в других справочных статьях, например, в статьях об объектах Form, Input и Table. Большинство современных броузеров дополнительно реализуют следующий нестандартный метод:

`scrollIntoView()`

Метод прокручивает документ так, чтобы элемент стал видимым в верхней или нижней части окна.

Обработчики событий

Все HTML-элементы откликаются на события от мыши и клавиатуры и могут вызывать перечисленные далее обработчики событий. Некоторые элементы, такие как ссылки и кнопки, выполняют действия, предлагаемые по умолчанию при возникновении этих событий. Для таких элементов приводятся дополнительные сведения в соответствующих им справочных статьях (см. например, описание объектов Input и Link):

`onclick`

Вызывается, когда пользователь щелкает мышью на элементе.

`ondblclick`

Вызывается, когда пользователь выполняет двойной щелчок мышью на элементе.

`onkeydown`

Вызывается, когда пользователь нажимает клавишу.

`onkeypress`

Вызывается, когда пользователь нажимает и отпускает клавишу.

`onkeyup`

Вызывается, когда пользователь отпускает клавишу.

`onmousedown`

Вызывается, когда пользователь нажимает кнопку мыши.

`onmousemove`

Вызывается, когда пользователь перемещает указатель мыши.

`onmouseout`

Вызывается, когда пользователь перемещает указатель мыши за пределы элемента.

`onmouseover`

Вызывается, когда пользователь наводит указатель мыши на элемент.

`onmouseup`

Вызывается, когда пользователь отпускает кнопку мыши.

Описание

Каждый элемент HTML-документа представлен своим объектом `HTMLElement`, который определяет свойства, представляющие атрибуты, общие для всех HTML-элементов. Перечисленные далее теги не имеют дополнительных свойств, кроме представленных ранее, и полностью описываются интерфейсом `HTMLElement`:

`<abbr>`	`<acronym>`	`<address>`	`<b>`
`<bdo>`	`<big>`	`<center>`	`<cite>`
`<code>`	`<dd>`	`<dfn>`	`<dt>`
`<em>`	`<i>`	`<kbd>`	`<noframes>`
`<noscript>`	`<s>`	`<samp>`	`<small>`
`<span>`	`<strike>`	`<strong>`	`<sub>`
`<sup>`	`<tt>`	`<u>`	`<var>`

Большинство HTML-тегов определяют дополнительные свойства, которые не были перечислены выше. Спецификация DOM Level 2 HTML определяет для таких тегов специфические интерфейсы, чтобы всем стандартным HTML-атрибутам соответствовали JavaScript-свойства. Как правило, тегу с именем *T* соответствует специальный интерфейс `HTMLTElement`. Например, тег `<head>` представлен интерфейсом `HTMLHeadElement`. В некоторых случаях два или более родственных тегов могут быть представлены единственным интерфейсом, как например в случае с тегами от `<h1>` до `<h6>`, которые все представлены интерфейсом `HTMLHeadingElement`.

Большинство этих специфических интерфейсов не делают ничего, кроме определения JavaScript-свойств для всех атрибутов HTML-тега. JavaScript-свойства имеют те же имена, что и атрибуты, и состоят из символов нижнего регистра (например, `id`)

или смешанного регистра, если имя атрибута состоит из нескольких слов (например, longDesc). Если имя HTML-атрибута является зарезервированным Java- или JavaScript-словом, то оно слегка меняется. Например, атрибут for тегов <label> и <script> становится свойством htmlFor интерфейсов HTMLLabelElement и HTMLScriptElement, т. к. for – это зарезервированное слово. Назначение этих свойств непосредственно соответствует HTML-атрибутам, определенным в спецификации HTML, и описание каждого из них выходит за рамки темы этой книги.

В следующей таблице перечислены все HTML-теги, для которых имеется соответствующий подынтерфейс HTMLElement. Для каждого тега в таблице указано имя DOM-интерфейса, а также имена определенных в нем свойств и методов. Все свойства являются доступными для чтения и записи строками, если не указано другое. Для свойств, не являющихся строками, доступными для чтения и записи, тип свойства указывается в квадратных скобках перед именем свойства. Достаточно много тегов и атрибутов в HTML 4 считаются устаревшими – они помечены символом звездочки (*).

Эти интерфейсы и их свойства настолько соответствуют HTML-элементам и атрибутам, что для большинства интерфейсов в этой книге нет отдельных справочных статей, и подробную информацию о них следует искать в литературе по HTML. Исключение составляют интерфейсы, в которых определены методы, а также интерфейсы, представляющие некоторые особенно важные теги, такие как <form> и <input>. Эти теги описываются в данной книге под именами, в которых отсутствуют префикс «HTML» и суффикс «Element». См. например, статьи об интерфейсах Anchor, Applet, Canvas, Form, Image, Input, Link, Option, Select, Table и Textarea.

HTML-тег	Интерфейс, свойства и методы модели DOM
Все теги	HTMLElement: id, title, lang, dir, className
<a>	HTMLAnchorElement: accessKey, charset, coords, href, hreflang, name, rel, rev, shape, [long] tabIndex, target, type, blur(), focus()
<applet>	HTMLAppletElement*: align*, alt*, archive*, code*, codeBase*, height*, hspace*, name*, object*, vspace*, width*
<area>	HTMLAreaElement: accessKey, alt, coords, href, [boolean] noHref, shape, [long] tabIndex, target
<base>	HTMLBaseElement: href, target
<basefont>	HTMLBaseFontElement*: color*, face*, size*
<blockquote>, <q>	HTMLQuoteElement: cite
<body>	HTMLBodyElement: aLink*, background*, bgColor*, link*, text*, vLink*
 	HTMLBRElement: clear*
<button>	HTMLButtonElement: [HTMLFormElement, только чтение] form, accessKey, [boolean] disabled, name, [long] tabIndex, [только чтение] type, value
<caption>	HTMLTableCaptionElement: align*
<col>, <colgroup>	HTMLTableColElement: align, ch, chOff, [long] span, vAlign, width
<del>, <ins>	HTMLModElement: cite, dateTime
<dir>	HTMLDirectoryElement*: [boolean] compact*
	HTMLDivElement: align*
<dl>	HTMLDListElement: [boolean] compact*
<fieldset>	HTMLFieldSetElement: [HTMLFormElement, только чтение] form

HTML-тег	Интерфейс, свойства и методы модели DOM
<font>	HTMLFontElement*: color*, face*, size*
<form>	HTMLFormElement: [HTMLCollection, только чтение] elements, [long, только чтение] length, name, acceptCharset, action, enctype, method, target, submit(), reset()
<frame>	HTMLFrameElement: frameBorder, longDesc, marginHeight, marginWidth, name, [boolean] noResize, scrolling, src, [Document, только чтение] contentDocument
<frameset>	HTMLFrameSetElement: cols, rows
<h1>, <h2>, <h3>, <h4>, <h5>, <h6>	HTMLHeadingElement: align*
<head>	HTMLHeadElement: profile
<hr>	HTMLHRElement: align*, [boolean] noShade*, size*, width*
<html>	HTMLHtmlElement: version*
<iframe>	HTMLIFrameElement: align*, frameBorder, height, longDesc, marginHeight, marginWidth, name, scrolling, src, width, [Document, только чтение] contentDocument
<img>	HTMLImageElement: align*, alt, [long] border*, [long] height, [long] hspace*, [boolean] isMap, longDesc, name, src, useMap, [long] vspace*, [long] width
<input>	HTMLInputElement: defaultValue, [boolean] defaultChecked, [HTMLFormElement, только чтение] form, accept, accessKey, align*, alt, [boolean] checked, [boolean] disabled, [long] maxLength, name, [boolean] readOnly, size, src, [long] tabIndex, type, useMap, value, blur(), focus(), select(), click()
<ins>	См. <del>
<isindex>	HTMLIsIndexElement*: [HTMLFormElement, только чтение] form, prompt*
<label>	HTMLLabelElement: [HTMLFormElement, только чтение] form, accessKey, htmlFor
<legend>	HTMLLegendElement: [HTMLFormElement, только чтение] form, accessKey, align*
	HTMLLIElement: type*, [long] value*
<link>	HTMLLinkElement: [boolean] disabled, charset, href, hreflang, media, rel, rev, target, type
<map>	HTMLMapElement: [массив HTMLCollection элементов HTMLAreaElement, только чтение] areas, name
<menu>	HTMLMenuElement*: [boolean] compact*
<meta>	HTMLMetaElement: content, httpEquiv, name, scheme
<object>	HTMLObjectElement: code, align*, archive, border*, codeBase, codeType, data, [boolean] declare, height, hspace*, name, standby, [long] tabIndex, type, useMap, vspace*, width, [Document, только чтение] contentDocument
	HTMLOListElement: [boolean] compact*, [long] start*, type*
<optgroup>	HTMLOptGroupElement: [boolean] disabled, label
<option>	HTMLOptionElement: [HTMLFormElement, только чтение] form, [boolean] defaultSelected, [только чтение] text, [long, только чтение] index, [boolean] disabled, label, [boolean] selected, value
<p>	HTMLParagraphElement: align*
<param>	HTMLParamElement: name, type, value, valueType

HTML-тег	Интерфейс, свойства и методы модели DOM
<pre>	HTMLPreElement: [long] width*
<q>	См. <blockquote>
<script>	HTMLScriptElement: text, htmlFor, event, charset, [boolean] defer, src, type
<select>	HTMLSelectElement: [только чтение] type, [long] selectedIndex, value, [long, только чтение] length, [HTMLFormElement, только чтение] form, [массив HTMLCollection элементов HTMLOptionElement, только чтение] options, [boolean] disabled, [boolean] multiple, name, [long] size, [long] tabIndex, add(), remove(), blur(), focus()
<style>	HTMLStyleElement: [boolean] disabled, media, type
<table>	HTMLTableElement: [HTMLTableCaptionElement] caption, [HTMLTableSectionElement] tHead, [HTMLTableSectionElement] tFoot, [массив HTMLCollection элементов HTMLTableRowElement, только чтение] rows, [массив HTMLCollection элементов HTMLTableSectionElement, только чтение] tBodies, align*, bgColor*, border, cellPadding, cellSpacing, frame, rules, summary, width, createTHead(), deleteTHead(), createTFoot(), deleteTFoot(), createCaption(), deleteCaption(), insertRow(), deleteRow()
<tbody>, <tfoot>, <thead>	HTMLTableSectionElement: align, ch, chOff, vAlign, [массив HTMLCollection элементов HTMLTableRowElement, только чтение] rows, insertRow(), deleteRow()
<td>, <th>	HTMLTableCellElement: [long, только чтение] cellIndex, abbr, align, axis, bgColor*, ch, chOff, [long] colSpan, headers, height*, [boolean] noWrap*, [long] rowSpan, scope, vAlign, width*
<textarea>	HTMLTextAreaElement: defaultValue, [HTMLFormElement, только чтение] form, accessKey, [long] cols, [boolean] disabled, name, [boolean] readOnly, [long] rows, [long] tabIndex, [только чтение] type, value, blur(), focus(), select()
<tfoot>	См. <tbody>
<th>	См. <td>
<thead>	См. <tbody>
<title>	HTMLTitleElement: text
<tr>	HTMLTableRowElement: [long, только чтение] rowIndex, [long, только чтение] sectionRowIndex, [массив HTMLCollection элементов HTMLTableCellElement, только чтение] cells, align, bgColor*, ch, chOff, vAlign, insertCell(), deleteCell()
	HTMLUListElement: [boolean] compact*, type*

См. также

Anchor, Element, Form, HTMLDocument, Image, Input, Link, Node, Option, Select, Table, TableCell, TableRow, TableSection, Textarea; **глава 15**

HTMLElement.onclick

DOM Level 0

обработчик события, который вызывается щелчком мыши на элементе

Синтаксис

Function onclick

Описание

Свойство onclick объекта HTMLElement определяет функцию-обработчик события, которая вызывается, когда пользователь щелкает мышью на элементе. Обратите внима-

ние: onclick и onmusedown — это разные обработчики. Событие щелчка мыши не может произойти без предшествующих ему событий нажатия и отпускания кнопки мыши.

См. также Element.addEventListener(), Event, MouseEvent; глава 17

HTMLElement.ondblclick DOM Level 0

обработчик события, который вызывается после двойного щелчка мыши на элементе

Синтаксис

```
Function ondblclick
```

Описание

Свойство ondblclick объекта HTMLElement определяет функцию-обработчик события, которая вызывается, когда пользователь дважды щелкает мышью на элементе.

См. также Element.addEventListener(), Event, MouseEvent; глава 17

HTMLElement.onkeydown DOM Level 0

обработчик события, который вызывается, когда пользователь нажимает клавишу

Синтаксис

```
Function onkeydown
```

Описание

Свойство onkeydown объекта HTMLElement определяет функцию-обработчик события, которая вызывается, когда пользователь нажимает клавишу, в то время как элемент обладает фокусом ввода. Порядок определения того, какая клавиша была нажата, зависит от типа броузера. За дополнительной информацией обращайтесь к главе 17.

Обработчик события onkeydown обычно предпочтительнее задействовать для обработки нажатий функциональных клавиш, а для обработки нажатий обычных алфавитно-цифровых клавиш лучше использовать обработчик onkeypress.

См. также HTMLElement.onkeypress; глава 17

HTMLElement.onkeypress DOM Level 0

обработчик события, который вызывается, когда пользователь нажимает клавишу

Синтаксис

```
Function onkeypress
```

Описание

Свойство onkeypress объекта HTMLElement определяет функцию-обработчик события, которая вызывается, когда пользователь нажимает и отпускает клавишу, в то время как элемент обладает фокусом ввода. Событие keypress возникает после события keydown и перед парным ему событием keyup. События keydown и keyup очень похожи на событие keypress, хотя для обработки нажатий алфавитно-цифровых клавиш удобнее использовать событие keypress, а функциональных — keydown.

Определить, какая клавиша была нажата и какая клавиша-модификатор при этом удерживалась, достаточно сложно, и сам порядок выявления нажатой клавиши зависит от типа броузера. За дополнительной информацией обращайтесь к главе 17.

См. также `HTMLElement.onkeydown`; глава 17

HTMLElement.onkeyup **DOM Level 0**

обработчик события, который вызывается, когда пользователь отпускает клавишу

Синтаксис

```
Function onkeyup
```

Описание

Свойство `onkeyup` объекта `HTMLElement` определяет функцию-обработчик события, которая вызывается, когда пользователь отпускает клавишу, в то время как элемент обладает фокусом ввода.

См. также `HTMLElement.onkeydown`; глава 17

HTMLElement.onmousedown **DOM Level 0**

обработчик события, который вызывается, когда пользователь нажимает кнопку мыши

Синтаксис

```
Function onmousedown
```

Описание

Свойство `onmousedown` объекта `HTMLElement` определяет функцию-обработчик события, которая вызывается, когда пользователь нажимает кнопку мыши на элементе.

См. также `Element.addEventListener()`, `Event`, `MouseEvent`; глава 17

HTMLElement.onmousemove **DOM Level 0**

обработчик события, вызываемый при перемещении указателя мыши внутрь элемента

Синтаксис

```
Function onmousemove
```

Описание

Свойство `onmousemove` объекта `HTMLElement` определяет функцию-обработчик события, которая вызывается, когда пользователь перемещает указатель мыши внутрь элемента.

Если определить обработчик события `onmousemove`, то при перемещении указателя мыши внутрь элемента события перемещения мыши будут генерироваться в огромных количествах. Имейте это в виду при написании функции, которая должна вызываться этим обработчиком. Если имеется необходимость отслеживать перемещение мыши для реализации алгоритма перетаскивания элементов, лучше всего регистрировать нужный обработчик по событию `mousedown` и отменять регистрацию по событию `mouseup`.

См. также Element.addEventListener(), Event, MouseEvent; глава 17

HTMLElement.onmouseout DOM Level 0

обработчик события, вызываемый при перемещении указателя мыши за пределы элемента

Синтаксис

```
Function onmouseout
```

Описание

Свойство onmouseout объекта HTMLElement определяет функцию-обработчик события, которая вызывается, когда пользователь перемещает указатель мыши за пределы элемента.

См. также Element.addEventListener(), Event, MouseEvent; глава 17

HTMLElement.onmouseover DOM Level 0

обработчик события, вызываемый при наведении указателя мыши на элемент

Синтаксис

```
Function onmouseover
```

Описание

Свойство onmouseover объекта HTMLElement определяет функцию-обработчик события, которая вызывается, когда пользователь наводит указатель мыши на элемент.

См. также Element.addEventListener(), Event, MouseEvent; глава 17

HTMLElement.onmouseup DOM Level 0

обработчик события, вызываемый при отпускании кнопки мыши

Синтаксис

```
Function onmouseup
```

Описание

Свойство onmouseup объекта HTMLElement определяет функцию-обработчик события, которая вызывается, когда пользователь отпускает кнопку мыши на элементе.

См. также Element.addEventListener(), Event, MouseEvent; глава 17

HTMLElement.scrollIntoView() Firefox 1.0, IE 4, Safari 2.02, Opera 8.5

делает элемент видимым

Синтаксис

элемент.scrollIntoView(*top*)

Аргументы

top

Необязательный логический аргумент, указывающий, должен ли элемент прокручиваться к верхнему или нижнему краю экрана. Если он равен true или опу-

щен, элемент перемещается к верхнему краю экрана. Если он равен `false`, элемент перемещается к нижнему краю экрана. Данный аргумент поддерживается не всеми броузерами, и элементы, расположенные у верхнего или нижнего края окна, не могут быть прокручены к противоположному краю, т. е. этот аргумент следует рассматривать лишь как рекомендацию.

Описание

Если HTML-элемент находится за пределами области видимости, данный метод прокручивает документ так, чтобы элемент стал видимым. Аргумент *top* является необязательным и служит для броузера рекомендацией, указывающей, возле какого края окна, верхнего или нижнего, должен расположиться элемент в результате прокрутки. Если элемент, такой как `Link` или `Input`, способен принимать фокус ввода, вызов метода `focus()` неявно выполняет ту же операцию прокрутки.

См. также `Anchor.focus()`, `Input.focus()`, `Link.focus()`, `Window.scrollTo()`

IFrame DOM Level 2 HTML

тег <iframe> HTML-документа **Node→Element→HTMLElement→IFrame**

Свойства

Как показано далее, плавающие фреймы могут быть доступны в качестве объектов `IFrame` или `Window`. Когда доступ к плавающим фреймам производится как к объектам `IFrame`, они наследуют свойства `HTMLElement` и определяют следующие дополнительные свойства:

`Document contentDocument`

Документ, в котором находится содержимое `<iframe>`.

`String src`

URL-адрес, с которого было загружено содержимое плавающего фрейма. Установка этого свойства вызывает загрузку в плавающий фрейм нового документа. Данное свойство является простым отражением атрибута `src` тега `<iframe>`.

Помимо этих свойств, объект `IFrame` определяет следующие свойства, которые непосредственно соответствуют HTML-атрибутам:

Свойство	Атрибут	Описание
устаревшее `String align`	`align`	Выравнивание относительно содержимого
`String frameBorder`	`frameborder`	Содержит строку `"0"` для фреймов без рамки
`String height`	`height`	Высота области видимости в пикселах
`String longDesc`	`longdesc`	URL-адрес описания фрейма
`String marginHeight`	`marginheight`	Верхнее и нижнее поля фрейма
`String marginWidth`	`marginwidth`	Левое и правое поля фрейма
`String name`	`name`	Имя фрейма для выполнения поиска средствами DOM Level 0 и указания в атрибуте `target` форм
`String scrolling`	`scrolling`	Политика прокрутки фрейма: `"auto"`, `"yes"` или `"no"`
`String width`	`width`	Ширина области видимости в пикселах

Описание

За исключением некоторых незначительных отличий в атрибутах, в клиентском JavaScript-коде элементы <iframe> очень похожи на элементы <frame>. Элементы <iframe> становятся также частью массива frames[] вмещающего окна. При обращении к ним с помощью этого массива они представляются как объекты Window, поэтому перечисленные ранее свойства становятся недоступными. Когда доступ к элементу <iframe> организован по значению атрибута id или по имени тега, он представляется объектом IFrame, обладающим вышеперечисленными свойствами. В этом случае для доступа к содержимому плавающего фрейма следует использовать свойство contentDocument, а для получения URL-адреса данного документа или загрузки нового – свойство src. Обратите внимание: доступ к свойству contentDocument может ограничиваться политикой общего происхождения (см. раздел 13.8.2).

См. также

Frame, Window; глава 14

Image

DOM Level 2 HTML

изображение в HTML-документе **Node→Element→HTMLElement→Image**

Конструктор

new Image(integer *ширина*, integer *высота*)

Аргументы

ширина, *высота*

Ширина и высота изображения; могут не задаваться.

Свойства

String name

Это свойство определяет имя объекта изображения. Когда тег <img> имеет атрибут name, обратиться к соответствующему ему объекту Image можно как к одноименному свойству объекта Document.

String src

Доступная для чтения и для записи строка, указывающая URL-адрес изображения, отображаемого в броузере. Начальное значение этого свойства задается атрибутом src тега <img>. Если установить это свойство равным URL-адресу нового изображения, броузер загрузит и отобразит новое изображение. Это свойство может применяться для изменения внешнего вида страниц в ответ на действия пользователя, а также для создания простой анимации.

В дополнение к этим свойствам объекты Image поддерживают следующие свойства, которые просто отражают HTML-атрибуты:

Свойство	Атрибут	Описание
String align (устаревшее)	align	Выравнивание относительно содержимого
String alt	alt	Текст, который выводится при невозможности отобразить изображение
String border (устаревшее)	border	Толщина рамки, окружающей изображение
long height	height	Высота изображения в пикселах
long hspace (устаревшее)	hspace	Ширина левого и правого полей в пикселах

Свойство	Атрибут	Описание
boolean isMap	ismap	Признак использования карты изображений на стороне сервера
String longDesc	longdesc	URI подробного описания изображения
String useMap	usemap	Определяет карту изображений на стороне клиента
long vspace (устаревшее)	vspace	Ширина верхнего и нижнего полей в пикселах
long width	width	Ширина изображения в пикселах

Обработчики событий

Изображение наследует обработчики событий от объекта HTMLElement и дополнительно определяет следующие обработчики:

onabort — Вызывается, если пользователь прерывает загрузку изображения.

onerror — Вызывается, если происходит ошибка при загрузке изображения.

onload — Вызывается при успешном завершении загрузки изображения.

Синтаксис HTML

Объект Image создается с помощью стандартного HTML-тега <img>. Некоторые атрибуты <img> в приведенном синтаксисе отсутствуют, т. к. в JavaScript они недоступны:

```
<img src="url"           // Отображаемое изображение
width="пикселы"          // Ширина изображения
height="пикселы"         // Высота изображения
alt="описание"           // Краткое описание изображения
[ onload="обработчик" ]  // Вызывается, когда изображение полностью загружено
[ onerror="обработчик" ] // Вызывается в случае ошибки загрузки
[ onabort="обработчик" ] // Вызывается, если пользователь прерывает загрузку
>
```

Описание

Объекты Image встраиваются в HTML-документ в виде тегов <img>. Изображения, присутствующие в документе, собираются в виде массива document.images[]. Изображения, у которых установлен атрибут name, доступны также в виде одноименных свойств объекта Document, например:

```
document.images[0] // Первое изображение в документе
document.banner    // Изображение с атрибутом name="banner"
```

Свойство src объекта Image представляет наибольший интерес. Когда вы устанавливаете это свойство, броузер загружает и показывает изображение, заданное новым значением. Это позволяет создавать такие эффекты, как смена изображений и анимация. Соответствующие примеры приводятся в главе 22.

Объекты Image можно динамически создавать в JavaScript-коде с помощью функции-конструктора Image(). Обратите внимание: этот конструктор не имеет аргумента, задающего загружаемое изображение. Как и для изображений, созданных с помощью HTML-кода, можно заставить броузер загрузить изображение, явно установив свойство src для любого созданного вами изображения. Нет возможности отобразить в веб-броузере созданный таким способом объект Image. Можно лишь заставить объект Image загрузить изображение с помощью свойства src. Однако это может быть удобно, поскольку при этом изображение попадает в кэш броузера. Позднее, если тот

же URL-адрес изображения указывается для одного из изображений в теге <img>, то оно, будучи загруженным заранее, выводится быстро.

См. также Глава 22

Image.onabort DOM Level 2 Events

обработчик, вызываемый при прерывании загрузки изображения

Синтаксис

```
Function onabort
```

Описание

Свойство onabort объекта Image определяет функцию-обработчик события, которая вызывается, когда пользователь прерывает загрузку страницы (например, щелкнув на кнопке Остановить) до того, как изображение успеет загрузиться.

Image.onerror DOM Level 2 Events

обработчик, вызываемый, когда в процессе загрузки изображения возникает ошибка

Синтаксис

```
Function onerror
```

Описание

Свойство onerror объекта Image определяет функцию-обработчик события, которая вызывается, когда в процессе загрузки изображения возникает какая-либо ошибка.

Image.onload DOM Level 2 Events

обработчик, вызываемый по окончании загрузки изображения

Синтаксис

```
Function onload
```

Описание

Свойство onload объекта Image определяет функцию-обработчик события, которая вызывается, когда загрузка изображения благополучно завершается. Подробности см. в справочной статье об обработчике Window.onload.

Input DOM Level 2 HTML

элемент ввода в HTML-форме **Node→Element→HTMLElement→Input**

Свойства

```
String accept
```

Если объект имеет тип file, данное свойство содержит список MIME-типов, разделенных запятыми и определяющих типы файлов, которые могут быть выгружены на сервер. Соответствует атрибуту accept.

`String accessKey`

Комбинация «горячих» клавиш (может состоять из одной клавиши), которая используется броузером для передачи фокуса ввода этому элементу ввода. Соответствует атрибуту `accesskey`.

`устаревшее String align`

Вертикальное выравнивание элемента относительно окружающего текста и других элементов формы слева или справа. Соответствует атрибуту `align`.

`String alt`

Альтернативный текст, который выводится в случае, когда броузер не может отобразить элемент ввода. Соответствует атрибуту `alt`.

`boolean checked`

Когда значением свойства `type` является строка `"radio"` или `"checkbox"`, данное свойство указывает, является соответствующий элемент установленным («помеченным») или нет. Изменение этого свойства вызывает изменение визуального представления элемента ввода. Соответствует атрибуту `checked`.

`boolean defaultChecked`

Когда значением свойства `type` является строка `"radio"` или `"checkbox"`, данное свойство хранит начальное значение атрибута `checked`, указанное в исходном коде документа. Когда выполняется сброс элементов формы, в свойство `checked` записывается значение этого свойства. Изменение значения этого свойства приводит к изменению значения свойства `checked` и текущего состояния элемента.

`String defaultValue`

Когда значением свойства `type` является строка `"text"`, `"password"` или `"file"`, данное свойство хранит начальное значение, отображаемое элементом. Когда выполняется сброс элементов формы, элемент восстанавливается в это значение. Изменение значения этого свойства также приводит к изменению значения свойства `value` и текущее отображаемое значение.

`boolean disabled`

Если свойство имеет значение `true`, элемент ввода неактивен и недоступен пользователю для ввода. Соответствует атрибуту `disabled`.

`readonly HTMLFormElement form`

Объект `Form`, представляющий элемент `<form>`, который содержит данный элемент ввода, или `null`, если элемент ввода находится за пределами формы.

`long maxLength`

Когда значением свойства `type` является строка `"text"` или `"password"`, данное свойство определяет максимальное число символов, которые пользователь сможет ввести. Не путайте это свойство со свойством `size`. Соответствует атрибуту `maxlength`.

`String name`

Имя элемента ввода, указанное в атрибуте `name`. Дополнительные сведения об именовании элементов форм приводятся в подразделе «Описание».

`boolean readOnly`

Если имеет значение `true` и значением свойства `type` является строка `"text"` или `"password"`, пользователь не сможет вводить текст в этот элемент. Соответствует атрибуту `readonly`.

`unsigned long size`

Если значением свойства `type` является строка `"text"` или `"password"`, данное свойство определяет ширину элемента в символах. Соответствует атрибуту `size`. См. также описание свойства `maxLength`.

`String src`

Для элементов ввода со значением свойства `type` равным `"image"` данное свойство определяет URL-адрес выводимого изображения. Соответствует атрибуту `src`.

`long tabIndex`

Порядковый номер элемента, используемый для организации переходов от элемента к элементу с помощью клавиши Tab. Соответствует атрибуту `tabindex`.

`String type`

Тип элемента ввода. Соответствует атрибуту `type`. Дополнительная информация о типах элементов форм приводится в подразделе «Описание».

`String useMap`

Для элементов со значением свойства `type` равным `"image"` данное свойство определяет имя элемента `<map>`, который на стороне клиента представляет карту изображений для элемента.

`String value`

Это значение передается веб-серверу при отправке данных формы. Для элементов со значением свойства `type` равным `"text"`, `"password"` или `"file"` данное свойство представляет редактируемый текст, содержащийся в элементе ввода. Для элементов со значением свойства `type` равным `"button"`, `"submit"` или `"reset"` данное свойство представляет недоступную для редактирования текстовую метку, которая отображается на кнопке. Из соображений безопасности значение свойства `value` элементов `FileUpload` может быть недоступно для изменения. Аналогичным образом значение, возвращаемое этим свойством для элементов `Password`, может не совпадать с тем, что фактически введено пользователем.

Методы

`blur()`

Убирает фокус ввода с элемента.

`click()`

Для элементов со значением свойства `type` равным `"button"`, `"checkbox"`, `"radio"`, `"reset"` или `"submit"` данный метод имитирует щелчок мыши на элементе.

`focus()`

Передает элементу фокус ввода.

`select()`

Для элементов со значением свойства `type` равным `"file"`, `"password"` или `"text"` данный метод выделяет текст, отображаемый элементом. Во многих броузерах это означает, что при вводе очередного символа выделенный текст будет удален и замещен введенным символом.

Обработчики событий

`onblur`

Вызывается, когда элемент теряет фокус ввода.

onchange

Для элементов ввода текста этот обработчик события вызывается, когда пользователь изменяет выводимый текст и затем «подтверждает» эти изменения, переходя к другому элементу нажатием клавиши Tab или щелчком мыши, в результате чего фокус ввода передается другому элементу. Данный обработчик не вызывается для каждого нажатия клавиши в процессе редактирования. Элементы с фиксацией состояния (флажки и переключатели) также могут генерировать это событие (в дополнение к событию onclick) при изменении их состояния пользователем.

onclick

Для элементов форм, являющихся кнопками, этот обработчик события вызывается, когда пользователь активизирует элемент с помощью мыши или клавиатуры.

onfocus

Вызывается, когда пользователь передает элементу фокус ввода.

Описание

Объект Input представляет HTML-тег <input>, с помощью которого определяются элементы ввода формы. Наиболее важными свойствами объекта Input являются type, value и name. Эти свойства описываются в следующих далее подразделах. Дополнительная информация о формах и элементах форм приводится в главе 18.

Типы элементов ввода

Атрибут type тега <input> определяет тип создаваемого элемента ввода. В клиентских JavaScript-сценариях этот атрибут доступен в виде свойства type объекта Input и может использоваться, например, для определения типов неизвестных элементов форм при обходе их с помощью массива elements[] объекта Form.

Свойство type может принимать следующие значения:

"button"

Элемент ввода представляет собой обычную кнопку, на поверхности которой может отображаться надпись, определяемая свойством value. Кнопка не имеет поведения, предлагаемого по умолчанию, и чтобы делать что-то полезное, должна иметь обработчик события onclick. Для кнопок отправки данных формы или сброса формы свойство type должно иметь значение "submit" или "reset". Обратите внимание: с помощью HTML-тега <button> можно создавать кнопки, отображающие произвольный HTML-код, а не только простой текст.

"checkbox"

Элемент ввода этого типа графически выглядит как *флажок*, который может быть установлен или сброшен пользователем. Текущее состояние флажка хранится в свойстве checked, а обработчик события onclick вызывается при изменении состояния флажка (броузеры могут при этом вызывать обработчик события onchange). Свойство value — это внутреннее представление значения, которое отправляется веб-серверу и не показывается пользователю. Чтобы разместить надпись рядом с тегом <input>, при желании можно задействовать тег <label>. Флажки часто размещаются в виде групп, причем для удобства анализа формы на стороне веб-сервера всем элементам одной группы присваивается одинаковое значение свойства name и разные значения свойства value.

"file"

Элемент ввода этого типа создает элемент выгрузки файла. Этот элемент содержит текстовое поле ввода, куда вводится имя файла, дополненное кнопкой, по щелчку

на которой открывается диалоговое окно выбора файла. Имя файла, выбранного пользователем, сохраняется в свойстве value, при этом когда выполняется отправка данных формы, содержащей элемент выгрузки файла, броузер отправляет на сервер не имя выбранного файла, а его содержимое. (Чтобы этот прием работал, форма должна использовать кодировку multipart/form-data и метод POST.)

Для безопасности элемент выгрузки файла не позволяет авторам HTML-страницы или JavaScript-сценария определять имя файла, предлагаемое по умолчанию. В элементах этого типа значения HTML-атрибута value и JavaScript-свойства value игнорируются и доступны только для чтения – это означает, что только пользователь может ввести имя файла. Когда пользователь выбирает или редактирует имя файла, вызывается обработчик события onchange.

"hidden"

Элемент ввода этого типа фактически невидим. Свойство value невидимого элемента формы может хранить произвольную строку, которая передается веб-серверу. Элемент этого типа может применяться для передачи данных, которые не вводятся пользователем непосредственно.

"image"

Элемент ввода этого типа представляет кнопку отправки данных формы, которая выглядит как изображение (определяемое свойством src), а не как кнопка с надписью. Свойство value этого элемента не используется. Дополнительная информация приводится в описании значения "submit".

"password"

Это текстовое поле ввода предназначено для ввода секретных данных, таких как пароли. Символы, вводимые пользователем, маскируются (например звездочками), чтобы предотвратить возможность просмотра вводимой информации посторонними наблюдателями. Однако следует заметить, что ввод пользователя не шифруется каким-либо образом: когда форма отправляется, текст передается в открытом виде. С целью обеспечения безопасности некоторые броузеры могут предотвращать возможность чтения значения свойства value. Во всем остальном элемент ввода типа password ведет себя аналогично элементу типа text. Он вызывает обработчик события onchange, когда пользователь изменяет выводимое значение.

"radio"

Элемент ввода этого типа выглядит как *переключатель*. Группа переключателей представляет набор взаимоисключающих вариантов. Когда устанавливается один из переключателей в группе, предыдущий установленный переключатель автоматически сбрасывается (на манер кнопок выбора радиостанций в старых автомобильных радиоприемниках). Для объединения переключателей в группу и обеспечения взаимоисключающего поведения они должны присутствовать в пределах одного и того же элемента <form> и иметь одно и то же значение атрибута name. Для флажков, которые не предлагают взаимоисключающий выбор, используется тип checkbox. Обратите внимание: для обеспечения возможности выбора вариантов (как взаимоисключающих, так и нет) можно также использовать HTML-тег <select> (см. описание объекта Select).

Свойство checked указывает, установлен ли данный переключатель. Не существует какого-либо другого способа узнать, какой переключатель в группе установлен, поэтому приходится проверять значение свойства checked у всех переключателей. При изменении значения свойства checked переключатели вызывают обработчик события onclick.

Свойство value определяет отправляемое на веб-сервер значение, которое не отображается в форме. Чтобы рядом с переключателем разместить надпись, необходимо делать это за пределами тега <input>, например, с помощью тега <label>.

"reset"

Элемент ввода этого типа напоминает обычную кнопку, созданную как элемент ввода типа button, но имеет предопределенное назначение. Когда выполняется щелчок мышью на элементе ввода типа reset, значения всех элементов ввода формы сбрасываются к их значениям, предлагаемым по умолчанию (определяются HTML-атрибутом value или JavaScript-свойством defaultValue).

Свойство value определяет текст, который выводится на кнопке. Перед сбросом формы вызывается обработчик события onclick и этот обработчик может отменить операцию сброса, вернув значение false или используя иные способы отмены, которые описываются в главе 17. Дополнительные сведения ищите в справочных статьях о методе Form.reset() и обработчике Form.onreset.

"submit"

Элемент ввода этого типа представляет собой кнопка, которая после щелчка на ней мыши выполняет отправку данных содержащей ее формы. Свойство value определяет текст, который выводится на кнопке. Перед отправкой формы вызывается обработчик события onclick и этот обработчик может отменить операцию отправки, вернув значение false. Дополнительные сведения ищите в справочных статьях о методе Form.submit() и обработчике Form.onsubmit.

"text"

Это значение по умолчанию для свойства type. Элемент ввода этого типа представляет собой однострочное текстовое поле ввода. HTML-атрибут value задает предлагаемый по умолчаию текст, который отображается в поле ввода, а JavaScript-свойство value хранит текущий текст. Когда пользователь отредактирует текст и передаст фокус ввода другому элементу, будет вызван обработчик события onchange. Свойство size служит для задания ширины поля ввода, а maxLength – для определения максимального числа символов, которые можно ввести в это поле. Когда форма содержит единственный элемент типа text, нажатие клавиши Enter вызывает отправку данных формы.

Для создания многострочных текстовых полей ввода используется тег <textarea> (см. справочную статью об объекте Textarea). Для ввода секретных данных используется текстовый элемент ввода типа password.

Значения элементов ввода

Свойство value объекта Input представляет собой доступное для чтения и записи строковое свойство, определяющее текст, который передается веб-серверу при отправке данных формы, содержащей элемент ввода.

В зависимости от значения свойства type свойство value может содержать видимый для пользователя текст. Для элементов ввода типа text и file это свойство хранит текст, который вводится пользователем. Для элементов типа button, reset и submit свойство value определяет текст надписи на кнопке. Для элементов ввода других типов, таких как checkbox, radio и image, значение свойства value не показывается пользователю и требуется только для представления формы.

Имена элементов ввода

Свойство name объекта Input – это строка, представляющая имя элемента ввода. Это значение определяется атрибутом name. Имя элемента формы требуется для достиже-

ния двух целей. Во-первых, оно используется при отправке формы. Обычно данные каждой формы передаются в формате:

```
имя=значение
```

Здесь *имя* и *значение* в случае необходимости кодируются. Если имя элемента формы не определено, данные этого элемента не могут быть отправлены на веб-сервер.

Во-вторых, свойство `name` может использоваться для получения ссылки на элемент ввода в JavaScript-сценарии. Имя элемента становится свойством формы, содержащей элемент. Значением этого свойства является ссылка на элемент. Например, если предположить, что `address` – это имя формы, содержащей текстовое поле ввода с именем `zip`, тогда свойство `address.zip` будет ссылаться на это текстовое поле ввода.

Для элементов ввода типа `radio` и `checkbox` обычной практикой считается присваивание свойству `name` нескольких родственных объектов одного и того же значения. В этом случае данные отправляются на сервер в следующем формате:

```
имя=значение1, значение2, . . . , значениеn
```

Аналогичным образом каждый элемент в JavaScript, который использует одно и то же имя, становится элементом массива с тем же именем. Таким образом, четыре объекта `Checkbox` с именем `options` на форме `order` в JavaScript-сценарии будут доступны как элементы массива `order.options[]`.

Родственные элементы форм

Тег `<input>` позволяет создавать разнообразные элементы форм. Но элементы форм можно также создавать с помощью тегов `<button>`, `<select>` и `<textarea>`.

См. также `Form`, `Form.elements[]`, `Option`, `Select`, `Textarea`; глава 18

Input.blur()

DOM Level 2 HTML

убирает фокус ввода с элемента формы

Синтаксис

```
void blur()
```

Описание

Метод `blur()` элемента формы убирает фокус ввода с этого элемента без вызова обработчика события `onblur`; по существу, он противоположен методу `focus()`. Однако метод `blur()` никуда не передает фокус ввода, вот почему вызов этого метода полезен в единственном случае – когда нужно передать фокус ввода другому элементу методом `focus()`, но при этом избежать вызова обработчика события `onblur`. То есть после явного удаления фокуса с элемента сценарий не получит уведомление, в противоположность тому, когда фокус убирается неявно вызовом метода `focus()` другого элемента.

Input.click()

DOM Level 2 HTML

имитирует щелчок мыши на элементе формы

Синтаксис

```
void click()
```

Описание

Метод `click()` элемента формы имитирует щелчок мыши на элементе формы, но не вызывает обработчик события `onclick` для элемента.

Метод click() редко может быть полезен. Он не вызывает обработчик события onclick, поэтому нет смысла вызывать его для элементов типа button – они не делают ничего кроме того, что определено обработчиком onclick. Вызов click() для элемента типа submit или reset передает данные или очищает форму, но это достигается и проще – методами submit() и reset() самого объекта Form.

Input.focus() DOM Level 2 HTML

передает фокус ввода элементу формы

Синтаксис

```
void focus()
```

Описание

Метод focus() элемента формы передает фокус ввода элементу, не вызывая обработчик события onfocus, делая элемент активным для клавиатурной навигации и клавиатурного ввода. Другими словами, если вы вызовете focus() для элемента типа text, то любой текст, набранный пользователем, появится в этом текстовом элементе. А если вы вызовете focus() для элемента типа button, пользователь сможет активизировать эту кнопку с клавиатуры.

Input.onblur DOM Level 0

обработчик, вызываемый элементом формы при потере фокуса ввода

Синтаксис

```
Function onblur
```

Описание

Свойство onblur объекта Input определяет функцию обработки события, вызываемую, когда пользователь убирает фокус ввода с элемента ввода. Вызов метода blur() при удалении фокуса с элемента не приводит к вызову обработчика события onblur для этого объекта. Однако следует отметить, что вызов метода focus() для передачи фокуса какому-либо другому элементу приводит к вызову обработчика события onblur для того элемента, который в данный момент имеет фокус.

См. также Element.addEventListener(), Window.onblur; глава 17

Input.onchange DOM Level 2 Events

обработчик, вызываемый элементом формы при изменении значения

Синтаксис

```
Function onchange
```

Описание

Свойство onchange объекта Input определяет функцию обработки события, которая вызывается, когда пользователь изменяет значение, отображаемое элементом формы. Таким изменением может быть редактирование текста, выводимого в элементах типа text, password и file, либо установка или сброс элементна типа radio и checkbox (флажки и переключатели всегда вызывают обработчик события onclick и могут также вызывать обработчик события onchange). Обратите внимание: этот обработчик вы-

зывается, только когда пользователь завершает изменение, но не вызывается, если отображаемое элементом значение изменяется JavaScript-программой.

Кроме того, следует отметить, что onchange не вызывается при вводе или удалении каждого символа в элементе ввода текста на форме. Обработчик onchange не предназначен для обработки событий на посимвольной основе; вместо этого он вызывается, когда пользователь завершает редактирование. Броузер предполагает, что редактирование завершено, когда фокус ввода перемещается от текущего элемента к какому-либо другому элементу, например, когда пользователь щелкает на следующем элементе формы. Уведомления о событиях на посимвольной основе рассматриваются в справочной статье об обработчике HTMLElement.onkeypress.

Обработчик события onchange не используется элементами типа button, hidden, image, reset и submit. Вместо него элементы этих типов задействуют обработчик события onclick.

См. также Element.addEventListener(), HTMLElement.onkeypress; глава 17

Input.onclick

DOM Level 2 Events

обработчик, вызываемый элементом формы при щелчке мыши

Синтаксис

```
Function onclick
```

Описание

Свойство onclick объекта Input определяет функцию обработки события, которая вызывается, когда пользователь активизирует элемент ввода. Обычно это производится щелчком мыши на элементе, но обработчик onclick вызывается также, когда пользователь активизирует элемент, выполняя переход к нему с помощью клавиатуры. Обработчик onclick не вызывается при обращении к методу click().

Обратите внимание: элементы типа reset и submit при щелчке выполняют действие, предлагаемое по умолчанию: они соответственно сбрасывают и передают данные формы, в которой они содержатся. При помощи обработчиков события onclick каждого из этих элементов можно осуществлять действия, дополняющие то, что предлагается по умолчанию. Можно также предотвратить выполнение действий, предлагаемых по умолчанию, вернув из обработчика значение false или воспользовавшись другими путями отмены реакции на событие, описываемыми в главе 17. Обратите внимание: то же самое может быть сделано с помощью обработчиков событий onsubmit и onreset самого объекта Form.

См. также Element.addEventListener(); глава 17

Input.onfocus

DOM Level 2 Events

обработчик, вызываемый элементом формы при получении фокуса ввода

Синтаксис

```
Function onfocus
```

Описание

Свойство onfocus объекта Input определяет функцию обработки события, вызываемую, когда пользователь передает фокус ввода этому элементу. Вызов метода focus()

для установки фокуса на элемент не приводит к возбуждению события `onfocus` для этого объекта.

См. также `Element.addEventListener()`, `Window.onfocus`; глава 17

Input.select() DOM Level 2 HTML

выделяет текст в элементе формы

Синтаксис

```
void select()
```

Описание

Метод `select()` выделяет текст, отображаемый в элементах типа `text`, `password` или `file`. На разных платформах это может выглядеть по-разному, но обычно вызов этого метода приводит к выделению текста, что позволяет копировать и вставлять его, а также удалить при вводе очередного символа.

JavaArray, JavaClass, JavaObject, JavaPackage

См. часть III книги

JSObject Java-класс в подключаемом Java-модуле

Java-представление JavaScript-объекта

Синтаксис

```
public final class netscape.javascript.JSObject extends Object
```

Методы

`call()`

Вызывает метод JavaScript-объекта.

`eval()`

Вычисляет строку JavaScript-кода в контексте JavaScript-объекта.

`getMember()`

Получает значение свойства JavaScript-объекта.

`getSlot()`

Получает значение элемента массива JavaScript-объекта.

`getWindow()`

Получает «корневой» элемент `JSObject`, представляющий JavaScript-объект `Window` веб-броузера.

`removeMember()`

Удаляет свойство из JavaScript-объекта.

`setMember()`

Устанавливает значение свойства JavaScript-объекта.

`setSlot()`

Устанавливает значение элемента массива в JavaScript-объекте.

`toString()`

Вызывает метод `toString()` JavaScript-объекта и возвращает его результат.

Описание

Объект `JSObject` – это Java-класс, а не JavaScript-объект; его нельзя использовать в JavaScript-программах. Однако `JSObject` применяется Java-апплетами, которым необходимо взаимодействовать с JavaScript-кодом, путем чтения и записи свойств объектов и элементов JavaScript-массивов, вызова JavaScript-методов, вычисления и исполнения произвольных строк JavaScript-кода. Поскольку `JSObject` представляет собой класс Java, необходимо разбираться в Java-программировании, чтобы его использовать.

Подробную информацию о программировании с применением Java-класса `JSObject` можно найти в главе 23.

См. также Главы 23 и 12; `JavaObject` в части III книги

JSObject.call()

Java-метод в подключаемом Java-модуле

вызывает метод JavaScript-объекта

Синтаксис

```
public Object call(String имяМетода, Object аргументы[])
```

Аргументы

имяМетода

Имя вызываемого JavaScript-метода.

аргументы[]

Массив Java-объектов, которые должны быть переданы методу в качестве аргументов.

Возвращаемое значение

Java-объект, представляющий возвращаемое значение JavaScript-метода.

Описание

Метод `call()` Java-класса `JSObject` вызывает именованный метод JavaScript-объекта, представленного этим классом. Аргументы передаются методу в виде массива Java-объектов, а возвращаемое значение JavaScript-метода представляет собой Java-объект.

В главе 23 описан механизм преобразования данных для аргументов метода из Java-объектов в JavaScript-значения и возвращаемого JavaScript-значения в Java-объект.

JSObject.eval()

Java-метод в подключаемом Java-модуле

вычисляет строку JavaScript-кода

Синтаксис

```
public Object eval(String s)
```

Аргументы

s Строка, содержащая произвольные JavaScript-инструкции, разделенные точками с запятой.

Возвращаемое значение

JavaScript-значение последнего вычисленного выражения в *s*, преобразованное затем в Java-объект.

Описание

Метод `eval()` Java-класса `JSObject` вычисляет JavaScript-код, содержащийся в строке *s*, в контексте JavaScript-объекта, определяемого Java-классом `JSObject`. Поведение метода `eval()` Java-класса `JSObject` во многом схоже с поведением глобальной JavaScript-функции `eval()`.

Аргумент *s* может содержать любое число JavaScript-инструкций, разделенных точками с запятой; эти инструкции исполняются в том порядке, в котором они указаны. Значение, возвращаемое `eval()`, – это значение последней вычисленной инструкции или выражения в *s*.

JSObject.getMember()

Java-метод в подключаемом Java-модуле

читает свойство JavaScript-объекта

Синтаксис

```
public Object getMember(String имя)
```

Аргументы

имя Имя свойства, которое должно быть прочитано.

Возвращаемое значение

Java-объект, содержащий значение именованного свойства указанного Java-класса `JSObject`.

Описание

Метод `getMember()` Java-класса `JSObject` читает и передает в Java значение указанного свойства JavaScript-объекта. Возвращаемое значение может быть другим объектом `JSObject` или объектом типа `Double`, `Boolean` или `String`, но возвращается в виде обобщенного объекта `Object`, который необходимо преобразовать так, как это требуется.

JSObject.getSlot()

Java-метод в подключаемом Java-модуле

читает элемент массива JavaScript-объекта

Синтаксис

```
public Object getSlot(int индекс)
```

Аргументы

индекс Индекс элемента массива, который должен быть прочитан.

Возвращаемое значение

Значение элемента массива по указанному индексу в JavaScript-объекте.

Описание

Метод `getSlot()` Java-класса `JSObject` читает и возвращает в Java значение элемента массива по указанному *индексу* в объекте JavaScript. Возвращаемое значение может

быть другим объектом JSObject или объектом типа Double, Boolean или String, но возвращается в виде обобщенного объекта Object, который необходимо преобразовать так, как это требуется.

JSObject.getWindow()

Java-метод в подключаемом Java-модуле

возвращает начальный объект JSObject для окна броузера

Синтаксис

```
public static JSObject getWindow(java.applet.Applet апплет)
```

Аргументы

апплет

Объект Applet, исполняющийся в окне веб-броузера, для которого должен быть получен объект JSObject.

Возвращаемое значение

Объект JSObject, представляющий JavaScript-объект Window для окна веб-броузера, которое содержит указанный *апплет*.

Описание

Метод getWindow() – это первый метод объекта JSObject, который вызывает любой Java-апплет. Объект JSObject не определяет конструктора, а статический метод getWindow() – это единственный метод, способный получить «корневой» объект JSObject, от которого могут быть получены другие объекты JSObject.

JSObject.removeMember()

Java-метод в подключаемом Java-модуле

удаляет свойство JavaScript-объекта

Синтаксис

```
public void removeMember(String имя)
```

Аргументы

имя Имя свойства, которое должно быть удалено из объекта JSObject.

Описание

Метод removeMember() Java-класса JSObject удаляет указанное свойство из JavaScript-объекта, представленного объектом JSObject.

JSObject.setMember()

Java-метод в подключаемом Java-модуле

устанавливает значение свойства JavaScript-объекта

Синтаксис

```
public void setMember(String имя, Object значение)
```

Аргументы

имя Имя устанавливаемого в объекте JSObject свойства.

значение Значение, которое должно быть присвоено указанному свойству.

Описание

Метод `setMember()` Java-класса `JSObject` устанавливает значение указанного свойства JavaScript-объекта из Java. Заданное *значение* может быть любым Java-объектом. Элементарные Java-значения не могут передаваться этому методу. В JavaScript *значение* доступно как объект `JavaObject`.

JSObject.setSlot()

Java-метод в подключаемом Java-модуле

устанавливает значение элемента массива в JavaScript-объекте

Синтаксис

```
public void setSlot(int индекс, Object значение)
```

Аргументы

индекс

Индекс элемента массива, который должен быть установлен в объекте `JSObject`.

значение

Значение, которое должно быть присвоено указанному элементу массива.

Описание

Метод `setSlot()` Java-класса `JSObject` устанавливает значение элемента массива с указанным номером в JavaScript-объекте из Java. Заданное *значение* может быть любым Java-объектом. Элементарные Java-значения не могут быть переданы этому методу. В JavaScript *значение* доступно в виде объекта `JavaObject`.

JSObject.toString()

Java-метод в подключаемом Java-модуле

возвращает строковое представление JavaScript-объекта

Синтаксис

```
public String toString()
```

Возвращаемое значение

Строка, получаемая методом `toString()` JavaScript-объекта, представленного указанным Java-объектом `JSObject`.

Описание

Метод `toString()` Java-класса `JSObject` вызывает метод `toString()` JavaScript-объекта, представленного объектом `JSObject`, и возвращает результат этого метода.

KeyEvent

Firefox и совместимые броузеры

сведения о событии клавиатуры **Event→UIEvent→KeyEvent**

Свойства

```
readonly boolean altKey
```

Признак, показывающий, удерживалась ли клавиша Alt в момент возникновения события.

`readonly integer charCode`

Это число – код печатного символа (если имеется) в кодировке Unicode, сгенерированного событием `keypress`. Данное свойство равно нулю в случае нажатия функциональной клавиши, оно не используется событиями `keydown` и `keyup`. Чтобы преобразовать это число в строку, можно задействовать метод `String.fromCharCode()`.

`readonly boolean ctrlKey`

Признак, показывающий, удерживалась ли клавиша Ctrl в момент возникновения события. Определено для всех типов событий от мыши.

`readonly integer keyCode`

Виртуальный код нажатой клавиши. Это свойство используется всеми типами событий от клавиатуры. Коды клавиш могут зависеть от типа броузера, операционной системы и аппаратных особенностей самой клавиатуры. Как правило, если на клавише нарисован печатный символ, виртуальный код клавиши совпадает с кодом символа. Коды функциональных клавиш могут изменяться в широком диапазоне, но наиболее часто используемые коды приводятся в примере 17.6.

`readonly boolean shiftKey`

Признак, показывающий, удерживалась ли клавиша Shift в момент возникновения события. Определено для всех типов событий от мыши.

Описание

Объект `KeyEvent` передается обработчикам событий `keydown`, `keypress` и `keyup` и содержит информацию о событии от клавиатуры. Стандарт DOM Level 2 не описывает события от клавиатуры, потому объект `KeyEvent` не стандартизован. Данная статья описывает реализацию в броузере Firefox. Многие из перечисленных свойств поддерживаются также в модели событий IE; свойства, специфичные для IE, описаны в справочной статье об объекте `Event`. Обратите внимание: в дополнение к перечисленным свойствам объекты `KeyEvent` наследуют свойства интерфейсов `Event` и `UIEvent`.

Практические примеры использования объектов `KeyEvent` приводятся в главе 17.

См. также

`Event`, `UIEvent`; глава 17

Layer

только Netscape 4; в Netscape 6 не поддерживается

устаревший прикладной интерфейс Netscape

Описание

Объект `Layer` использовался в Netscape 4 для поддержки динамического позиционирования HTML-элементов. Он так и не был стандартизован и ныне считается устаревшим.

См. также

Глава 16

Link

DOM Level 0

гиперссылка или якорный элемент в HTML-документе

Node→Element→HTMLElement→Link

Свойства

Наиболее важным свойством объекта `Link` является `href`, в котором хранится URL-адрес ссылки. Объект `Link` определяет также ряд дополнительных свойств, которые

представляют фрагменты URL-адреса. Для каждого из этих свойств дается пример фрагмента следующего (фиктивного) URL-адреса:

```
http://www.oreilly.com:1234/catalog/search.html?q=JavaScript&m=10#results
```

`String hash`

Соответствует якорной части URL-адреса ссылки, включая начальный символ решетки (#), например: `"#result"`. Эта якорная часть URL-адреса ссылается на именованную позицию внутри документа, на который указывает объект `Link`. В HTML-файлах позиции обозначаются якорными элементами, созданными с помощью атрибута `name` тега `<a>` (см. статью об объекте `Anchor`).

`String host`

Соответствует той части URL-адреса ссылки, которая содержит имя хоста и порт, например: `"www.oreilly.com:1234"`.

`String hostname`

Соответствует имени хоста в URL-адресе ссылки, например: `"www.oreilly.com"`.

`String href`

Соответствует полному тексту URL-адреса ссылки в отличие от других свойств объекта `Link`, описывающих лишь некоторую часть URL-адреса.

`String pathname`

Соответствует имени пути в URL-адреса ссылки, например: `"/catalog/search.html"`.

`String port`

Соответствует номеру порта в URL-адреса ссылки, например: `"1234"`.

`String protocol`

Соответствует спецификатору протокола в URL-адреса ссылки, включая замыкающее двоеточие, например: `"http:"`.

`String search`

Соответствует той части URL-адреса ссылки, которая содержит строку запроса, включая начальный вопросительный знак, например: `"?q=JavaScript&m=10"`.

В дополнение к перечисленным свойствам, имеющим отношение к URL-адресу, объект `Link` определяет свойства, которые соответствуют атрибутам тегов `<a>` и `<area>`:

Свойство	Атрибут	Описание
`String accessKey`	`accesskey`	Комбинация «горячих» клавиш
`String charset`	`charset`	Кодировка целевого документа
`String coords`	`coords`	Используется тегами `<area>`
`String hreflang`	`hreflang`	Язык связанного документа
`String name`	`name`	Имя якорного элемента. Подробности см. в статье об объекте `Anchor`
`String rel`	`rel`	Тип ссылки
`String rev`	`rev`	Тип обратной ссылки
`String shape`	`shape`	Используется тегами `<area>`
`long tabIndex`	`tabIndex`	Порядковый номер при переходе между элементами с помощью клавиши табуляции

Свойство	Атрибут	Описание
String target	target	Имя фрейма или окна, в котором должен быть отображен целевой документ
String type	type	Тип содержимого целевого документа

Методы

blur()

Отбирает фокус ввода у ссылки.

focus()

Прокручивает документ, пока ссылка не станет видимой, и передает ей фокус ввода.

Обработчики событий

Объект Link определяет особое поведение для следующих трех обработчиков событий:

onclick

Вызывается, когда пользователь щелкает на ссылке.

onmouseout

Вызывается, когда пользователь уводит указатель мыши от ссылки.

onmouseover

Вызывается, когда пользователь наводит указатель мыши на ссылку.

Синтаксис HTML

Объект Link создается с помощью стандартных тегов <a> и . Для всех объектов Link атрибут href обязателен. Если также указан атрибут name, создается еще и объект Anchor:

```
<a href="url"                  // Цель ссылки
  [ name="якорь" ]             // Создать объект Anchor
  [ target="имя_окна" ]        // Место вывода нового документа
  [ onclick="обработчик" ]     // Вызывается при щелчке на ссылке
  [ onmouseover="обработчик" ] // Вызывается при наведении указателя мыши на ссылку
  [ onmouseout="обработчик" ]  // Вызывается при смещении указателя мыши со ссылки
>текст или изображение ссылки  // Видимая часть ссылки
</a>
```

Описание

Объект Link представляет гиперссылку в HTML-документе. Обычно ссылки создаются тегами <a> с определенным атрибутом href, но они могут также создаваться тегами <area> внутри карты изображений на стороне клиента. Когда вместо атрибута href в теге <a> определяется атрибут name, он задает именованную позицию в документе и представляется объектом Anchor, а не Link. Подробности см. в справочной статье об объекте Anchor.

Все ссылки в документе (созданные с помощью тегов <a> и <area>) представляются объектами Link и сохраняются в массиве links[] объекта Document.

Целью гиперссылки является, очевидно, URL-адрес, поэтому многие свойства объекта Link задают содержимое этого URL-адреса. Объект Link похож на объект Location, который также имеет полный набор свойств URL-адреса.

Пример

```
// Получить URL-адрес первой гиперссылки в документе
var url = document.links[0].href;
```

См. также Anchor, Location

Link.blur() DOM Level 0

убирает фокус ввода с гиперссылки

Синтаксис

```
void blur()
```

Описание

В веб-броузерах, которые допускают передачу фокуса ввода гиперссылкам, этот метод убирает фокус ввода с гиперссылки.

Link.focus() DOM Level 0

передает фокус ввода гиперссылке, делая ее видимой

Синтаксис

```
void focus()
```

Описание

Этот метод выполняет прокрутку документа, пока заданная гиперссылка не станет видимой. Если веб-броузер допускает передачу фокуса ввода гиперссылкам, этот метод передает фокус ввода гиперссылке.

Link.onclick DOM Level 0

обработчик, вызываемый при щелчке на ссылке

Синтаксис

```
Function onclick
```

Описание

Свойство onclick объекта Link определяет функцию обработки события, которая вызывается, когда пользователь щелкает на ссылке. По умолчанию после вызова обработчика события броузер выполняет переход по ссылке, на которой произошел щелчок. Предотвратить это действие можно, если вернуть из обработчика события значение false или воспользоваться одним из приемов отмены события, описываемых в главе 17.

См. также Element.addEventListener(), MouseEvent; глава 17

Link.onmouseout DOM Level 0

обработчик, вызываемый, когда указатель мыши перемещается за пределы ссылки

Синтаксис

```
Function onmouseout
```

Описание

Свойство onmouseout объекта Link определяет функцию обработки события, которая вызывается, когда пользователь уводит указатель мыши от гиперссылки. Зачастую используется в паре с обработчиком события onmouseover.

См. также Element.addEventListener(), Link.onmouseover, MouseEvent; глава 17

Link.onmouseover DOM Level 0

обработчик, вызываемый, когда указатель мыши наводится на ссылку

Синтаксис

```
Function onmouseout
```

Описание

Свойство onmouseover объекта Link задает функцию обработки события, вызываемую, когда пользователь наводит указатель мыши на гиперссылку. Если пользователь удерживает некоторое время указатель мыши на ссылке, броузер отображает URL-адрес этой ссылки в строке состояния. В старых версиях броузеров имелась возможность отменить это действие, предлагаемое по умолчанию, чтобы вывести в строке состояния собственный текст. По соображениям безопасности (например, чтобы предотвратить возможность обмана) в большинстве современных броузеров такая возможность отсутствует.

См. также Element.addEventListener(), Link.onmouseout, MouseEvent; глава 17

Location JavaScript 1.0

представление адреса в броузере и управление им **Object→Location**

Синтаксис

```
location
окно.location
```

Свойства

Свойства объекта Location ссылаются на различные фрагменты URL-адреса текущего документа. Для каждого из следующих свойств дается пример фрагмента следующего (фиктивного) URL-адреса:

```
http://www.oreilly.com:1234/catalog/search.html?q=JavaScript&m=10#results
```

hash

Доступное для чтения и записи свойство, задающее якорную часть URL-адреса, включая начальный символ решетки (#), например "#result". Эта часть URL-адреса документа задает имя якорного элемента внутри документа.

host

Доступное для чтения и записи строковое свойство, которое задает часть URL-адреса, содержащую имя хоста и порт, например "www.oreilly.com:1234".

hostname

Доступное для чтения и записи строковое свойство, задающее имя хоста в URL-адресе, например "www.oreilly.com".

`href`

Доступное для чтения и записи строковое свойство, задающее полный текст URL-адреса документа, в отличие от других свойств объекта `Location`, которые определяют только части URL-адреса. Присваивание этому свойству нового URL-адреса приводит к тому, что броузер читает и отображает содержимое нового URL-адреса.

`pathname`

Доступное для чтения и записи строковое свойство, задающее путь в URL-адресе, например `"/catalog/search.html"`.

`port`

Доступное для чтения и записи строковое (не числовое) свойство, задающее порт в URL-адресе ссылки, например `"1234"`.

`protocol`

Доступное для чтения и записи строковое свойство, задающее протокол в URL-адресе, включая замыкающее двоеточие, например `"http:"`.

`search`

Доступное для чтения и записи строковое свойство, задающее часть URL-адреса, которая содержит строку запроса, включая начальный вопросительный знак, например `"?q=JavaScript&m=10"`.

Методы

`reload()`

Повторно загружает текущий документ из кэша или с сервера.

`replace()`

Заменяет текущее окно новым, при этом новая запись в истории сеанса броузера не генерируется.

Описание

Объект `Location` хранится в свойстве `location` объекта `Window` и представляет веб-адрес («местоположение») документа, отображаемого в данный момент в этом окне. Свойство `href` содержит полный URL-адрес этого документа, а каждое из оставшихся свойств объекта `Location` описывает фрагмент этого URL-адреса. Эти свойства очень похожи на свойства URL-адреса объекта `Link`. Когда объект `Location` преобразуется в строку, возвращается значение свойства `href`. Это означает, что вместо выражения `location.href` можно использовать просто `location`.

Если объект `Link` представляет гиперссылку в документе, то объект `Location` представляет URL-адрес, отображаемый в данный момент в броузере. Но объект `Location` еще и *управляет* адресом, отображаемым в броузере. Если строку, содержащую URL-адрес, присвоить объекту `Location` или его свойству `href`, то веб-броузер загрузит указанный URL-адрес и отобразит документ, на который он ссылается.

Замена текущего URL-адреса совершенно новым адресом осуществляется не только присваиванием значения `location` или `location.href`, но и путем изменения части текущего URL-адреса. Таким образом создается новый URL-адрес с одним измененным фрагментом, и этот новый адрес загружается и отображается броузером. Так, установив свойство `hash` объекта `Location`, можно заставить броузер переместиться к именованному месту в текущем документе. Аналогично, установив свойство `search`, можно заставить броузер заново загрузить текущий URL-адрес, дополненный новой строкой запроса.

В дополнение к свойствам URL-адреса объект Location определяет два метода. Метод reload() повторно загружает текущий документ, а метод replace() загружает новый документ без создания для него новой записи в списке истории броузера – новый документ просто заменяет в этом списке текущий.

См. также Link, свойство URL объекта HTMLDocument

Location.reload() JavaScript 1.1

перезагружает текущий документ

Синтаксис

location.reload()
location.reload(*заново*)

Аргументы

заново

Логический аргумент, определяющий, должен ли документ загружаться заново, даже если сервер сообщает о том, что документ не был изменен с момента его последней загрузки. Если этот аргумент опущен или равен false, метод загружает страницу целиком, только если она была изменена с момента последней загрузки.

Описание

Метод reload() объекта Location заново загружает документ, выводимый в окне, к которому относится объект Location. При вызове без аргументов или с аргументом false он определяет, был ли документ изменен на веб-сервере, по HTTP-заголовку If-Modified-Since. Если документ был изменен, метод reload заново загружает документ с сервера, а если нет, загружает его из кэша. Это то же самое действие, которое выполняется, когда пользователь щелкает на кнопке Перезагрузить в броузере.

Когда метод reload() вызывается с аргументом, равным true, он всегда загружает документ с сервера в обход кэша независимо от времени последней модификации документа. Это действие идентично тому, которое выполняется, когда пользователь в броузере щелкает на кнопке Перезагрузить, удерживая нажатой клавишу Shift.

Location.replace() JavaScript 1.1

заменяет один отображаемый документ другим

Синтаксис

location.replace(*url*)

Аргументы

url Строка, задающая URL-адрес нового документа, который должен заменить текущий.

Описание

Метод replace() объекта Location загружает и отображает новый документ. Важное отличие этого способа загрузки документа от простого присваивания значения location или location.href состоит в том, что метод replace() не генерирует новую запись в объекте History. В последнем случае новый URL-адрес заменяет текущую запись

в объекте History. После вызова replace() кнопка Назад броузера возвращает нас не к предыдущему URL-адресу, а к предпоследнему.

См. также

History

MimeType

JavaScript 1.1; не поддерживается в IE

представляет тип данных MIME **Object→MimeType**

Синтаксис

```
navigator.mimeTypes[i]
navigator.mimeTypes["тип"]
navigator.mimeTypes.length
```

Свойства

description

Строка, доступная только для чтения, которая предоставляет понятное человеку описание (по-английски) типа данных, характеризуемого объектом MimeType.

enabledPlugin

Доступная только для чтения ссылка на объект Plugin, представляющий установленный и включенный модуль расширения, управляющий указанным MIME-типом. Если такого модуля нет, то значение этого свойства равно null.

suffixes

Доступная только для чтения строка, содержащая список суффиксов имен (через запятую, не включая символ «.»), обычно используемых для файлов указанного MIME-типа. Например, для MIME-типа text/html это свойство выглядит так: "html, htm".

type

Доступная только для чтения строка, задающая имя MIME-типа и отличающая данный MIME-тип от всех остальных, например: "text/html" или "image/jpeg". Это свойство описывает общий тип данных и используемый формат данных. Значение свойства type может также выступать в качестве индекса для доступа к элементам в массиве navigator.mimeTypes[].

Описание

Объект MimeType представляет MIME-тип (т. е. формат данных), поддерживаемый веб-броузером. Формат может поддерживаться для встраиваемых данных либо броузером непосредственно, либо через внешнее вспомогательное приложение или модуль расширения. Объекты MimeType являются элементами массива mimeTypes[] объекта Navigator. В IE массив mimeTypes[] всегда пуст и не имеет эквивалентной функциональности.

Порядок использования

Массив navigator.mimeTypes[] может индексироваться по номеру или по имени требуемого MIME-типа (являющегося значением свойства type). Чтобы проверить, какие MIME-типы поддерживаются броузером, можно выполнить цикл по всем числовым индексам массива. Если же надо просто проверить, поддерживается ли определенный тип, можно написать следующий код:

```
var show_movie = (navigator.mimeTypes["video/mpeg"] != null);
```

См. также `Navigator`, `Plugin`

MouseEvent

DOM Level 2 Events

информация о событии от мыши **Event→UIEvent→MouseEvent**

Свойства

`readonly boolean altKey`

Признак, показывающий, удерживалась ли нажатой клавиша Alt, когда произошло событие. Свойство определено для всех типов событий от мыши.

`readonly unsigned short button`

Признак, показывающий, какая кнопка мыши изменила состояние во время события `mousedown`, `mouseup` или `click`. Значение 0 обозначает левую кнопку, 2 – правую, 1 – среднюю. Обратите внимание: это свойство определяется, когда кнопка меняет состояние. Оно, например, не позволяет выяснить, была ли нажата кнопка во время события `mousemove`. Кроме того, это свойство не является битовой маской – оно ничего не сообщит, если нажата более чем одна кнопка.

`readonly long clientX`, `clientY`

Координаты X и Y указателя мыши относительно *клиентской области* или окна броузера. Обратите внимание: эти координаты не учитывают прокрутку документа; если событие происходит на верхнем краю окна, свойство `clientY` равно 0 независимо от того, как далеко выполнена прокрутка документа. Эти свойства определены для всех типов событий от мыши.

`readonly boolean ctrlKey`

Признак, показывающий, удерживалась ли клавиша Ctrl, когда произошло событие. Свойство определено для всех типов событий от мыши.

`readonly boolean metaKey`

Признак, показывающий, удерживалась ли клавиша Meta, когда произошло событие. Свойство определено для всех типов событий от мыши.

`readonly EventTarget relatedTarget`

Ссылается на узел, который имеет отношение к целевому узлу события. Для событий `mouseover` это узел, от которого отходит указатель мыши при наведении на целевой узел. Для событий `mouseout` это узел, на который наводится указатель мыши, когда он отходит от целевого узла. Для других типов событий мыши значение `relatedTarget` не определено.

`readonly long screenX`, `screenY`

Координаты X и Y указателя мыши относительно верхнего левого угла монитора пользователя. Эти свойства определены для всех типов событий от мыши.

`readonly boolean shiftKey`

Признак, показывающий, удерживалась ли клавиша Shift, когда произошло событие. Свойство определено для всех типов событий от мыши.

Методы

`initMouseEvent()`

Инициализирует свойства вновь созданного объекта `MouseEvent`.

Описание

Этот интерфейс определяет тип объекта Event, который передается событиям типа click, mousedown, mousemove, mouseout, mouseover и mouseup. Обратите внимание: в дополнение к перечисленным здесь свойствам этот интерфейс наследует свойства интерфейсов UIEvent и Event.

См. также Event, UIEvent; глава 17

MouseEvent.initMouseEvent() DOM Level 2 Events

инициализирует свойства объект MouseEvent

Синтаксис

```
void initMouseEvent(String typeArg,
                    boolean canBubbleArg,
                    boolean cancelableArg,
                    AbstractView viewArg,
                    long detailArg,
                    long screenXArg,
                    long screenYArg,
                    long clientXArg,
                    long clientYArg,
                    boolean ctrlKeyArg,
                    boolean altKeyArg,
                    boolean shiftKeyArg,
                    boolean metaKeyArg,
                    unsigned short buttonArg,
                    EventTarget relatedTargetArg);
```

Аргументы

Многочисленные аргументы метода задают начальные значения свойств данного объекта MouseEvent, включая свойства, унаследованные от интерфейсов Event и UIEvent. Имя каждого аргумента ясно указывает на свойство, для которого указано значение, поэтому здесь они отдельно не перечисляются.

Описание

Этот метод инициализирует различные свойства вновь созданного объекта MouseEvent. Он может вызываться только для объекта MouseEvent, созданного методом Document.createEvent(), и только до того, как этот объект будет передан методу EventTarget.dispatchEvent().

См. также Document.createEvent(), Event.initEvent(), UIEvent.initUIEvent()

Navigator JavaScript 1.0

информация об используемом броузере **Object→Navigator**

Синтаксис

```
navigator
```

Свойства

appCodeName

Доступная для чтения и записи строка, задающая кодовое имя броузера. Во всех броузерах, созданных на основе Netscape (Netscape, Mozilla, Firefox), она равна "Mozilla". Для совместимости в броузерах от Microsoft она также равна "Mozilla".

appName

Доступное только для чтения свойство, задающее имя броузера. Для броузеров, созданных на основе Netscape, значение этого свойства равно "Netscape". В Internet Explorer оно равно "Microsoft Internet Explorer". Другие броузеры могут также корректно идентифицировать себя или симулировать другие броузеры с целью обеспечения совместимости.

appVersion

Доступная только для чтения строка, задающая информацию о версии и платформе броузера. Первая часть этой строки представляет собой номер версии. Чтобы получить только последний номер версии, передайте строку в метод parseInt(), а чтобы получить последний и первый номера версий – в метод parseFloat(). Остальная часть строкового значения этого свойства предоставляет другие сведения о версии броузера, включая операционную систему, на которой он работает. Однако, к сожалению, формат этой информации в разных броузерах разный.

cookieEnabled

Доступное только для чтения логическое значение, равное true, если в броузере включен режим использования cookie-файлов, и false, если выключен.

mimeTypes[]

Массив объектов MimeType, каждый из которых представляет один из MIME-типов (например, text/html и image/gif), поддерживаемый броузером. Этот массив может индексироваться числами или именами MIME-типов. Массив mimeTypes[] определен в Internet Explorer, но всегда пуст, т. к. IE не поддерживает объект MimeType.

platform

Доступная только для чтения строка, задающая операционную систему и/или аппаратную платформу, на которой запущен броузер. Стандартный набор значений для этого свойства не определен, но вот некоторые типичные значения: "Win32", "MacPPC" и "Linux i586".

plugins[]

Массив объектов Plugin, каждый из которых представляет один модуль расширения, установленный в броузере. Объект Plugin предоставляет информацию о подключаемом модуле, в том числе список поддерживаемых им MIME-типов.

Массив plugins[] определен в Internet Explorer, но всегда пуст, т. к. IE не поддерживает объект Plugin.

userAgent

Доступная только для чтения строка, задающая значение, которое броузер подставляет в заголовок user-agent в HTTP-запросах. Обычно оно равно значению свойства navigator.appCodeName, за которым следует символ слэша и значение свойства navigator.appVersion. Например:

```
Mozilla/4.0 (compatible; MSIE 4.01; Windows 95)
```

Функции

`navigator.javaEnabled()`

Проверяет, поддерживается ли Java и включен ли режим поддержки Java в текущем броузере.

Описание

Объект `Navigator` содержит свойства, описывающие используемый веб-броузер. Посредством этих свойств можно выполнить специфическую для платформы настройку сценария. Имя этого объекта, очевидно, ссылается на броузер Netscape Navigator, но другие броузеры, поддерживающие JavaScript, поддерживают и этот объект. Имеется только один экземпляр объекта `Navigator`, к которому можно обращаться через свойство `navigator` любого объекта `Window`.

Исторически объект `Navigator` использовался для «опознавания клиента», позволяя выбирать, какой фрагмент программного кода исполнять в зависимости от типа броузера. В примере 14.3 демонстрируется простейший прием, а в сопутствующем ему описании упоминается масса ловушек, которые имеются в объекте `Navigator`. Наилучший способ проверки совместимости, не зависящий от типа броузера, описывается в разделе 13.6.3.

См. также `MimeType`, `Plugin`

Navigator.javaEnabled() JavaScript 1.1

проверяет, включен ли режим поддержки Java

Синтаксис

`navigator.javaEnabled()`

Возвращаемое значение

Значение `true`, если режим поддержки Java в текущем броузере включен, и `false`, если нет.

Описание

Метод `navigator.javaEnabled()` позволяет проверить, включен ли режим поддержки Java в текущем броузере и, следовательно, поддерживает ли он Java-апплеты.

Node DOM Level 1 Core

узел в дереве документа

Подынтерфейсы

`Attr`, `CDATASection`, `CharacterData`, `Comment`, `Document`, `DocumentFragment`, `DocumentType`, `Element`, `ProcessingInstruction`, `Text`.

Константы

Все объекты `Node` реализуют один из перечисленных подынтерфейсов. В любом объекте `Node` имеется свойство `nodeType`, определяющее, какой из подынтерфейсов этот объект реализует. Соответствующие константы представляют собой допустимые значения данного свойства; их имена понятны без объяснений. Обратите внимание: это статические свойства функции-конструктора `Node()`; они не являются свойствами от-

дельных объектов Node. Кроме того, они не поддерживаются в Internet Explorer. Для совместимости с IE необходимо использовать непосредственно числовые литералы. Например, вместо Node.ELEMENT_NODE указывать число 1:

```
Node.ELEMENT_NODE = 1;                // Element
Node.ATTRIBUTE_NODE = 2;              // Attr
Node.TEXT_NODE = 3;                   // Text
Node.CDATA_SECTION_NODE = 4;          // CDATASection
Node.PROCESSING_INSTRUCTION_NODE = 7; // ProcessingInstruction
Node.COMMENT_NODE = 8;                // Comment
Node.DOCUMENT_NODE = 9;               // Document
Node.DOCUMENT_TYPE_NODE = 10;         // DocumentType
Node.DOCUMENT_FRAGMENT_NODE = 11;     // DocumentFragment
```

Свойства

`readonly Attr[] attributes`

Если данный узел является элементом, это свойство задает атрибуты этого элемента. Здесь attributes – это объект, подобный массиву, в котором хранятся узлы Attr, представляющие атрибуты элемента. Обратите внимание: возвращаемый массив «живой» – любые изменения в атрибутах элемента немедленно отражаются через него.

Формально массив attributes[] представлен объектом NamedNodeMap. Интерфейс NamedNodeMap задается стандартом DOM Level 1 и определяет множество методов чтения, записи и удаления элементов. Интерфейс Element определяет намного лучшие методы чтения и записи значений атрибутов, сам интерфейс NamedNodeMap не имеет иного прикладного значения в клиентском JavaScript-коде. По этим причинам описание интерфейса NamedNodeMap не приводится в данной книге. Массив attributes[] следует интерпретировать как доступный только для чтения массив объектов Attr, а для чтения, изменения значений и удаления атрибутов нужно использовать методы, определяемые интерфейсом Element.

`readonly Node[] childNodes`

Содержит дочерние узлы текущего узла. Это свойство никогда не должно быть равно null: для узлов, не имеющих дочерних узлов, childNodes – это массив со свойством length, равным нулю. Это свойство формально представляет собой объект NodeList, но ведет себя как массив объектов Node, доступный только для чтения. Обратите внимание: объект NodeList «живой», т. е. любое изменение в списке дочерних узлов элемента немедленно становится видимым через NodeList.

`readonly Node firstChild`

Первый дочерний узел этого узла или null, если узел не имеет дочерних узлов.

`readonly Node lastChild`

Последний дочерний узел данного узла или null, если у него нет дочерних узлов.

`readonly String localName` *[DOM Level 2]*

В XML-документах, использующих пространства имен, задает локальную часть имени элемента или атрибута. Это свойство никогда не используется при работе с HTML-документами. См. также описание свойств namespaceURI и prefix.

`readonly String namespaceURI` *[DOM Level 2]*

В XML-документах, использующих пространства имен, задает URI пространства имен узла Element или Attr. Это свойство никогда не применяется при работе с HTML-документами. См. также описание свойств localName и prefix.

`readonly Node nextSibling`

Смежный узел, непосредственно следующий за данным узлом в массиве `childNodes[]` узла `parentNode`, или `null`, если такого узла нет.

`readonly String nodeName`

Имя узла. Для узлов `Element` задает имя тега элемента, которое может быть также получено с помощью свойства `tagName` интерфейса `Element`. Для других типов узлов значение зависит от типа. Подробности см. в таблице, приведенной в подразделе «Описание».

`readonly unsigned short nodeType`

Тип узла, т. е. информация о том, какой подынтерфейс реализует узел. Допустимые значения определяются перечисленными ранее константами. Однако т. к. эти константы не поддерживаются в Internet Explorer, вместо них могут использоваться жестко закодированные значения. В HTML-документах распространенные значения для этого свойства таковы: 1 – для узлов `Element`, 3 – для узлов `Text`, 8 – для узлов `Comment` и 9 – для единственного узла `Document` верхнего уровня.

`String nodeValue`

Значение узла. Для узлов `Text` содержит текстовое содержимое. Для других типов узлов значение зависит от свойства `nodeType`, как показано в таблице, приведенной в разделе «Описание».

`readonly Document ownerDocument`

Объект `Document`, частью которого является узел. Для узлов `Document` это свойство равно `null`.

`readonly Node parentNode`

Родительский узел (или узел-контейнер) этого узла или `null`, если родительского узла не существует. Обратите внимание: узлы `Document` и `Attr` никогда не имеют родительских узлов. Кроме того, в узлах, удаленных из документа, а также в только что созданных, но еще не вставленных в дерево документа узлах, свойство `parentNode` равно `null`.

`String prefix` ***[DOM Level 2]***

Для XML-документов, использующих пространства имен, задает префикс пространства имен узла `Element` или `Attr`. Это свойство никогда не применяется при работе с HTML-документами. См. также описание свойств `localName` и `namespaceURL`.

`readonly Node previousSibling`

Смежный узел, непосредственно предшествующий данному узлу в массиве `childNodes[]` родительского узла `parentNode`, или `null`, если такого узла нет.

`readonly String xml` ***[только в IE]***

Если узел представляет XML-документ или элемент XML-документа, это свойство, определенное только в IE, содержит текст элемента или документа в виде строки. Сравните это свойство со свойством `innerHTML` объекта `HTMLElement`. В качестве кросс-платформенной альтернативы обратите внимание на объект `XMLSerializer`.

Методы

`appendChild()`

Добавляет узел в дерево документа, дописывая его к массиву `childNodes[]` этого узла. Если узел уже находится в дереве документа, он удаляется и вставляется в новую позицию.

`cloneNode()`

Создает копию данного узла или узла и его потомков.

`hasAttributes()` *[DOM Level 2]*

Возвращает `true`, если узел является узлом `Element` и имеет какие-либо атрибуты.

`hasChildNodes()`

Возвращает `true`, если в этом узле имеются какие-либо дочерние элементы.

`insertBefore()`

Вставляет узел в дерево документа непосредственно перед указанным дочерним узлом данного узла. Если вставляемый узел уже имеется в дереве, он удаляется и вставляется в новую позицию.

`isSupported()` *[DOM Level 2]*

Возвращает `true`, если указанный номер версии модуля поддерживается данным узлом.

`normalize()`

«Нормализует» все текстовые узлы-потомки этого узла путем удаления пустых узлов `Text` и объединения смежных узлов `Text`.

`removeChild()`

Удаляет (и возвращает) указанный дочерний узел из дерева документа.

`replaceChild()`

Удаляет (и возвращает) указанный дочерний узел из дерева документа, заменяя его другим узлом.

`selectNodes()` *[только в IE]*

Данный метод, реализованный только в IE, выполняет XPath-запрос, используя указанный узел в качестве корня, и возвращает результат в виде объекта `NodeList`. Альтернативные способы, реализованные в соответствии со стандартами DOM, описываются в справочных статьях о методах `Document.evaluate()` и `Document.createExpression()`.

`selectSingleNode()` *[только в IE]*

Данный метод, реализованный только в IE, выполняет XPath-запрос, используя данный узел в качестве корня, и возвращает результат в виде единственного узла. Альтернативные способы, реализованные в соответствии со стандартами DOM, описываются в справочных статьях о методах `Document.evaluate()` и `Document.createExpression()`.

`transformNode()` *[только в IE]*

Данный метод, реализованный только в IE, применяет таблицу XSLT-стилей к данному узлу и возвращает строковый результат. Альтернативный вариант, реализованный в броузерах, отличных от IE, описывается в справочной статье об объекте `XSLTProcessor`.

`transformNodeToObject()` *[только в IE]*

Данный метод, реализованный только в IE, применяет таблицу XSLT-стилей к данному узлу и возвращает результат в виде объекта `Document`. Альтернативный вариант, реализованный в броузерах, отличных от IE, описывается в справочной статье об объекте `XSLTProcessor`.

Описание

Все объекты в дереве документа (включая сам объект Document) реализуют интерфейс Node, который предоставляет фундаментальные свойства и узлы для обхода дерева и манипуляции деревом. (В Internet Explorer интерфейс Node определяет некоторые дополнительные свойства и методы для работы с XML-документами, XPath-выражениями и XSLT-преобразованиями. Подробности см. в главе 21.)

Свойство parentNode и массив childNodes[] позволяют передвигаться вверх и вниз по дереву документа. Можно перечислить дочерние узлы данного узла, выполнив цикл по элементам childNodes[] или используя свойства firstChild и nextSibling (или свойства lastChild и previousSibling для обхода в обратном порядке). Методы appendChild(), insertBefore(), removeChild() и replaceChild() позволяют модифицировать дерево документа, изменяя дочерние узлы данного узла.

Каждый объект в дереве документа реализует как интерфейс Node, так и более специализированный интерфейс, такой как Element или Text. Свойство nodeType указывает, какой подынтерфейс реализует узел. Это свойство позволяет проверить тип узла перед тем, как использовать свойства и методы более специализированного интерфейса. Например:

```
var n;                        // Содержит узел, с которым мы работаем
if (n.nodeType == 1) {        // Сравнить с константой Node.ELEMENT_NODE
    var tagname = n.tagName; // Если узел является узлом Element, это имя тега
}
```

Свойства nodeName и nodeValue задают дополнительную информацию об узле, но, как показано в следующей таблице, их значения зависят от nodeType. Обратите внимание: подынтерфейсы обычно определяют специальные свойства (такие как свойство tagName для узлов Element или свойство data для узлов Text) при получении этой информации:

nodeType	nodeName	nodeValue
ELEMENT_NODE	Имя тега элемента	null
ATTRIBUTE_NODE	Имя атрибута	Значение атрибута
TEXT_NODE	#text	Текст узла
CDATA_SECTION_NODE	#cdata-section	Текст узла
PROCESSING_INSTRUCTION_NODE	Цель исполняемой инструкции	Остальная часть исполняемой инструкции
COMMENT_NODE	#comment	Текст комментария
DOCUMENT_NODE	#document	null
DOCUMENT_TYPE_NODE	Имя типа документа	null
DOCUMENT_FRAGMENT_NODE	#document-fragment	null

См. также

Document, Element, Text, XMLSerializer, XPathExpression, XSLTProcessor; глава 15

Node.appendChild() DOM Level 1 Core

вставляет узел в качестве последнего дочернего узла данного узла

Синтаксис

```
Node appendChild(Node новыйДочернийУзел)
    throws DOMException;
```

Аргументы

новыйДочернийУзел

Узел, который должен быть вставлен в документ. Если это узел DocumentFragment, он не вставляется непосредственно, вставляются его дочерние узлы.

Возвращаемое значение

Добавленный узел.

Исключения

Этот метод может генерировать исключение DOMException с одним из перечисленных далее кодов в следующих ситуациях:

HIERARCHY_REQUEST_ERR

Узел не допускает наличия дочерних узлов или дочерних узлов указанного типа либо узел *новыйДочернийУзел* является предком данного узла (или им самим).

WRONG_DOCUMENT_ERR

Свойство ownerDocument узла *новыйДочернийУзел* не совпадает со свойством ownerDocument данного узла.

NO_MODIFICATION_ALLOWED_ERR

Этот узел доступен только для чтения и не допускает добавления дочерних узлов либо добавляемый узел уже является частью дерева документа, а его родительский узел доступен только для чтения и не допускает удаления дочерних узлов.

Описание

Этот метод добавляет узел *новыйДочернийУзел* в документ, вставляя его в качестве последнего дочернего узла. Если узел *новыйДочернийУзел* уже присутствует в дереве документа, он удаляется из дерева и вставляется в новое место. Если узел *новыйДочернийУзел* является узлом DocumentFragment, сам узел не вставляется, а вместо этого в конец массива childNodes[] данного узла вставляются по порядку все дочерние узлы DocumentFragment. Обратите внимание, что узел из другого документа (или созданный другим документом) не может быть вставлен в текущий документ. То есть свойство ownerDocument узла *новыйДочернийУзел* должно совпадать со свойством ownerDocument данного узла.

Пример

Следующая функция вставляет новый абзац в конец документа:

```
function appendMessage(message) {
    var pElement = document.createElement("p");
    var messageNode = document.createTextNode(message);
    pElement.appendChild(messageNode);   // Добавляет текст к абзацу
    document.body.appendChild(pElement); // Добавляет абзац к телу документа
}
```

См. также Node.insertBefore(), Node.removeChild(), Node.replaceChild()

Node.cloneNode() DOM Level 1 Core

дублирует узел и (необязательно) всех его потомков

Синтаксис

```
Node cloneNode(boolean глубина);
```

Аргументы

глубина

Если этот аргумент равен `true`, метод `cloneNode()` рекурсивно «клонирует» всех потомков данного узла. В противном случае копируется только данный узел.

Возвращаемое значение

Копия узла.

Описание

Метод `cloneNode()` создает копию узла, для которого он вызван, и возвращает ее. Если ему передается аргумент `true`, он также рекурсивно копирует всех потомков узла. В противном случае он копирует только данный узел, но не его дочерние узлы. Возвращаемый узел не является частью дерева документа, а его свойство `parentNode` равно `null`. Когда копируется узел `Element`, то копируются и все его атрибуты. Однако следует отметить, что функции-обработчики событий, зарегистрированные для узла, не копируются.

Node.hasAttributes() DOM Level 2 Core

определяет, есть ли в узле атрибуты

Синтаксис

```
boolean hasAttributes();
```

Возвращаемое значение

Значение `true`, если в узле имеется один или несколько атрибутов, и `false`, если их нет. Обратите внимание: только узлы `Element` имеют атрибуты.

См. также `Element.getAttribute()`, `Element.hasAttribute()`, `Node`

Node.hasChildNodes() DOM Level 1 Core

определяет, имеются ли в узле дочерние узлы

Синтаксис

```
boolean hasChildNodes();
```

Возвращаемое значение

Значение `true`, если в этом узле есть один или более дочерних узлов, и `false`, если их нет.

Node.insertBefore() DOM Level 1 Core

вставляет узел в дерево документа перед указанным узлом

Синтаксис

```
Node insertBefore(Node новыйДочернийУзел,
                  Node дочернийУзел)
    throws DOMException;
```

Аргументы

новыйДочернийУзел

Узел, который должен быть вставлен в дерево. Если это узел `DocumentFragment`, вместо него вставляются его дочерние узлы.

дочернийУзел

Дочерний узел данного узла, перед которым должен быть вставлен узел *новыйДочернийУзел*. Если этот аргумент равен `null`, узел *новыйДочернийУзел* вставляется в качестве последнего дочернего узла данного.

Возвращаемое значение

Вставленный узел.

Исключения

Этот метод может генерировать исключение `DOMException` со следующими значениями `code`:

`HIERARCHY_REQUEST_ERR`

Узел не поддерживает дочерние узлы или не допускает наличия дочерних узлов указанного типа, либо узел *новыйДочернийУзел* является предком данного узла (или самим узлом).

`WRONG_DOCUMENT_ERR`

Свойство `ownerDocument` узла *новыйДочернийУзел* и данного узла не совпадают.

`NO_MODIFICATION_ALLOWED_ERR`

Узел доступен только для чтения и не допускает вставки или родительский узел узла *новыйДочернийУзел* доступен только для чтения и не допускает удаления.

`NOT_FOUND_ERR`

Узел *дочернийУзел* не является дочерним для данного узла.

Описание

Этот метод вставляет узел *новыйДочернийУзел* в дерево документа в качестве дочернего узла данного. Новый узел позиционируется в массиве `childNodes[]` так, что оказывается непосредственно перед узлом *дочернийУзел*. Если узел *дочернийУзел* равен *null*, *новыйДочернийУзел* вставляется в конец `childNodes[]` точно так же, как в случае вызова метода `appendChild()`. Обратите внимание: не допускается вызов этого метода с аргументом *дочернийУзел*, не являющимся дочерним узлом данного.

Если *новыйДочернийУзел* уже присутствует в дереве документа, он удаляется из дерева и затем вставляется заново в новую позицию. Если *новыйДочернийУзел* является узлом `DocumentFragment`, он не вставляется сам; вместо этого в указанную позицию вставляются по порядку все его дочерние узлы.

Пример

Следующая функция вставляет новый абзац в начало документа:

```
function insertMessage(message) {
    var paragraph = document.createElement("p"); // Создаем элемент <p>
    var text = document.createTextNode(message); // Создаем узел Text
    paragraph.appendChild(text);                 // Добавляем текст в абзац
    // Теперь вставляем абзац перед первым дочерним узлом тела документа
    document.body.insertBefore(paragraph, document.body.firstChild)
}
```

См. также Node.appendChild(), Node.removeChild(), Node.replaceChild()

Node.isSupported() DOM Level 2 Core

определяет, поддерживает ли узел данный модуль

Синтаксис

```
boolean isSupported(String модуль,
                    String версия);
```

Аргументы

модуль

Имя проверяемого модуля.

версия

Номер версии проверяемого модуля или пустая строка (проверка поддержки любой версии модуля).

Возвращаемое значение

Значение true, если узел поддерживает указанную версию указанного модуля, или false, если не поддерживает.

Описание

Стандарт W3C DOM имеет модульную структуру, и реализации не обязаны поддерживать все модули или возможности стандарта. Этот метод проверяет, поддерживает ли реализация узла указанную версию модуля с указанным именем. Список значений аргументов *модуль* и *версия* приведен в справочной статье о методе DOMImplementation.hasFeature().

См. также DOMImplementation.hasFeature()

Node.normalize() DOM Level 1 Core

объединяет смежные узлы Text и удаляет пустые

Синтаксис

```
void normalize();
```

Описание

Данный метод обходит всех потомков узла и «нормализует» документ, удаляя любые пустые узлы Text и собирая все смежные узлы Text в один узел. Это иногда позволяет упростить структуру дерева после вставки и удаления узлов.

См. также Text

Node.removeChild() DOM Level 1 Core

удаляет (и возвращает) указанный дочерний узел данного узла

Синтаксис

```
Node removeChild(Node старыйДочернийУзел)
    throws DOMException;
```

Аргументы

старыйДочернийУзел Удаляемый дочерний узел.

Возвращаемое значение

Удаленный узел.

Исключения

Этот метод может генерировать исключение DOMException с одним из перечисленных далее кодов в следующих ситуациях:

NO_MODIFICATION_ALLOWED_ERR

Этот узел доступен только для чтения и не допускает удаления дочерних узлов.

NOT_FOUND_ERR

Узел *старыйДочернийУзел* не является дочерним для данного узла.

Описание

Этот метод удаляет указанный дочерний узел из массива childNodes[] данного узла. Вызов этого метода с узлом, не являющимся дочерним, будет ошибкой. Метод removeChild() возвращает *старыйДочернийУзел* после его удаления. *старыйДочернийУзел* продолжает быть действительным узлом и может быть позднее вставлен в документ.

Пример

Последний дочерний узел можно удалить из тела документа так:

```
document.body.removeChild(document.body.lastChild);
```

См. также Node.appendChild(), Node.insertBefore(), Node.replaceChild()

Node.replaceChild() DOM Level 1 Core

заменяет дочерний узел новым узлом

Синтаксис

```
Node replaceChild(Node новыйДочернийУзел,
                  Node старыйДочернийУзел)
    throws DOMException;
```

Аргументы

новыйДочернийУзел

Узел для замены.

старыйДочернийУзел

Заменяемый узел.

Возвращаемое значение

Узел, который был удален из документа и заменен.

Исключения

Этот метод может генерировать исключение DOMException со следующими значениями code:

HIERARCHY_REQUEST_ERR

Узел не допускает наличия дочерних узлов или дочерних узлов указанного типа либо *новыйДочернийУзел* является предком данного узла (или им самим).

WRONG_DOCUMENT_ERR

Свойство ownerDocument узла *новыйДочернийУзел* не совпадает со свойством ownerDocument данного узла.

NO_MODIFICATION_ALLOWED_ERR

Данный узел доступен только для чтения и не допускает замены или *новыйДочернийУзел* является дочерним для данного узла и не допускает удаления.

NOT_FOUND_ERR

Узел *старыйДочернийУзел* не является дочерним для данного узла.

Описание

Этот метод заменяет один узел в дереве документа другим. *старыйДочернийУзел* – это узел, который должен быть заменен, и он должен быть дочерним для данного узла. *новыйДочернийУзел* – это узел, который займет его место в массиве childNodes[] данного узла.

Если *новыйДочернийУзел* уже является частью документа, то он сначала удаляется из документа перед повторной вставкой в новую позицию. Если *новыйДочернийУзел* – это узел DocumentFragment, сам узел не вставляется; вместо этого в позицию, ранее занятую узлом *старыйДочернийУзел*, по порядку вставляются все его дочерние узлы DocumentFragment.

Пример

Следующий код заменяет узел n элементом <b>, а затем вставляет заменяемый узел в элемент <b>, в результате чего этот узел отображается жирным шрифтом:

```
// Получаем первый дочерний узел первого абзаца документа
var n = document.getElementsByTagName("p")[0].firstChild;
var b = document.createElement("b"); // Создает элемент <b>
n.parentNode.replaceChild(b, n);     // Заменяет узел узлом <b>
b.appendChild(n);                    // Вставляет узел как дочерний по отношению к <b>
```

См. также Node.appendChild(), Node.insertBefore(), Node.removeChild()

Node.selectNodes() IE 6

выбирает узлы с помощью XPath-выражения

Синтаксис

NodeList selectNodes(String *запрос*)

Аргументы

запрос Строка XPath-запроса.

Возвращаемое значение

Объект `NodeList`, содержащий узлы, соответствующие запросу.

Описание

Этот метод, реализованный только в IE, вычисляет XPath-выражение, используя данный узел в качестве корневого узла запроса, и возвращает результат в виде объекта `NodeList`. Метод `selectNodes()` может присутствовать только в XML-документах, но не в HTML-документах. Обратите внимание: поскольку объект `Document` также является узлом, данный метод может использоваться для работы с целым XML-документом.

Альтернативный способ, не зависящий от типа броузера, приводится в справочной статье о методе `Document.evaluate()`.

См. также `Document.evaluate()`, `XPathExpression`; глава 21

Node.selectSingleNode() IE 6

отыскивает узел, совпадающий с XPath-запросом

Синтаксис

```
Node selectSingleNode(String запрос)
```

Аргументы

запрос Строка XPath-запроса.

Возвращаемое значение

Единственный узел `Node`, совпадающий с *запросом*, или `null`, если нет ни одного узла, соответствующего запросу.

Описание

Этот метод, реализованный только в IE, вычисляет XPath-выражение, используя данный узел в качестве корневого узла запроса. Возвращает первый найденный узел или `null`, если нет ни одного узла, соответствующего запросу. Метод `selectSingleNode()` может присутствовать только в XML-документах, но не в HTML-документах. Обратите внимание: поскольку объект `Document` также является узлом, данный метод может использоваться для работы с целым XML-документом.

Альтернативный способ, не зависящий от типа броузера, приводится в справочной статье о методе `Document.evaluate()`.

См. также `Document.evaluate()`, `XPathExpression`; глава 21

Node.transformNode() IE 6

преобразует узел в строку средствами XSLT

Синтаксис

```
String transformNode(Document xslt)
```

Аргументы

xslt Таблица стилей XSLT, преобразованная в объект Document.

Возвращаемое значение

Текст, полученный в результате применения указанной таблицы стилей к данному узлу и его потомкам.

Описание

Этот метод, реализованный только в IE, выполняет преобразование узла и его потомков в соответствии с правилами из таблицы стилей XSLT и возвращает результат в виде неразобранной строки. Метод transformNode() может присутствовать только в XML-документах, но не в HTML-документах. Обратите внимание: поскольку объект Document также является узлом, данный метод может использоваться для работы с целым XML-документом.

В других броузерах аналогичная функциональность предоставляется объектом XSLTProcessor.

См. также XSLTProcessor, Node.transformNodeToObject(); глава 21

Node.transformNodeToObject() IE 6

преобразует узел в документ средствами XSLT

Синтаксис

```
Document transformNode(Document xslt)
```

Аргументы

xslt Таблица стилей XSLT, преобразованная в объект Document.

Возвращаемое значение

Результат преобразования в виде объекта Document.

Описание

Этот метод, реализованный только в IE, выполняет преобразование узла и его потомков в соответствии с правилами из таблицы XSLT-стилей и возвращает результат в виде объекта Document. Метод transformNodeToObject() может присутствовать только в XML-документах, но не в HTML-документах. Обратите внимание: поскольку объект Document также является узлом, данный метод может использоваться для работы с целым XML-документом.

В других броузерах аналогичная функциональность предоставляется объектом XSLTProcessor.

См. также XSLTProcessor, Node.transformNodeToObject(); глава 21

NodeList DOM Level 1 Core

массив узлов, доступный только для чтения **Object→NodeList**

Свойства

```
readonly unsigned long length
```

Количество узлов в массиве.

Методы

item()

Возвращает указанный элемент массива.

Описание

Интерфейс NodeList определяет доступный только для чтения упорядоченный список (т. е. массив) объектов Node. Свойство length указывает, сколько узлов находится в списке, а метод item() позволяет получить узел в указанной позиции в списке. Элементы NodeList всегда являются корректными объектами Node: объект NodeList никогда не содержит пустых (null) элементов.

В JavaScript объекты NodeList ведут себя так же, как JavaScript-массивы, и элемент из списка можно получить, используя нотацию массивов с квадратными скобками вместо вызова метода item(). Однако присваивать новые узлы NodeList с помощью квадратных скобок нельзя. Поскольку всегда проще рассматривать объект NodeList как доступный только для чтения JavaScript-массив, в этой книге вместо NodeList используется нотация Element[] или Node[] (т. е. массив объектов Element или Node). Методы Document.getElementsByTagName(), Element.getElementsByTagName() и HTMLDocument.getElementsByName() описываются в данной книге как возвращающие Element[], а не NodeList. Аналогично свойство childNodes объекта Node формально является объектом NodeList, но в справочной статье об объекте Node оно определяется как Node[], а само свойство обычно упоминается как «массив объектов childNodes[]».

Обратите внимание: объекты NodeList «живые»: они немедленно отражают изменения в дереве документа. Например, если NodeList представляет дочерние узлы для указанного узла, и вы удалите один из этих дочерних узлов, он будет удален и из вашего объекта NodeList. Будьте аккуратны при выполнении цикла по элементам NodeList, если тело цикла вносит изменения в дерево документа (например, удаляет узлы), которые могут влиять на содержимое NodeList!

См. также Document, Element

NodeList.item() DOM Level 1 Core

получает элемент из NodeList

Синтаксис

```
Node item(unsigned long индекс);
```

Аргументы

индекс

Позиция (или индекс) нужного узла в NodeList. Индекс первого узла в NodeList равен 0, индекс последнего узла равен length–1.

Возвращаемое значение

Узел в указанной позиции в NodeList или null, если *индекс* меньше нуля или больше либо равен длине NodeList.

Описание

Этот метод возвращает указанный элемент NodeList. В JavaScript вместо вызова item() можно использовать нотацию массивов с квадратными скобками.

Option

DOM Level 2 HTML

вариант выбора в теге Select **Node→Element→HTMLElement→HTMLOptionElement**

Конструктор

Объекты `Option` могут создаваться с помощью метода `Document.createElement()`, как и любой другой тег. Кроме того, в DOM Level 0 объекты `Option` могут создаваться динамически с помощью конструктора `Option()`:

```
new Option(String текст, String значение,
           boolean выбранПоУмолчанию, boolean выбран)
```

Аргументы

текст

Необязательный строковый аргумент, задающий значение свойства `text` объекта `Option`.

значение

Необязательный строковый аргумент, задающий значение свойства `value` объекта `Option`.

выбранПоУмолчанию

Необязательный логический аргумент, задающий значение свойства `defaultSelected` объекта `Option`.

выбран

Необязательный логический аргумент, задающий значение свойства `selected` объекта `Option`.

Свойства

`boolean defaultSelected`

Начальное значение атрибута `selected` элемента `<option>`. Это значение используется для возвращения свойства `selected` в его начальное состояние при сбросе формы. При установке этого свойства также происходит установка свойства `selected`.

`boolean disabled`

Значение `true` означает, что элемент `<option>` недоступен и пользователь не сможет его выбрать. Соответствует атрибуту `disabled`.

`readonly HTMLFormElement form`

Ссылка на элемент `<form>`, содержащий данный элемент.

`readonly long index`

Позиция данного элемента `<option>` в содержащем его элементе `<select>`.

`String label`

Текст, отображаемый элементом. Соответствует атрибуту `label`. Если значение этого свойства не определено, вместо него отображается текст, содержащийся в элементе `<option>`.

`boolean selected`

Текущее состояние элемента: если `true` — элемент выбран. Начальное значение этого свойства соответствует значению атрибута `selected`.

`readonly String text`

Простой текст, содержащийся в элементе `<option>`. Этот текст используется в качестве метки при выводе элемента.

`String value`

Значение, отправляемое серверу вместе с данными формы. Соответствует атрибуту `value`.

Синтаксис HTML

Объект `Option` создается с помощью тега `<option>` внутри тега `<select>`, находящегося внутри `<form>`. В теге `<select>` обычно присутствуют несколько тегов `<option>`:

```
<form ...>
<select ...>
<option
  [ value="значение" ] // Значение, возвращаемое при передаче формы
  [ selected ] >       // Указывает, выбран ли этот элемент первоначально
  текстовая_метка      // Текст, отображаемый для этого элемента
[ </option> ]
  ...
</select>
  ...
</form>
```

Описание

Объект `Option` описывает один из вариатнов выбора внутри объекта `Select`. Свойства этого объекта определяют, выбран ли вариант по умолчанию, выбран ли он в данный момент, а также задают позицию, которую он занимает в массиве `options[]` содержащего его объекта `Select`, отображаемый им текст и значение, которое он передает на сервер при передаче данных родительской формы.

Обратите внимание: хотя текст, выводимый этим элементом, задается вне тега `<option>`, он должен быть обычным неформатированным текстом без каких-либо HTML-тегов, чтобы нормально отображаться в списках и выпадающих меню, которые не поддерживают HTML-средств форматирования.

Существует возможность динамически создавать новые объекты `Option` для отображения в объекте `Select` с помощью конструктора `Option()`. Создав новый объект `Option`, можно добавить его к списку вариантов выбора в объекте `Select` с помощью метода `Select.add()`. Дополнительные сведения приводятся в справочной статье о свойстве `Select.options[]`.

См. также `Select`, `Select.options[]`; глава 18

Packages

См. описание объекта Packages в части III книги

Password

См. описание объекта Input

Plugin

JavaScript 1.1; не поддерживается в IE

описывает установленный модуль расширения **Object→Plugin**

Синтаксис

```
navigator.plugins[i]
navigator.plugins['имя']
```

Свойства

`description`

Доступная только для чтения строка, которая содержит понятное человеку описание указанного подключаемого модуля. Текст этого описания предоставляется создателями модуля и может содержать информацию о разработчике и версии, а также о функциональном назначении модуля.

`filename`

Доступная только для чтения строка, задающая имя файла на диске, который содержит собственно программу подключаемого модуля. Это имя может отличаться от платформы к платформе. Для идентификации модуля более полезным, чем `filename`, является свойство `name`.

`length`

Каждый объект `Plugin` содержит элементы массива `MimeType`, задающие форматы данных, которые поддерживаются подключаемым модулем. Как и во всех массивах, свойство `length` задает количество элементов в массиве.

`name`

Свойство `name` объекта `Plugin` – это доступная только для чтения строка, задающая имя подключаемого модуля. Каждый модуль должен иметь уникальное имя. Имя может выступать в качестве индекса для обращения к массиву `navigator.plugins[]`. Используя этот факт, нетрудно определить, установлен ли указанный подключаемый модуль в текущем броузере:

```
var flash_installed = (navigator.plugins["Shockwave Flash"] != null);
```

Элементы массива

Элементы массива для объекта `Plugin` – это объекты `MimeType`, задающие форматы данных, которые поддерживаются подключаемым модулем. Свойство `length` указывает количество объектов `MimeType` в массиве.

Описание

Модуль расширения, или *подключаемый модуль* (*plugin*), – это программный модуль, который может вызываться броузером для вывода специализированных типов данных, встраиваемых в окно броузера. Подключаемые модули представлены объектом `Plugin`, а свойство `plugins[]` объекта `Navigator` – это массив объектов `Plugin`, представляющих модули расширения, установленные в броузере. Броузер IE не поддерживает объект `Plugin` и потому в нем массив `navigator.plugins[]` всегда пуст.

Свойство `navigator.plugins[]` может индексироваться числовыми значениями на случай, если возникнет необходимость обойти в цикле все установленные модули в поисках требуемого (например, чтобы отыскать модуль, поддерживающий некоторый MIME-тип данных, встроенных в страницу). Но этот массив точно так же может индексироваться именами модулей. Так, если необходимо проверить, установлен ли

какой-то определенный модуль в броузере у пользователя, сделать это можно примерно таким образом:

```
var flash_installed = (navigator.plugins["Shockwave Flash"] != null);
```

В таких случаях в качестве индекса массива используется имя, которое служит значением свойства `name` объекта `Plugin`.

Этот объект несколько необычен тем, что он имеет как традиционные свойства, так и элементы массива. Свойства объекта `Plugin` предоставляют различную информацию о подключаемом модуле, а его элементы массива – это объекты `MimeType`, задающие форматы встраиваемых данных, которые поддерживает этот модуль. Не запутайтесь: объекты `Plugin` в объекте `Navigator` хранятся в массиве, при этом каждый объект `Plugin` сам является массивом объектов `MimeType`. Поскольку существует два массива, может получиться следующий код:

```
navigator.plugins[i][j]           // j-й MIME-тип в i-м модуле
navigator.plugins["LiveAudio"][0] // Первый MIME-тип модуля LiveAudio
```

И наконец, следует заметить, что хотя элементы массива в объекте `Plugin` задают MIME-типы, поддерживаемые этим подключаемым модулем, с помощью свойства `enabledPlugin` объекта `MimeType` можно также определить, какой модуль поддерживает данный MIME-тип.

См. также `Navigator`, `MimeType`

ProcessingInstruction

DOM Level 1 XML

инструкция обработки в XML-документе **Node→ProcessingInstruction**

Свойства

`String data`

Содержимое инструкции обработки (т. е. от первого непробельного символа после цели до закрывающих символов `?>`, но не включая их).

`readonly String target`

Цель инструкции обработки. Это первый идентификатор инструкции обработки, следующий за открывающими символами `<?`; он задает «обработчик», для которого предназначена инструкция обработки.

Описание

Этот редко используемый интерфейс представляет инструкцию обработки в XML-документе. Программисты, работающие с HTML-документами, никогда не столкнутся с узлом `ProcessingInstruction`.

См. также `Document.createProcessingInstruction()`

Radio

См. описание объекта Input

Range

DOM Level 2 Range

представляет непрерывную область документа **Object→Range**

Константы

Эти константы устанавливают, как должны сравниваться граничные точки двух объектов Range. Они являются допустимыми значениями аргумента *какСравнивать* метода compareBoundaryPoints(). Подробности см. в справочной статье о методе Range.compareBoundaryPoints().

`unsigned short START_TO_START = 0`

Сравнивает начало указанной области с началом данной области.

`unsigned short START_TO_END = 1`

Сравнивает начало указанной области с концом данной области.

`unsigned short END_TO_END = 2`

Сравнивает конец указанной области с концом данной области.

`unsigned short END_TO_START = 3`

Сравнивает конец указанной области с началом данной области.

Свойства

Интерфейс Range определяет следующие свойства. Обратите внимание: все эти свойства доступны только для чтения. Нельзя изменить начальную и конечную точки области путем установки свойств; вместо этого необходимо вызывать методы setEnd() или setStart(). Кроме того, следует отметить, что после вызова метода detach() объекта Range любые последующие попытки прочитать любые из этих свойств генерируют исключение DOMException с кодом INVALID_STATE_ERR.

`readonly boolean collapsed`

Значение true, если начало и конец области находятся в одной точке документа, т. е. область пуста или «сжата».

`readonly Node commonAncestorContainer`

Наиболее глубоко вложенный узел документа, содержащий (т. е. являющийся предком) как начальную, так и конечную точки области.

`readonly Node endContainer`

Узел документа, содержащий конечную точку области.

`readonly long endOffset`

Позиция конечной точки внутри endContainer.

`readonly Node startContainer`

Узел документа, содержащий начальную точку области.

`readonly long startOffset`

Позиция начальной точки области внутри startContainer.

Методы

Далее перечислены методы, определенные интерфейсом Range. Обратите внимание: если метод detach() вызван для области, то любые последующие вызовы любых методов для этой области генерируют исключение DOMException с кодом INVALID_STATE_ERR. Так как это исключение повсеместно встречается в данном интерфейсе, оно не упоминается в справочных статьях об отдельных методах Range:

`cloneContents()`

Возвращает новый объект DocumentFragment, содержащий копию области документа, представляемой данным объектом Range.

`cloneRange()`

Создает новый объект `Range`, представляющий такую же область документа, как данный объект.

`collapse()`

Сжимает область так, что одна граничная точка становится такой же, как другая.

`compareBoundaryPoints()`

Сравнивает граничную точку указанной области с граничной точкой данной области и возвращает -1, 0 или 1 в зависимости от их порядка. Первый аргумент определяет, какие точки должны сравниваться, и его значение должно быть одной из ранее определенных констант.

`deleteContents()`

Удаляет область документа, представляемую данным объектом `Range`.

`detach()`

Сообщает реализации, что эта область больше не будет использоваться, и можно прекратить следить за изменениями в ней. Если вызвать этот метод для области, последующие вызовы или обращения к свойствам этой области будут генерировать исключение `DOMException` с кодом `INVALID_STATE_ERR`.

`extractContents()`

Удаляет область документа, представленную данным объектом `Range`, но возвращает содержимое этой области в виде объекта `DocumentFragment`. Действие этого метода эквивалентно сумме действий методов `cloneContents()` и `deleteContents()`.

`insertNode()`

Вставляет указанный узел в документ в качестве начальной точки области.

`selectNode()`

Устанавливает граничные точки этой области так, чтобы она содержала указанный узел и всех его потомков.

`selectNodeContents()`

Устанавливает граничные точки этой области так, чтобы она содержала всех потомков указанного узла, но не сам узел.

`setEnd()`

Устанавливает конечную точку этой области на указанные узел и смещение.

`setEndAfter()`

Устанавливает конечную точку области непосредственно после указанного узла.

`setEndBefore()`

Устанавливает конечную точку области непосредственно перед указанным узлом.

`setStart()`

Устанавливает начальную позицию области с указанным смещением внутри указанного узла.

`setStartAfter()`

Устанавливает начальную позицию этой области непосредственно после указанного узла.

`setStartBefore()`

Устанавливает начальную позицию этой области непосредственно перед указанным узлом.

`surroundContents()`

Вставляет указанный узел в документ в начальную позицию области и затем меняет родителей всех узлов в области таким образом, чтобы они стали потомками вновь вставленного узла.

`toString()`

Возвращает текстовое содержимое области документа, описываемой данным объектом `Range`.

Описание

Объект `Range` представляет непрерывную область документа, например, область, которую пользователь может выделить перетаскиванием указателя мыши в окне веб-броузера. Если реализация поддерживает модуль `Range` (к моменту написания этих строк объект `Range` полностью поддерживался броузерами Firefox и Opera, Safari поддерживал частично, а Internet Explorer вообще не поддерживал), объект `Document` определяет метод `createRange()`, который можно вызвать для создания нового объекта `Range`. Однако будьте осторожны: Internet Explorer определяет несовместимый метод `Document.createRange()`, возвращающий объект, аналогичный, но не совместимый с интерфейсом `Range`. Интерфейс `Range` определяет несколько методов для задания выделенной области документа и еще несколько методов для реализации операций вставки-замены в выделенной области.

Область имеет две граничные точки: начальную и конечную. Каждая граничная точка задается комбинацией узла и смещения внутри узла. Узел – это обычно `Element`, `Document` или `Text`. Для узлов `Element` и `Document` смещение относится к дочерним элементам узла. Смещение 0 определяет граничную точку перед первым дочерним узлом данного. Смещение 1 определяет граничную точку после первого дочернего узла и перед вторым дочерним узлом. Однако если граничный узел – это узел `Text`, смещение определяет позицию между двумя символами этого текста.

Свойства интерфейса `Range` обеспечивают получение граничных узлов и смещений области. Методы интерфейса предоставляют несколько возможностей установки границ области. Обратите внимание, что границы области могут быть установлены в узлах внутри документа или внутри объекта `DocumentFragment`.

Определив граничные точки области, можно вызывать методы `deleteContents()`, `extractContents()`, `cloneContents()` и `insertNode()` для реализации операций вырезания, копирования и вставки.

Когда документ изменяется путем вставки или удаления узлов, все объекты `Range`, представляющие измененные части документа, по необходимости изменяются, чтобы их граничные точки оставались действующими и представляли (насколько это возможно) то же содержимое документа.

См. также

`Document.createRange()`, `DocumentFragment`

Range.cloneContents()

DOM Level 2 Range

копирует содержимое области в DocumentFragment

Синтаксис

```
DocumentFragment cloneContents()
    throws DOMException;
```

Возвращаемое значение

Объект DocumentFragment, заключающий в себе копию содержимого документа внутри данной области.

Исключения

Если эта область включает узел DocumentType, то данный метод генерирует исключение DOMException с кодом HIERARCHY_REQUEST_ERR.

Описание

Этот метод дублирует содержимое области и возвращает результат в объекте DocumentFragment.

См. также DocumentFragment, Range.deleteContents(), Range.extractContents()

Range.cloneRange() DOM Level 2 Range

создает копию данной области

Синтаксис

```
Range cloneRange();
```

Возвращаемое значение

Новый объект Range, имеющий те же граничные точки, что и данная область.

См. также Document.createRange()

Range.collapse() DOM Level 2 Range

делает одну граничную точку равной другой

Синтаксис

```
void collapse(boolean вНачало)
    throws DOMException;
```

Аргументы

вНачало Если этот аргумент равен true, метод устанавливает конечную точку области равной начальной точке. В противном случае он устанавливает начальную точку равной конечной точке.

Описание

Этот метод устанавливает одну граничную точку области равной другой точке. Модифицируемая точка определяется аргументом *вНачало*. После завершения метода про область говорят, что она «сжатая», т. е. представляет одну точку в документе и не имеет содержимого. Если область сжата подобным образом, свойство collapsed равно true.

Range.compareBoundaryPoints() DOM Level 2 Range

сравнивает позиции двух областей

Синтаксис

```
short compareBoundaryPoints(unsigned short какСравнивать,
                            Range исходнаяОбласть)
    throws DOMException;
```

Аргументы

какСравнивать

Указывает, как выполнять сравнение (т. е. задает сравниваемые граничные точки). Допустимые значения представляют собой константы, определенные в интерфейсе `Range`.

исходнаяОбласть

Область, которая должна сравниваться с данной.

Возвращаемое значение

Значение -1, если указанная граничная точка области находится перед граничной точкой, заданной аргументом *исходнаяОбласть*; значение 0, если две указанные граничные точки совпадают, или значение 1, если указанная граничная точка находится после граничной точки, заданной аргументом *исходнаяОбласть*.

Исключения

Если исходная область представляет собой область другого документа, этот метод генерирует исключение `DOMException` с кодом `WRONG_DOCUMENT_ERR`.

Описание

Этот метод сравнивает граничную точку области с граничной точкой, указанной аргументом *исходнаяОбласть*, и возвращает значение, задающее их относительный порядок в документе. Аргумент *какСравнивать* показывает, какие граничные точки каждой из областей должны сравниваться. Далее приведены допустимые значения этого аргумента и объяснен их смысл:

`Range.START_TO_START`

Сравнивает начальные точки двух объектов `Range`.

`Range.END_TO_END`

Сравнивает конечные точки двух объектов `Range`.

`Range.START_TO_END`

Сравнивает начальную точку исходной области с конечной точкой данной области.

`Range.END_TO_START`

Сравнивает конечную точку исходной области с начальной точкой данной области.

Возвращаемое значение данного метода – число, задающее относительную позицию данной области по отношению к указанной исходной области. Следовательно, можно предполагать, что константы для аргумента *какСравнивать* будут определять граничную точку данной области как первую, а граничную область исходной области – как вторую. Однако вопреки здравому смыслу константа `Range.START_TO_END` задает сравнение конечной точки данной области с начальной точкой указанной исходной области. Аналогично константа `Range.END_TO_START` задает сравнение начальной точки данной области с конечной точкой указанной исходной области.

Range.deleteContents() DOM Level 2 Range

удаляет область из документа

Синтаксис

```
void deleteContents()
    throws DOMException;
```

Исключения

Если какая-либо из частей документа, представляемая данной областью, доступна только для чтения, метод генерирует исключение DOMException с кодом NO_MODIFICATION_ALLOWED_ERR.

Описание

Метод удаляет все содержимое документа, заданное областью. Когда он завершает работу, область сжимается и обе граничные точки оказываются в одной позиции. Обратите внимание: удаление может приводить к созданию смежных узлов Text, которые могут быть объединены с помощью метода Node.normalize().

Способ копирования содержимого описывается в справочной статье о методе cloneContents(), а способ копирования и удаления содержимого документа одной операцией — в статье о методе extractContents().

См. также Node.normalize(),Range.cloneContents(), Range.extractContents()

Range.detach() DOM Level 2 Range

освобождает объект Range

Синтаксис

```
void detach()
    throws DOMException;
```

Исключения

Как все методы Range, метод detach() генерирует исключение DOMException с кодом INVALID_STATE_ERR, если он вызывается для объекта Range, который уже был отсоединен.

Описание

Реализации DOM отслеживают все объекты Range, созданные для документа, т. к. при изменении документа может потребоваться изменить граничные точки области. Если вы уверены, что объект Range больше не потребуется, вызовите метод detach(), сообщая реализации, что эту область больше не надо отслеживать. Обратите внимание: после вызова метода detach() для объекта Range любое использование данного объекта будет генерировать исключение. Вызов detach() не является обязательным, но может повысить производительность при изменениях документа, если объект Range не уничтожается сборщиком мусора немедленно.

Range.extractContents() DOM Level 2 Range

удаляет содержимое документа и возвращает его в виде DocumentFragment

Синтаксис

```
DocumentFragment extractContents()
    throws DOMException;
```

Возвращаемое значение

Узел DocumentFragment, в котором находится содержимое данной области.

Исключения

Метод генерирует исключение DOMException с кодом NO_MODIFICATION_ALLOWED_ERR, если какая-либо часть извлекаемого содержимого документа доступна только для чтения, или с кодом HIERARCHY_REQUEST_ERR, если область содержит узел DocumentType.

Описание

Метод удаляет указанную область документа и возвращает узел DocumentFragment, содержащий удаленное содержимое (или копию удаленного содержимого). Когда этот метод завершает свою работу, область сжимается, и документ может содержать смежные узлы Text (которые могут быть нормализованы методом Node.normalize()).

См. также DocumentFragment, Range.cloneContents(), Range.deleteContents()

Range.insertNode() DOM Level 2 Range

вставляет узел в начало области

Синтаксис

```
void insertNode(Node новыйУзел)
    throws RangeException, DOMException;
```

Аргументы

новыйУзел

Узел, который должен быть вставлен в документ.

Исключения

Метод генерирует исключение RangeException с кодом INVALID_NODE_TYPE_ERR, если узел *новыйУзел* является узлом Attr, Document, Entity или Notation.

Он также генерирует исключение DOMException с одним из перечисленных ниже кодов в следующих ситуациях:

HIERARCHY_REQUEST_ERR

Узел, содержащий начало области, не допускает наличия дочерних узлов или дочерних узлов указанного типа либо новый узел является предком данного узла.

NO_MODIFICATION_ALLOWED_ERR

Узел, содержащий начало области, или любые его предки доступны только для чтения.

WRONG_DOCUMENT_ERR

Новый узел – это часть документа, отличного от данного.

Описание

Метод вставляет указанный узел (и всех его потомков) в документ в начальной позиции этой области. Когда метод завершает работу, область содержит только вставленный узел. Если *новыйУзел* уже является частью документа, он удаляется из своей текущей позиции и вставляется заново в начале области. Если *новыйУзел* является узлом DocumentFragment, он не вставляется сам, а в начало области вставляются по порядку все его дочерние элементы.

Если узел, содержащий начало области, – это узел Text, то перед вставкой он разбивается на два смежных узла. Если *новыйУзел* – это узел Text, то он после вставки не объе-

диняется с любыми смежными узлами Text. Для объединения смежных узлов следует использовать метод Node.normalize().

См. также DocumentFragment, Node.normalize()

Range.selectNode() DOM Level 2 Range

ограничивает область указанным узлом

Синтаксис

```
void selectNode(Node выбираемыйУзел)
    throws RangeException, DOMException;
```

Аргументы

выбираемыйУзел

Узел, который должен быть «выбран» (т. е. узел, который должен стать содержимым данной области).

Исключения

Если *выбираемыйУзел* является узлом Attr, Document или DocumentFragment, генерируется исключение RangeException с кодом INVALID_NODE_TYPE_ERR.

Если *выбираемыйУзел* является частью документа, отличного от того, в котором был создан данный объект Range, генерируется исключение DOMException с кодом WRONG_DOCUMENT_ERR.

Описание

Этот метод устанавливает содержимое области равным указанному *выбираемомуУзлу*, т. е. «выбирает» этот узел и всех его потомков.

См. также Range.selectNodeContents()

Range.selectNodeContents() DOM Level 2 Range

устанавливает границы области в дочерних узлах данного узла

Синтаксис

```
void selectNodeContents(Node выбираемыйУзел)
    throws RangeException, DOMException;
```

Аргументы

выбираемыйУзел

Узел, дочерние элементы которого становятся содержимым данной области.

Исключения

Если *выбираемыйУзел* или один из его предков является узлом DocumentType, Entity или Notation, генерируется исключение RangeException с кодом INVALID_NODE_TYPE_ERR.

Если *выбираемыйУзел* является частью документа, отличного от того, с помощью которого был создан данный объект Range, генерируется исключение DOMException с кодом WRONG_DOCUMENT_ERR.

Описание

Данный метод устанавливает граничные точки области таким образом, чтобы область содержала дочерние узлы *выбираемогоУзла*.

См. также `Range.selectNode()`

Range.setEnd() DOM Level 2 Range

устанавливает конечную точку области

Синтаксис

```
void setEnd(Node выбираемыйУзел,
            long смещение)
    throws RangeException, DOMException;
```

Аргументы

выбираемыйУзел

Узел, содержащий новую конечную точку.

смещение

Позиция конечной точки внутри *выбираемогоУзла*.

Исключения

Исключение `RangeException` генерируется с кодом `INVALID_NODE_TYPE_ERR`, если *выбираемыйУзел* или один из его предков является узлом `DocumentType`.

Исключение `DOMException` генерируется с кодом `WRONG_DOCUMENT_ERR`, если *выбираемыйУзел* представляет собой часть документа, отличного от того, с помощью которого был создан данный объект `Range`, или с кодом `INDEX_SIZE_ERR`, если смещение отрицательно или больше количества дочерних узлов или символов в *выбираемомУзле*.

Описание

Этот метод устанавливает конечную точку области путем задания значений свойств `endContainer` и `endOffset`.

Range.setEndAfter() DOM Level 2 Range

завершает область после указанного узла

Синтаксис

```
void setEndAfter(Node выбираемыйУзел)
    throws RangeException,DOMException;
```

Аргументы

выбираемыйУзел

Узел, после которого должна быть установлена конечная точка области.

Исключения

Исключение `RangeException` генерируется с кодом `INVALID_NODE_TYPE_ERR`, если *выбираемыйУзел* является узлом `Document`, `DocumentFragment` или `Attr` либо если корневой контейнер *выбираемогоУзла* не является узлом `Document`, `DocumentFragment` или `Attr`.

Исключение DOMException **генерируется с кодом** WRONG_DOCUMENT_ERR, **если** *выбираемыйУзел* **представляет собой часть документа, отличного от того, с помощью которого был создан данный объект** Range.

Описание

Метод устанавливает конечную точку данной области непосредственно после указанного *выбираемогоУзла*.

Range.setEndBefore() DOM Level 2 Range

завершает область перед указанным узлом

Синтаксис

```
void setEndBefore(Node выбираемыйУзел)
    throws RangeException, DOMException;
```

Аргументы

выбираемыйУзел

Узел, перед которым должна быть установлена конечная точка области.

Исключения

Этот метод генерирует те же исключения и при тех же обстоятельствах, что и метод Range.setEndAfter(). Подробности см. в описании этого метода.

Описание

Метод устанавливает конечную точку данной области непосредственно перед *выбираемымУзлом*.

Range.setStart() DOM Level 2 Range

устанавливает начальную точку области

Синтаксис

```
void setStart(Node выбираемыйУзел,
              long смещение)
throws RangeException, DOMException;
```

Аргументы

выбираемыйУзел

Узел, содержащий новую начальную точку.

смещение

Позиция новой начальной точки внутри *выбираемогоУзла*.

Исключения

Этот метод генерирует те же исключения и при тех же обстоятельствах, что и метод Range.setEnd(). Подробности см. в описании этого метода.

Описание

Метод устанавливает начальную точку данной области путем задания значений свойств startContainer и startOffset.

Range.setStartAfter() DOM Level 2 Range

начинает область непосредственно после указанного узла

Синтаксис

```
void setStartAfter(Node выбираемыйУзел)
    throws RangeException, DOMException;
```

Аргументы

выбираемыйУзел

Узел, после которого должна быть установлена начальная точка области.

Исключения

Этот метод генерирует те же исключения и при тех же обстоятельствах, что и метод `Range.setEndAfter()`. Подробности см. в описании этого метода.

Описание

Метод устанавливает начальную точку данной области непосредственно после указанного *выбираемогоУзла*.

Range.setStartBefore() DOM Level 2 Range

начинает область перед указанным узлом

Синтаксис

```
void setStartBefore(Node выбираемыйУзел)
    throws RangeException, DOMException;
```

Аргументы

выбираемыйУзел

Узел, перед которым должна быть установлена начальная точка области.

Исключения

Этот метод генерирует те же исключения и при тех же обстоятельствах, что и метод `Range.setEndAfter()`. Подробности см. в описании этого метода.

Описание

Метод устанавливает начальную точку данной области непосредственно перед указанным *выбираемымУзлом*.

Range .surroundContents() DOM Level 2 Range

окружает содержимое области указанным узлом

Синтаксис

```
void surroundContents(Node новыйРодитель)
    throws RangeException, DOMException;
```

Аргументы

новыйРодитель

Узел, становящийся новым родительским узлом для содержимого данной области.

Исключения

Этот метод генерирует исключение DOMException или RangeException с одним из перечисленных ниже кодов в следующих ситуациях:

DOMException.HIERARCHY_REQUEST_ERR

Узел-контейнер начала области не допускает наличия дочерних узлов или дочерних узлов типа узла *новыйРодитель* либо *новыйРодитель* является предком данного узла-контейнера.

DOMException.NO_MODIFICATION_ALLOWED_ERR

Предок граничной точки области доступен только для чтения и не допускает вставок.

DOMException.WRONG_DOCUMENT_ERR

Узел *новыйРодитель* и эта область созданы с помощью разных объектов Document.

RangeException.BAD_BOUNDARYPOINTS_ERR

Область частично выбирает узел (отличный от узла Text), так что представляемая этим объектом область документа не может быть окружена узлом.

RangeException.INVALID_NODE_TYPE_ERR

Узел *новыйРодитель* — это узел Document, DocumentFragment, DocumentType, Attr, Entity или Notation.

Описание

Этот метод меняет родителя для содержимого данной области на узел *новыйРодитель* и затем вставляет этот узел в документ в начальной позиции области. Это, например, может использоваться для помещения области документа внутрь элемента <b> или .

Если узел *новыйРодитель* уже является частью документа, он сначала удаляется из документа и все его дочерние элементы уничтожаются. После завершения работы метода область начинается непосредственно перед узлом *новыйРодитель* и заканчивается сразу после него.

Range.toString() DOM Level 2 Range

извлекает содержимое области в виде строки текста

Синтаксис

```
String toString();
```

Возвращаемое значение

Содержимое данной области в виде строки обычного текста без какой-либо разметки.

RangeException DOM Level 2 Range

сигнализирует об исключении, относящемся к API модуля Range **Object→RangeException**

Константы

Следующие константы определяют допустимые значения свойства code объекта RangeException. Обратите внимание: эти константы представляют собой статические свойства RangeException, а не свойства отдельных объектов исключений.

`unsigned short BAD_BOUNDARYPOINTS_ERR = 1`

Граничные точки области недопустимы для данной операции.

`unsigned short INVALID_NODE_TYPE_ERR = 2`

Была предпринята попытка установить узел-контейнер граничной точки области равным недопустимому узлу или узлу с недопустимым предком.

Свойства

`unsigned short code`

Код ошибки, предоставляющий сведения о причинах ошибки. Допустимые значения данного свойства (и их смысл) определяются только что перечисленными константами.

Описание

Исключение `RangeException` генерируется определенными методами интерфейса `Range`, сигнализируя о какой-либо проблеме. Обратите внимание: большинство исключений, генерируемых методами `Range`, представляют собой объекты `DOMException`. Исключение `RangeException` генерируется, только если ни одна из существующих констант ошибок `DOMException` не подходит для описания исключения.

Reset

См. описание объекта Input

Screen JavaScript 1.2

предоставляет информацию о дисплее **Object→Screen**

Синтаксис

`screen`

Свойства

`availHeight`

Указывает доступную высоту экрана (в пикселах), на котором отображается веб-броузер. В операционных системах линейки Windows эта доступная высота не включает пространство, занятое полупостоянными элементами, такими как панель задач в нижней части экрана.

`availWidth`

Задает доступную ширину экрана (в пикселах), на котором отображается веб-броузер. В операционных системах линейки Windows эта доступная ширина не включает пространство, занимаемое полупостоянными элементами, такими как панели быстрого доступа к приложениям.

`colorDepth`

Определяет глубину цвета в битах на пиксел.

`height`

Задает общую высоту экрана (в пикселах), на котором отображается веб-броузер. См. также описание свойства `availHeight`.

`width`

Задает общую ширину экрана (в пикселах), на котором отображается веб-броузер. См. также описание свойства `availWidth`.

Описание

Свойство `screen` любого объекта `Window` ссылается на объект `Screen`. Свойства этого глобального объекта содержат информацию об экране, на котором отображается броузер. JavaScript-программы могут руководствоваться этой информацией для оптимизации вывода в соответствии с возможностями дисплея пользователя. Например, программа может выбирать между большими и маленькими изображениями в зависимости от размера экрана и между изображениями с 16- и 8-разрядным представлением цветов в зависимости от глубины цвета. JavaScript-программа может также использовать информацию о размере экрана для создания новых окон в центре экрана.

См. также Свойство `screen` объекта `Window`

Select

DOM Level 2 HTML

графический список для выбора **Node→Element→HTMLElement→Select**

Свойства

`readonly Form form`

Элемент `<form>`, в котором содержится элемент `<select>`.

`readonly long length`

Количество элементов `<option>`, содержащихся в элементе `<select>`. Значение этого свойства совпадает со значением свойства `options.length`.

`readonly HTMLCollection options`

Массив (`HTMLCollection`) объектов `Option`, которые представляют элементы `<option>`, содержащиеся в данном элементе `<select>`, в порядке их расположения в исходном коде документа. Подробные сведения об этом массиве можно найти в справочной статье о свойстве `Select.options[]`.

`long selectedIndex`

Индекс выбранного варианта в массиве `options`. Если ни один из вариантов не выбран, значение `selectedIndex` равно -1. Если выбрано более одного варианта, свойство `selectedIndex` задает индекс только первого из них.

Установка значения этого свойства приводит к выбору указанного варианта и отменяет выбор всех остальных, даже если в объекте `Select` указан атрибут `multiple`. Если выбор реализован в виде списка (когда значение свойства `size` > 1), то выбор всех вариантов можно отменить, установив свойство `selectedIndex` равным -1. Обратите внимание: этот способ изменения выбора не приводит к вызову обработчика события `onchange()`.

`readonly String type`

Если свойство `multiple` имеет значение `true`, данное свойство имеет значение `"select-multiple"`, в противном случае — `"select-one"`. Это свойство призвано обеспечить совместимость со свойством `type` объекта `Input`.

Помимо перечисленных выше свойств объекты `Select` имеют дополнительные свойства, соответствующие HTML-атрибутам:

Свойство	Атрибут	Описание
boolean disabled	disabled	Определяет, доступен ли элемент для пользователя
boolean multiple	multiple	Определяет, возможен ли одновременный выбор нескольких вариантов
String name	name	Имя элемента используется при отправке данных формы
long size	size	Количество одновременно отображаемых вариантов
long tabIndex	tabindex	Порядковый номер элемента, используемый для организации перехода от элемента к элементу с помощью клавиши Tab

Методы

`add()`

Вставляет новый объект `Option` в массив `options` либо в конец массива, либо перед указанным элементом.

`blur()`

Убирает фокус ввода с этого элемента.

`focus()`

Переносит фокус ввода на данный элемент.

`remove()`

Удаляет элемент `<option>` из заданной позиции.

Обработчики событий

`onchange`

Вызывается, когда пользователь выбирает элемент или отменяет выбор элемента.

Синтаксис HTML

Элемент `Select` создается с помощью стандартного HTML-тега `<select>`. Опции, присутствующие внутри элемента `Select`, создаются с помощью тега `<option>`:

```
<form>
  ...
<select
  name="имя" // Имя, идентифицирующее этот элемент; задает свойство name
   [ size="целое" ] // Количество видимых в элементе Select опций
   [ multiple ] // Если указано, может быть выбрано несколько опций
   [ onchange="обработчик" ] // Вызывается при изменении выбора
>
<option value="значение1" [selected]> метка_опции1
<option value="значение2" [selected]> метка_опции2
// Другие опции
</select>
  ...
</form>
```

Описание

Элемент `Select` представляет HTML-тег `<select>`, который для пользователя отображается как графический список выбора. Если в определении HTML-элемента присутствует атрибут `multiple`, пользователь может одновременно выбрать в списке любое число опций. Если этот атрибут отсутствует, пользователь может выбрать только

одну опцию, и опции ведут себя как переключатели – выбор одной из них приводит к отмене предыдущего выбора.

Опции в элементе Select могут отображаться двумя различными способами. Во-первых, если атрибут size имеет значение, большее 1, или если присутствует атрибут multiple, опции отображаются в окне броузера в виде списка высотой size строк. Если значение size меньше, чем число опций, в списке появляется полоса прокрутки, чтобы были доступны все опции. Во-вторых, если значение атрибута size равно 1 и атрибут multiple не указан, текущая выбранная опция отображается в единственной строке, а список всех остальных опций доступен через выпадающее меню. Первый стиль представления позволяет видеть все доступные опции, но требует больше пространства в окне броузера. Второй стиль требует минимум пространства, но не дает возможности сразу увидеть альтернативные опции.

Самый большой интерес представляет свойство options[] элемента Select. Это массив объектов Option, описывающий варианты выбора, представленные в элементе Select. Свойство length задает длину массива (как и options.length). Подробности см. в справочной статье об объекте Option.

Если в элементе Select отсутствует атрибут multiple, определить, какая опция выбрана, можно с помощью свойства selectedIndex. Однако если допускается возможность одновременного выбора нескольких опций, это свойство содержит индекс первой выбранной опции. Чтобы определить все множество выбранных опций, необходимо обойти в цикле массив options[] и проверить свойство selected каждого объекта Option.

Опции, отображаемые в элементе Select, могут динамически меняться. Добавить новую опцию можно с помощью метода add() и функции-конструктора Option(), а удалить опцию – с помощью метода remove(). Кроме того, существует возможность внесения изменений путем прямого манипулирования массивом options[].

См. также Form, Option, Select.options[]; глава 18

Select.add() DOM Level 2 HTML

вставляет элемент <option>

Синтаксис

```
void add(HTMLElement элемент,
         HTMLElement до)
    throws DOMException;
```

Аргументы

элемент

Добавляемый элемент Option.

до

Элемент в массиве options, перед которым должен быть вставлен *элемент*. Если этот аргумент имеет значение null, *элемент* добавляется в конец массива options.

Исключения

Этот метод генерирует исключение DOMException с кодом NOT_FOUND_ERR, если аргумент *до* указывает на объект, который не является элементом массива options.

Описание

Данный метод добавляет новый элемент <option>. Аргумент *элемент* – это объект Option, который представляет добавляемый элемент <option>. Аргумент *до* указывает на объект Option, перед которым должен быть вставлен *элемент*. Если *до* является частью группы <optgroup>, *элемент* всегда добавляется как часть этой же группы. Если аргумент *до* имеет значение null, *элемент* становится последним потомком элемента <select>.

См. также Option

Select.blur() DOM Level 2 HTML

убирает фокус ввода с данного элемента

Синтаксис

```
void blur();
```

Описание

Метод убирает фокус ввода с данного элемента.

Select.focus() DOM Level 2 HTML

переносит фокус ввода на данный элемент

Синтаксис

```
void focus();
```

Описание

Метод передает фокус ввода на данный элемент <select>, благодаря чему пользователь может взаимодействовать с ним не только с помощью мыши, но и с помощью клавиатуры.

Select.onchange() DOM Level 0

обработчик, вызываемый при смене элемента выбора

Синтаксис

```
Function onchange
```

Описание

Свойство onchange объекта Select ссылается на функцию обработки события, вызываемую, когда пользователь выбирает опцию или отменяет ее выбор. Событие не несет информацию о вновь выбранной или выбранных опциях, для этого необходимо проверить свойство selectedIndex объекта Select или свойства selected объектов Option.

Другой способ регистрации обработчиков событий описывается в справочной статье о методе Element.addEventListener().

См. также Element.addEventListener(), Option; глава 17

Select.options[] DOM Level 2 HTML

варианты выбора в объекте Select

Синтаксис

```
readonly HTMLCollection options
```

Описание

Свойство `options[]` — это подобный массиву объект `HTMLCollection`, содержащий объекты `Option`, каждый из которых описывает один из вариантов выбора, представленных в объекте `Select`.

Свойство `options[]` не является обычным объектом `HTMLCollection`. Для обратной совместимости с ранними версиями броузеров эта коллекция наделена специальными функциональными возможностями, которые позволяют изменять опции, отображаемые в объекте `Select`:

- Если установить значение `options.length` равным 0, все опции в объекте `Select` удаляются.
- Если установить значение `options.length` меньше текущего значения, количество опций в объекте `Select` уменьшается, а те опции, которые находятся в конце массива, исчезают.
- Если установить элемент массива `options[]` равным `null`, эта опция будет удалена из объекта `Select`, а элементы, расположенные в массиве над ней, переместятся вниз, при этом их индексы изменятся таким образом, чтобы элементы заняли освободившееся место в массиве.
- Если новый объект `Option` создается с помощью конструктора `Option()` (см. справочную статью об объекте `Option`), то можно будет добавить эту опцию в конец списка опций в объекте `Select`, присвоив только что созданную опцию позиции в конце массива `options[]`. Это можно сделать при помощи выражения `options[options.length]` (см. справочную статью о методе `Select.add()`).

См. также

`Option`

Select.remove() DOM Level 2 HTML

удаляет элемент <option>

Синтаксис

```
void remove(long индекс);
```

Аргументы

индекс Позиция удаляемого элемента `<option>` в массиве `options`.

Описание

Этот метод удаляет элемент `<option>`, находящийся в указанной позиции в массиве `options`. Если *индекс* меньше нуля или больше числа имеющихся опций, метод `remove()` игнорирует его и ничего не делает.

См. также

`Option`

Style

См. описание объекта CSS2Properties

Submit

См. описание объекта Input

Table

DOM Level 2 HTML

элемент <table> в HTML-документе **Node→Element→HTMLElement→Table**

Свойства

`HTMLElement caption`

Ссылка на элемент `<caption>` таблицы или `null`, если он отсутствует.

`readonly HTMLCollection rows`

Массив (`HTMLCollection`) объектов `TableRow`, представляющих все строки в таблице. Включает все строки, определенные с помощью тегов `<thead>`, `<tfoot>` и `<tbody>`.

`readonly HTMLCollection tBodies`

Массив (`HTMLCollection`) объектов `TableSection`, представляющий все разделы `<tbody>` в таблице.

`TableSection tFoot`

Элемент `<tfoot>` таблицы или `null`, если он отсутствует.

`TableSection tHead`

Элемент `<thead>` таблицы или `null`, если он отсутствует.

В дополнение к только что перечисленным свойствам этот интерфейс определяет свойства, приведенные в следующей таблице и представляющие HTML-атрибуты элемента `<table>`.

Свойство	Атрибут	Описание
`String align` (устаревшее)	`align`	Горизонтальное выравнивание таблицы в документе
`String bgColor` (устаревшее)	`bgcolor`	Цвет фона таблицы
`String border`	`border`	Ширина границы вокруг таблицы
`String cellPadding`	`cellpadding`	Пространство между содержимым и границей ячейки
`String cellSpacing`	`cellspacing`	Пространство между границами ячеек
`String frame`	`frame`	Управляет отображением границ таблицы
`String rules`	`rules`	Управляет отображением линий внутри таблицы
`String summary`	`summary`	Общее описание таблицы
`String width`	`width`	Ширина таблицы

Методы

`createCaption()`

Возвращает элемент `<caption>` для таблицы или создает (и вставляет) новый элемент, если он еще не существует.

`createTFoot()`

Возвращает существующий элемент `<tfoot>` для таблицы или создает (и вставляет) новый элемент, если он еще не существует.

`createTHead()`

Возвращает существующий элемент `<thead>` для таблицы или создает (и вставляет) новый элемент, если он еще не существует.

`deleteCaption()`

Удаляет элемент `<caption>` из таблицы, если он существует.

`deleteRow()`

Удаляет строку в указанной позиции таблицы.

`deleteTFoot()`

Удаляет элемент `<tfoot>` из таблицы, если он существует.

`deleteTHead()`

Удаляет элемент `<thead>` из таблицы, если он существует.

`insertRow()`

Вставляет новый, пустой элемент `<tr>` в указанной позиции.

Описание

Объект `Table` представляет HTML-элемент `<table>` и определяет несколько удобных свойств и методов для получения и модификации различных частей таблицы. Эти методы и свойства облегчают работу с таблицами, но они также могут быть продублированы с помощью базовых DOM-методов.

См. также

`TableCell`, `TableRow`, `TableSection`

Table.createCaption() DOM Level 2 HTML

получает или создает элемент <caption>

Синтаксис

```
HTMLElement createCaption();
```

Возвращаемое значение

Объект `HTMLElement`, представляющий элемент `<caption>` (название) таблицы. Если у таблицы уже есть название, метод просто возвращает его. Если у таблицы еще нет названия, метод создает новый (пустой) элемент `<caption>`, вставляет его в таблицу, а затем возвращает.

Table.createTFoot() DOM Level 2 HTML

получает или создает элемент <tfoot>

Синтаксис

```
HTMLElement createTFoot();
```

Возвращаемое значение

Объект `TableSection`, представляющий элемент `<tfoot>` (нижний колонтитул) таблицы. Если в таблице уже есть нижний колонтитул, этот метод просто возвращает его.

Если в таблице еще нет нижнего колонтитула, этот метод создает новый (пустой) элемент <tfoot>, вставляет его в таблицу, а затем возвращает.

Table.createTHead() DOM Level 2 HTML

получает или создает элемент <thead>

Синтаксис

```
HTMLElement createTHead();
```

Возвращаемое значение

Объект TableSection, представляющий элемент <thead> (верхний колонтитул, или шапка) таблицы. Если в таблице уже есть верхний колонтитул, этот метод просто возвращает его. Если в таблице еще нет верхнего колонтитула, этот метод создает новый (пустой) элемент <thead>, вставляет его в таблицу, а затем возвращает.

Table.deleteCaption() DOM Level 2 HTML

удаляет элемент <caption> из таблицы

Синтаксис

```
void deleteCaption();
```

Описание

Если в таблице есть элемент <caption>, метод удаляет его из дерева документа, в противном случае не делает ничего.

Table.deleteRow() DOM Level 2 HTML

удаляет строку из таблицы

Синтаксис

```
void deleteRow(long индекс)
    throws DOMException;
```

Аргументы

индекс

Указывает позицию удаляемой строки внутри таблицы.

Исключения

Этот метод генерирует исключение DOMException с кодом INDEX_SIZE_ERR, если индекс меньше нуля или больше либо равен количеству строк в таблице.

Описание

Этот метод удаляет строку в указанной позиции в таблице. Строки нумеруются в том порядке, в котором они присутствуют в исходном тексте документа. Строки в разделах <thead> и <tfoot> нумеруются вместе с другими строками в таблице.

См. также TableSection.deleteRow()

Table.deleteTFoot() DOM Level 2 HTML

удаляет элемент <tfoot> из таблицы

Синтаксис

```
void deleteTFoot();
```

Описание

Если в таблице есть элемент <tfoot>, метод удаляет его из дерева документа. Если таблица не имеет нижнего колонтитула, этот метод не делает ничего.

Table.deleteTHead() DOM Level 2 HTML

удаляет элемент <thead> из таблицы

Синтаксис

```
void deleteTHead();
```

Описание

Если в таблице есть элемент <thead>, метод удаляет его из дерева документа. Если таблица не имеет верхнего колонтитула, метод не делает ничего.

Table.insertRow() DOM Level 2 HTML

добавляет в таблицу новую пустую строку

Синтаксис

```
HTMLElement insertRow(long индекс)
    throws DOMException;
```

Аргументы

индекс

Позиция, в которую должна быть вставлена новая строка.

Возвращаемое значение

Элемент TableRow, представляющий только что вставленную строку.

Исключения

Метод генерирует исключение DOMException с кодом INDEX_SIZE_ERR, если *индекс* меньше нуля или больше количества строк в таблице.

Описание

Этот метод создает новый элемент TableRow, представляющий тег <tr>, и вставляет его в таблицу в указанной позиции.

Новая строка вставляется в том же разделе таблицы и непосредственно перед существующей строкой, в позиции, заданной аргументом *индекс*. Если *индекс* равен количеству строк в таблице, новая строка дописывается к последнему разделу таблицы. Если таблица первоначально пуста, новая строка вставляется в новый раздел <tbody>, который, в свою очередь, вставляется в таблицу.

Для добавления содержимого к только что созданной строке можно использовать вспомогательный метод `TableRow.insertCell()`.

См. также `TableSection.insertRow()`

TableCell

DOM Level 2 HTML

ячейки <td> или <th> в HTML-таблице **Node→Element→HTMLElement→TableCell**

Свойства

`readonly long cellIndex`

Позиция данной ячейки внутри строки.

В дополнение к свойству `cellIndex` этот интерфейс определяет свойства, приведенные в следующей таблице. Эти свойства непосредственно соответствуют HTML-атрибутам элементов `<td>` и `<th>`:

Свойство	Атрибут	Описание
`String abbr`	`abbr`	См. спецификацию по HTML
`String align`	`align`	Горизонтальное выравнивание ячейки
`String axis`	`axis`	См. спецификацию по HTML
`String bgColor` (устаревшее)	`bgcolor`	Цвет фона ячейки
`String ch`	`ch`	Символ выравнивания
`String chOff`	`choff`	Смещение символа выравнивания
`long colSpan`	`colspan`	Столбцы, охваченные ячейкой
`String headers`	`headers`	Значения `id` для заголовков ячейки
`String height` (устаревшее)	`height`	Высота ячейки в пикселах
`boolean noWrap` (устаревшее)	`nowrap`	Ячейка, в которой не выполняется перенос слов
`long rowSpan`	`rowspan`	Строки, охваченные ячейкой
`String scope`	`scope`	Диапазон ячейки заголовка
`String vAlign`	`valign`	Вертикальное выравнивание ячейки
`String width` (устаревшее)	`width`	Ширина ячейки в пикселах

Описание

Этот интерфейс представляет элементы `<td>` и `<th>` в HTML-таблицах.

TableRow

DOM Level 2 HTML

элемент <tr> в HTML-таблице **Node→Element→HTMLElement→TableRow**

Свойства

`readonly HTMLCollection cells`

Массив (`HTMLCollection`) объектов `HTMLTableCellElement`, представляющих ячейки в строке.

`readonly long rowIndex`

Позиция строки в таблице.

`readonly long sectionRowIndex`

Позиция строки в данном разделе (т. е. внутри данного элемента `<thead>`, `<tbody>` или `<tfoot>`).

В дополнение к только что перечисленным свойствам данный интерфейс определяет свойства, приведенные в следующей таблице. Эти свойства соответствуют HTML-атрибутам элемента `<tr>`.

Свойство	Атрибут	Описание
`String align`	`align`	Горизонтальное выравнивание ячейки, применяемое в этой строке по умолчанию
`String bgColor` **(устаревшее)**	`bgcolor`	Цвет фона ячейки
`String ch`	`ch`	Символ выравнивания
`String chOff`	`choff`	Смещение символа выравнивания
`String vAlign`	`valign`	Вертикальное выравнивание ячейки

Методы

`deleteCell()`

Удаляет указанную ячейку из строки.

`insertCell()`

Вставляет пустой элемент `<td>` в строку в указанной позиции.

Описание

Этот интерфейс представляет строку в HTML-таблице.

TableRow.deleteCell() DOM Level 2 HTML

удаляет ячейку из строки таблицы

Синтаксис

```
void deleteCell(long индекс)
    throws DOMException;
```

Аргументы

индекс

Позиция удаляемой ячейки в строке.

Исключения

Метод генерирует исключение `DOMException` с кодом `INDEX_SIZE_ERR`, если *индекс* меньше нуля либо больше или равен количеству ячеек в строке.

Описание

Этот метод удаляет ячейки в указанной позиции в строке таблицы.

TableRow.insertCell() DOM Level 2 HTML

вставляет новый пустой элемент <td> в строку таблицы

Синтаксис

```
HTMLElement insertCell(long индекс)
    throws DOMException;
```

Аргументы

индекс

Позиция, в которую должна быть вставлена новая ячейка.

Возвращаемое значение

Объект `TableCell`, представляющий вновь созданный и вставленный элемент `<td>`.

Исключения

Этот метод генерирует исключение `DOMException` с кодом `INDEX_SIZE_ERR`, если *индекс* меньше нуля или больше, чем количество ячеек в строке.

Описание

Этот метод создает новый элемент `<td>` и вставляет его в строку в указанной позиции. Новая ячейка вставляется непосредственно перед ячейкой, находящейся в данный момент в позиции, заданной аргументом *индекс*. Если *индекс* равен количеству ячеек в строке, новая ячейка дописывается в конец строки.

Обратите внимание: этот вспомогательный метод позволяет вставлять только ячейки данных – `<td>`. Чтобы вставить ячейку верхнего колонтитула в строку, необходимо создать и вставить элемент `<th>` методами `Document.createElement()` и `Node.insertBefore()` или другими родственными им методами.

TableSection DOM Level 2 HTML

раздел верхнего или нижнего колонтитула либо тела таблицы Node→Element→HTMLElement→TableSection

Свойства

```
readonly HTMLCollection rows
```

Массив (`HTMLCollection`) объектов `TableRow`, представляющих строки в этой секции таблицы.

В дополнение к свойству `rows` этот интерфейс определяет свойства, приведенные в следующей таблице и представляющие атрибуты данного HTML-элемента.

Свойство	Атрибут	Описание
`String align`	`align`	Горизонтальное выравнивание ячейки, применяемое в этом разделе по умолчанию
`STRING CH`	`ch`	Символ выравнивания, применяемый в этом разделе по умолчанию
`String chOff`	`choff`	Смещение символа выравнивания, применяемое в этом разделе по умолчанию
`String vAlign`	`valign`	Вертикальное выравнивание ячейки, применяемое в этом разделе по умолчанию

Методы

`deleteRow()`

Удаляет строку с указанным номером из данной секции.

`insertRow()`

Вставляет пустую строку в данную секцию в указанной позиции.

Описание

Этот интерфейс представляет раздел `<tbody>`, `<thead>` или `<tfoot>` HTML-таблицы. Свойства `tHead` и `tFoot` объектов `Table` и `TableSection`, а также свойство `tBodies` — это объекты `TableSection`.

TableSection.deleteRow() DOM Level 2 HTML

удаляет строку внутри раздела таблицы

Синтаксис

```
void deleteRow(long индекс)
    throws DOMException;
```

Аргументы

индекс

Позиция в строке внутри данного раздела.

Исключения

Этот метод генерирует исключение `DOMException` с кодом `INDEX_SIZE_ERR`, если *индекс* меньше нуля либо больше или равен количеству строк в данном разделе.

Описание

Этот метод удаляет строку из указанной позиции в данной секции (разделе). Обратите внимание: для этого метода аргумент *индекс* задает позицию строки внутри раздела, а не в таблице в целом.

См. также

`Table.deleteRow()`

TableSection.insertRow() DOM Level 2 HTML

вставляет новую пустую строку в секцию таблицы

Синтаксис

```
HTMLElement insertRow(long индекс)
    throws DOMException;
```

Аргументы

индекс

Позиция внутри раздела, в который должна быть вставлена новая строка.

Возвращаемое значение

Элемент `TableRow`, представляющий вновь созданный и вставленный элемент `<tr>`.

Исключения

Этот метод генерирует исключение DOMException с кодом INDEX_SIZE_ERR, если *индекс* меньше нуля или больше количества строк в данном разделе.

Описание

Этот метод создает новый элемент <tr> и вставляет его в данный раздел таблицы в указанной позиции. Если аргумент *индекс* равен количеству строк в разделе, новая строка дописывается в конец раздела. В противном случае новая строка вставляется непосредственно перед строкой, находящейся в данный момент в позиции, заданной аргументом *индекс*. Обратите внимание: для этого метода *индекс* задает позицию строки внутри одного раздела, а не в таблице в целом.

См. также Table.insertRow()

Text DOM Level 1 Core

последовательность текста в HTML- или XML-документе **Node→CharacterData→Text**

Подынтерфейсы

CDATASection

Методы

splitText()

Разбивает данный узел Text на два в указанной символьной позиции и возвращает новый текстовый узел.

Описание

Узел Text представляет последовательность обычного текста в HTML- или XML-документе. Обычный текст находится внутри HTML- и XML-элементов и их атрибутов, а узлы Text обычно являются дочерними по отношению к узлам Element и Attr. Узлы Text — это наследники CharacterData, а текстовое содержимое узла Text доступно через свойство data, унаследованное от CharacterData, или через свойство nodeValue, унаследованное от Node. Узлами Text можно манипулировать, используя любые методы, унаследованные от CharacterData, или метод splitText(), определенный в самом интерфейсе Text. В узлах Text никогда не бывает дочерних узлов.

Способ удаления пустых узлов Text и объединения смежных узлов Text из поддерева документа приводится в справочной статье о методе Node.normalize().

См. также CharacterData, Node.normalize()

Text.splitText() DOM Level 1 Core

разбивает узел Text на два

Синтаксис

```
Text splitText(unsigned long смещение)
    throws DOMException;
```

Аргументы

смещение

Символьная позиция, в которой нужно разбить узел Text.

Возвращаемое значение

Узел Text, выделенный из данного узла.

Исключения

Этот метод может генерировать исключение DOMException с одним из перечисленных ниже кодов:

INDEX_SIZE_ERR

Смещение отрицательно или больше длины узла Text или Comment.

NO_MODIFICATION_ALLOWED_ERR

Узел доступен только для чтения и не может быть модифицирован.

Описание

Этот метод разбивает узел Text на два по указанному *смещению*. Исходный узел Text модифицируется так, чтобы он содержал весь текст до символа в позиции *смещение*, но не включая его. Создается новый узел, который содержит все символы от позиции *смещение* (и включая ее) до конца строки. Этот новый узел Text представляет собой значение, возвращаемое методом. Кроме того, если исходный узел Text имеет родительский узел, то новый узел вставляется в родительский непосредственно после исходного узла.

Интерфейс CDATASection является наследником Text, и его метод splitText() также может использоваться с узлами CDATASection, в этом случае только что созданный узел будет узлом CDATASection, а не узлом Text.

См. также

Node.normalize()

Textarea

DOM Level 2 HTML

многострочная область ввода текста **Node→Element→HTMLElement→Textarea**

Свойства

String defaultValue

Начальное содержимое текстовой области. Когда выполняется сброс формы, содержимое текстовой области восстанавливается в это значение. Изменение этого свойства влечет за собой изменение отображаемого текста.

readonly Form form

Объект Form, который представляет элемент <form>, содержащий данный объект Textarea, или null, если элемент находится за пределами формы.

readonly String type

Тип данного элемента. Это свойство введено для сохранения совместимости с объектами Input. Данное свойство всегда имеет значение "textarea".

String value

Доступное для чтения и записи свойство. Начальное значение этого свойства совпадает со значением свойства defaultValue. Когда пользователь вводит символы в объект Textarea, свойство value обновляется в соответствии с введенными данными. Если свойство value устанавливается явно, указанная строка выводится в объекте Textarea. Свойство value содержит строку, отправляемую на сервер при передаче данных формы.

Помимо этих свойств объекты Textarea определяют дополнительные свойства, соответствующие HTML-атрибутам:

Свойство	Атрибут	Описание
String accessKey	accesskey	Комбинация «горячих» клавиш для перехода к данной текстовой области
long cols	cols	Ширина области в символах
boolean disabled	disabled	Признак активности области ввода
String name	name	Имя текстовой области; используется для организации доступа к элементу и отправки данных вместе с формой
boolean readOnly	readonly	Признак доступности текстовой области для редактирования
long rows	rows	Высота области в строках
long tabIndex	tabindex	Порядковый номер при переходе между элементами с помощью клавиши табуляции

Методы

blur()

Убирает фокус ввода с данного элемента.

focus()

Переносит фокус ввода на данный элемент.

select()

Выделяет все содержимое текстовой области.

Обработчики событий

onchange

Вызывается, когда пользователь изменяет значение в элементе Textarea и перемещает фокус ввода в другое место. Этот обработчик события вызывается не для каждого нажатия клавиши в элементе Textarea, а только когда пользователь завершает редактирование.

Синтаксис HTML

Объект Textarea создается с помощью стандартных HTML-тегов <textarea> и </textarea>:

```
<form>
  ...
  <textarea
    [ name="имя" ]      // Имя, по которому можно ссылаться на элемент
    [ rows="целое" ]          // Высота элемента в строках
    [ cols="целое" ]          // Ширина элемента в символах
    [ onchange="обработчик" ] // Обработчик события onchange()
>
    обычный_текст             // Первоначальный текст; задает defaultValue
  </textarea>
  ...
</form>
```

Описание

Элемент Textarea представляет HTML-тег <textarea> — многострочное текстовое поле на форме. Начальное содержимое текстовой области вставляется между тегами <textarea> и </textarea>. Получить и изменить текст можно с помощью свойства value.

Объект Textarea — это элемент ввода формы, подобный Input и Select. Аналогично этим объектам он определяет свойства form, name и type.

См. также Form, Input; глава 18

Textarea.blur() DOM Level 2 HTML

убирает фокус ввода с элемента

Синтаксис

```
void blur();
```

Описание

Метод убирает фокус ввода с данного элемента.

Textarea.focus() DOM Level 2 HTML

переносит фокус ввода на данный элемент

Синтаксис

```
void focus();
```

Описание

Метод передает фокус ввода на данный элемент, благодаря чему пользователь может редактировать отображаемый текст без предварительного щелчка мышью на текстовой области.

Textarea.onchange() DOM Level 0

обработчик, вызываемый при изменении значения

Синтаксис

```
Function onchange
```

Описание

Свойство onchange элемента Textarea ссылается на функцию обработки события, вызываемую, когда пользователь изменяет значение в текстовой области и затем «фиксирует» эти изменения, переместив фокус ввода в другое место.

Обратите внимание: обработчик onchange *не* вызывается, когда значение свойства value объекта Text устанавливается из JavaScript-кода. Кроме того, следует отметить, что этот обработчик предназначен для обработки завершенного изменения во введенном значении и потому не вызывается при каждом нажатии клавиши. Информация о том, как фиксировать каждое событие нажатия клавиши, приводится в справочной статье о свойстве HTMLElement.onkeypress. Альтернативный способ регистрации обработчиков событий описывается в справочной статье о методе Element.addEventListener().

См. также

Element.addEventListener(), HTMLElement.onkeypress, Input.onchange; глава 17

Textarea.select() DOM Level 2 HTML

выбирает текст в этом элементе

Синтаксис

```
void select();
```

Описание

Метод выделяет весь текст, выводимый в данном элементе <textarea>. Для большинства броузеров это означает, что при последующем нажатии клавиши пользователем весь текст заменяется новым.

TextField

См. описание объекта Input

UIEvent DOM Level 2 Events

сведения о событиях пользовательского интерфейса **Event→UIEvent**

Подынтерфейсы

KeyEvent, MouseEvent

Свойства

readonly long detail

Сведения о событии (число). Для событий click, mousedown и mouseup (MouseEvent) это свойство показывает количество щелчков: 1 – одинарный щелчок, 2 – двойной щелчок, 3 – тройной щелчок и т. д. Для событий DOMActivate значение этого поля, равное 1, соответствует обычной активации, равное 2 – «гиперактивации», например двойному щелчку или нажатию комбинации клавиш Shift-Enter.

readonly Window view

Окно (представление), в котором было сгенерировано событие.

Методы

initUIEvent()

Инициализирует свойства вновь созданного объекта UIEvent, включая свойства, унаследованные от интерфейса Event.

Описание

Интерфейс UIEvent – это подынтерфейс интерфейса Event, который определяет тип объекта Event, передаваемого событиями типов DOMFocusIn, DOMFocusOut и DOMActivate. Эти типы событий не часто применяются в веб-броузерах, а что более важно – интерфейс UIEvent является родительским интерфейсом для MouseEvent.

См. также

Event, KeyEvent, MouseEvent; глава 17

UIEvent.initUIEvent() DOM Level 2 Events

инициализирует свойства объекта UIEvent

Синтаксис

```
void initUIEvent(String typeArg,
                 boolean canBubbleArg,
                 boolean cancelableArg,
                 Window viewArg,
                 long detailArg);
```

Аргументы

typeArg

Тип события.

canBubbleArg

Признак «всплытия» события.

cancelableArg

Возможность отменены события методом `preventDefault()`.

viewArg

Окно, в котором произошло событие.

detailArg

Свойство `detail` для события.

Описание

Метод инициализирует свойства `view` и `detail` данного объекта `UIEvent`, а также свойства `type`, `bubbles` и `cancelable`, унаследованные от интерфейса `Event`. Этот метод может быть вызван исключительно для вновь созданных объектов `UIEvent` до того, как они будут переданы методу `Element.dispatchEvent()`.

См. также `Document.createEvent()`, `Event.initEvent()`, `MouseEvent.initMouseEvent()`

Window JavaScript 1.0

окно веб-броузера или фрейм **Object→Global→Window**

Синтаксис

```
self
window
window.frames[i]
```

Свойства

Объект `Window` определяет следующие свойства, а также наследует все глобальные свойства базового JavaScript (см. справочную статью об объекте `Global` в части III книги):

`closed`

Доступное только для чтения свойство, указывающее, было ли окно закрыто. Когда окно броузера закрывается, представляющий его объект `Window` не исчезает

просто так. Объект Window продолжает существовать, но его свойство closed устанавливается равным true.

defaultStatus

Доступная для чтения и записи строка, задающая сообщение, по умолчанию выводимое в строке состояния. Дополнительные сведения см. в справочной статье об объекте Window.defaultStatus.

document

Доступная только для чтения ссылка на объект Document, который описывает документ, содержащийся в этом окне или фрейме. Подробности см. в справочной статье об объекте Document.

event *[только в IE]*

В Internet Explorer это свойство ссылается на объект Event, содержащий сведения о самом последнем произошедшем в окне событии. Данное свойство используется в модели обработки событий IE. В стандартной модели событий DOM объект Event передается функциям-обработчикам в виде аргумента. Дополнительная информация приводится в справочной статье об объекте Event, а также в главе 17.

frames[]

Массив объектов Window, по одному на каждый фрейм или тег <iframe>, содержащийся в этом окне. Свойство frames.length содержит количество элементов в массиве frames[]. Обратите внимание: фреймы, на которые ссылается массив frames[], могут сами содержать фреймы и иметь свой массив frames[].

history

Доступная только для чтения ссылка на объект History данного окна или фрейма. Подробности см. в справочной статье об объекте History.

innerHeight, innerWidth

Доступные только для чтения свойства, задающие высоту и ширину в пикселах экранной области вывода окна. Эти размеры не включают высоту строки меню, полос прокрутки и тому подобное. Эти свойства не поддерживаются в IE, поэтому в данном броузере вместо них следует использовать свойства clientHeight и clientWidth объекта document.documentElement или document.body (в зависимости от версии IE). Дополнительную информацию можно найти в разделе 14.3.1.

location

Объект Location для окна или фрейма. Это свойство задает URL-адрес текущего загруженного документа. Установка этого свойства равным новой строке URL-адреса приводит к загрузке и выводу содержимого с этого URL-адреса в броузере. Более подробную информацию можно найти в справочной статье об объекте Location.

name

Строка, содержащая имя окна. Имя может быть задано, когда окно создается методом open() или в виде значения атрибута name тега <frame>. Имя окна может использоваться в качестве значения атрибута target тега <a> или <form>. При таком применении атрибута target указывается, что документ, загружаемый по гиперссылке, или результаты отправки данных формы должны отображаться в заданном окне или фрейме.

navigator

Доступное только для чтения свойство, представляющее ссылку на объект Navigator, позволяющий получить информацию о версии и конфигурации веб-броузера. Подробности см. в справочной статье об объекте Navigator.

opener

Доступное для чтения и записи свойство, представляющее ссылку на объект Window, в котором содержится сценарий, вызвавший метод open() для открытия в броузере окна верхнего уровня. Это свойство действительно только для объектов Window, представляющих окна верхнего уровня, но не для объектов, представляющих фреймы. Свойство opener может использоваться, чтобы созданное окно могло ссылаться на переменные и функции, определенные в создавшем его окне.

outerHeight, outerWidth

Доступные только для чтения целые свойства, задающие общую высоту и ширину окна в пикселах. Эти размеры включают высоту и ширину строки меню, панелей инструментов, полос прокрутки, границ окна и тому подобное. Эти свойства не поддерживаются в IE, причем этот броузер не предоставляет альтернативных свойств с той же функциональностью.

pageXOffset, pageYOffset

Доступные только для чтения целые, задающее число пикселов, на которые текущий документ был прокручен вправо (pageXOffset) и вниз (pageYOffset). Эти свойства не поддерживаются в Internet Explorer, поэтому в данном броузере следует использовать свойства scrollLeft и scrollTop объекта document.documentElement или document.body (в зависимости от версии IE). Дополнительную информацию можно найти в разделе 14.3.1.

parent

Доступная только для чтения ссылка на объект Window, содержащий данное окно или фрейм. Если окно является окном верхнего уровня, parent ссылается на само окно. Если окно является фреймом, свойство parent ссылается на окно или фрейм, в котором содержится данное окно.

screen

Это доступное только для чтения свойство представляет ссылку на объект Screen, свойства которого содержат информацию об экране, включая число доступных пикселов и цветов. Подробности см. в справочной статье об объекте Screen.

screenLeft, screenTop, screenX, screenY

Доступные только для чтения целые, задающие координаты верхнего левого угла окна на экране. Броузеры IE, Safari и Opera поддерживают свойства screenLeft и screenTop, тогда как Firefox и Safari – свойства screenX и screenY.

self

Доступная только для чтения ссылка на само окно. Синоним свойства window.

status

Доступная для чтения и записи строка, задающая текущее содержимое строки состояния броузера. Подробности см. в справочной статье о свойстве Window.status.

top

Доступная только для чтения ссылка на окно верхнего уровня, содержащее данное окно. Если данное окно само является окном верхнего уровня, top просто содержит ссылку на само окно. Если данное окно представляет собой фрейм, свойство top ссылается на окно верхнего уровня, содержащее данный фрейм. Сравните со свойством parent.

window

Свойство window идентично свойству self – оно содержит ссылку на окно.

Методы

Объект Window определяет следующие методы, а также наследует все глобальные функции, определяемые в базовом языке JavaScript (подробности см. в справочной статье об объекте Global и в третьей части книги).

addEventListener()

Добавляет функцию-обработчик события в набор обработчиков данного окна. Этот метод поддерживается всеми современными броузерами, за исключением IE. Альтернатива для IE описывается в справочной статье о методе attachEvent().

alert()

Выводит простое сообщение в диалоговом окне.

attachEvent()

Добавляет функцию-обработчик события в набор обработчиков событий документа. Это метод, реализованный в IE и представляющий альтернативу методу addEventListener().

blur()

Убирает фокус ввода в броузере с окна верхнего уровня.

clearInterval()

Отменяет периодическое исполнение кода.

clearTimeout()

Отменяет действие времени ожидания.

close()

Закрывает окно.

confirm()

Предлагает в диалоговом окне альтернативу «да» или «нет».

detachEvent()

Удаляет функцию-обработчик события из данного окна. Этот метод представляет собой альтернативу стандартному методу removeEventListener(), реализованную в IE.

focus()

Передает фокус ввода окну броузера верхнего уровня; на большинстве платформ это приводит к переводу окна на передний план.

getComputedStyle()

Определяет CSS-стили, применявшиеся к элементу документа.

moveBy()

Перемещает окно на относительную позицию.

moveTo()

Перемещает окно на абсолютную позицию.

open()

Создает и открывает новое окно.

print()

Имитирует щелчок по кнопке Печать в броузере.

prompt()

Запрашивает ввод простой строки в диалоговом окне.

`removeEventListener()`

Удаляет функцию-обработчик события из набора обработчиков событий данного окна. Этот стандартный метод реализован во всех современных броузерах, за исключением IE. Вместо него Internet Explorer предоставляет метод `detachEvent()`.

`resizeBy()`

Изменяет размер окна на указанную величину.

`resizeTo()`

Изменяет размер окна в соответствии с указанным размером.

`scrollBy`

Прокручивает окно на указанную величину.

`scrollTo()`

Прокручивает окно до указанной позиции.

`setInterval()`

Исполняет код через указанные промежутки времени.

`setTimeout()`

Исполняет код по истечении указанного промежутка времени.

Обработчики событий

`onblur`

Вызывается, когда окно теряет фокус.

`onerror`

Вызывается в случае возникновения JavaScript-ошибки.

`onfocus`

Вызывается, когда окно получает фокус.

`onload`

Вызывается, когда документ (или набор фреймов) полностью загружается.

`onresize`

Вызывается при изменении размера окна.

`onunload`

Вызывается, когда броузер выходит из текущего документа или набора фреймов.

Описание

Объект `Window` представляет окно или фрейм в броузере. Он подробно описан в главе 14. В клиентском JavaScript-коде объект `Window` выступает в качестве «глобального» объекта, и все выражения вычисляются в контексте текущего объекта `Window`. Это значит, что для обращения к текущему окну не требуется специального синтаксиса и свойства этого объекта можно использовать, как если бы они были глобальными переменными. Например, вместо записи *`window`*`.document` можно писать `document`. Аналогично можно вызывать методы текущего объекта окна, как если бы они были функциями, например `alert()` вместо *`window`*`.alert()`. Помимо перечисленных здесь свойств и методов объект `Window` реализует все глобальные функции, определяемые базовым языком JavaScript. Подробности см. в справочной статье об объекте `Global` в третьей части книги.

В объекте `Window` имеются свойства `window` и `self`, которые ссылаются на само окно. Они позволяют явно задать ссылку на окно. В дополнение к этим двум свойствам

свойства parent, top и frames[] ссылаются на другие объекты Window, имеющие отношение к текущему окну.

Так можно обратиться к фрейму в окне:

```
frames[i]      // Фреймы текущего окна
self.frames[i] // Фреймы текущего окна
w.frames[i]    // Фреймы определенного окна w
```

А к родительскому окну (или фрейму) данного фрейма – так:

```
parent      // Родительское окно по отношению к текущему
self.parent // Родительское окно по отношению к текущему
w.parent    // Родительское окно по отношению к заданному окну w
```

Следующий код позволяет обратиться к окну броузера верхнего уровня из любого фрейма, содержащегося в нем (независимо от глубины вложения):

```
top      // Окно верхнего уровня для текущего фрейма
self.top // Окно верхнего уровня для текущего фрейма
f.top    // Окно верхнего уровня для заданного фрейма f
```

Новые окна верхнего уровня создаются методом Window.open(). Вызывая этот метод, сохраните возвращаемое при вызове open() значение в переменной, чтобы иметь ссылку на новое окно. Свойство opener нового окна является ссылкой на открывшее его окно.

Как правило, методы объекта Window выполняют какие-либо манипуляции с окном или фреймом броузера. Стоит отметить методы alert(), confirm() и prompt(), которые служат для взаимодействия с пользователем через простые диалоговые окна.

Всесторонний обзор объекта Window приведен в главе 14, а подробности обо всех свойствах, методах и обработчиках событий можно найти в соответствующих справочных статьях.

См. также Document; описание объекта Global в третьей части книги; глава 14

Window.addEventListener()

См. описание метода Element.addEventListener()

Window.alert() JavaScript 1.0

отображает сообщение в диалоговом окне

Синтаксис

window.alert(*сообщение*)

Аргументы

сообщение

Строка простого (не HTML) текста, которая должна выводится в диалоговом окне, всплывающем поверх окна *window*.

Описание

Метод alert() показывает пользователю указанное сообщение в диалоговом окне. Диалоговое окно содержит кнопку OK, на которой пользователь может щелкнуть,

чтобы закрыть диалоговое окно. Обычно метод alert() выводит модальное диалоговое окно, и исполнение JavaScript-кода приостанавливается до тех пор, пока пользователь это окно не закроет.

Порядок использования

Пожалуй, чаще всего метод alert() служит для вывода сообщений об ошибках, когда пользователь вводит в какой-либо элемент некорректные данные. Предупреждающее окно может информировать пользователя об ошибке и объяснять, что должно быть исправлено, чтобы избежать ошибок в будущем.

Внешний вид диалогового окна alert() зависит от платформы, но, как правило, оно содержит графическое изображение, идентифицирующее какую-либо ошибку или предупреждение. Хотя alert() может выводить любое желаемое сообщение, предупреждающее графическое изображение в диалоговом окне означает, что этот метод не подходит для простых информационных сообщений вроде «Добро пожаловать на мою домашнюю страницу» или «Вы – 177-й посетитель за эту неделю!».

Обратите внимание: сообщение, выводимое в диалоговом окне, – это строка простого текста, а не форматированный HTML-текст. В сообщении может присутствовать символ перевода строки \n, разбивающий его на несколько строк. Можно выполнять некоторое простейшее форматирование с помощью пробелов и имитировать горизонтальные линии с помощью символов подчеркивания, но результат в диалоговом окне в значительной степени зависит от шрифта, потому в разных системах выглядит поразному.

См. также Window.confirm(), Window.prompt()

Window.attachEvent()

См. описание метода Element.attachEvent()

Window.blur() JavaScript 1.1

убирает клавиатурный фокус с окна верхнего уровня

Синтаксис

window.blur()

Описание

Метод blur() убирает в броузере фокус ввода с окна верхнего уровня, заданного объектом Window. Точно не определено, какому окну передается фокус в результате вызова этого метода. В некоторых броузерах и/или на некоторых платформах данный метод может не оказывать никакого эффекта.

См. также Window.focus()

Window.clearInterval() JavaScript 1.2

останавливает периодическое исполнение кода

Синтаксис

window.clearInterval(*intervalId*)

Аргументы

intervalId Значение, возвращаемое при соответствующем вызове `setInterval()`.

Описание

Метод `clearInterval()` останавливает периодическое исполнение кода, которое было начато вызовом метода `setInterval()`. В качестве аргумента *intervalId* должно передаваться значение, полученное при вызове метода `setInterval()`.

См. также `Window.setInterval()`

Window.clearTimeout() JavaScript 1.0

отменяет отложенное исполнение

Синтаксис

window.clearTimeout(*timeoutId*)

Аргументы

timeoutId

Значение, возвращаемое методом `setTimeout()` и идентифицирующее отменяемое действие.

Описание

Метод `clearTimeout()` отменяет исполнение кода, отложенное методом `setTimeout()`. Аргумент *timeoutId* – это значение, возвращаемое вызовом `setTimeout()` и идентифицирующее блок программного кода, отложенное исполнение которого отменяется.

См. также `Window.setTimeout()`

Window.close() JavaScript 1.0

закрывает окно броузера

Синтаксис

window.close()

Описание

Метод `close()` закрывает в броузере окно верхнего уровня, заданное объектом *window*. Окно может закрыть само себя методом `self.close()` или просто `close()`. Закрыты могут быть только те окна, которые были открыты JavaScript-сценарием. Благодаря этому злонамеренные сценарии не смогут закрыть окно броузера, открытое самим пользователем.

См. также `Window.open()`, свойства `closed` и `opener` объекта `Window`

Window.confirm() JavaScript 1.0

выводит вопрос, допускающий два варианта ответа: «да» или «нет»

Синтаксис

window.confirm(*вопрос*)

Аргументы

вопрос

Строка простого (не HTML) текста, выводимая в диалоговом окне. Обычно она представляет собой вопрос, на который требуется получить ответ пользователя.

Возвращаемое значение

Значение `true`, если пользователь щелкнул на кнопке OK, и `false`, если пользователь щелкнул на кнопке Отмена.

Описание

Метод `confirm()` выводит указанный *вопрос* в диалоговом окне, содержащем кнопки OK и Отмена, с помощью которых пользователь должен ответить на вопрос. Если пользователь щелкает на кнопке OK, метод `confirm()` возвращает `true`. Если пользователь щелкает на кнопке Отмена, метод `confirm()`возвращает `false`.

Метод `confirm()` выводит *модальное* диалоговое окно, т. е. окно, блокирующее любой ввод пользователя в главном окне броузера до тех пор, пока пользователь не закроет диалоговое окно, щелкнув на кнопке OK или Отмена. Данный метод возвращает значение, зависящее от реакции пользователя, поэтому исполнение JavaScript-кода приостанавливается на вызове `confirm()` и последующие инструкции не исполняются, пока пользователь не щелкнет на одной из кнопок в диалоговом окне.

Не существует способа изменить метки на кнопках диалогового окна (чтобы, например, вместо кнопок OK и Отмена в окне появлялись кнопки Да и Нет). Следовательно, необходимо сформулировать вопрос так, чтобы на него можно было «ответить» щелчком либо на кнопке OK, либо на кнопке Отмена.

См. также `Window.alert()`, `Window.prompt()`

Window.defaultStatus JavaScript 1.0

текст строки состояния, предлагаемый по умолчанию

Синтаксис

window.defaultStatus

Описание

Строковое свойство `defaultStatus` доступно для чтения и записи; оно определяет текст, выводимый по умолчанию в строке состояния окна. Обычно веб-броузеры отображают в строке состояния ход загрузки файла или адреса гиперссылок, на которые наводится указатель мыши. Если эти временные сообщения в строке состояния не выводятся, по умолчанию она обычно остается пустой. Однако в свойстве `defaultStatus` можно задать сообщение, выводимое в строке состояния по умолчанию, а прочитав значение свойства `defaultStatus`, можно узнать, что это за сообщение. Заданный текст может быть временно заменен другими сообщениями, например теми, которые выводятся, когда пользователь наводит указатель мыши на гиперссылку, но сообщение `defaultStatus` всегда появляется снова по завершении вывода временного сообщения.

В некоторых современных броузерах свойство `defaultStatus` не работает. Дополнительная информация приводится в справочной статье о свойстве `Window.status`.

См. также `Window.status`

Window.detachEvent()

См. описание метода Element.detachEvent()

Window.focus() JavaScript 1.1

передает фокус ввода окну верхнего уровня

Синтаксис

```
window.focus()
```

Описание

Метод `focus()` передает в броузере фокус ввода окну верхнего уровня, заданному объектом `Window`.

На большинстве платформ окно верхнего уровня при получении фокуса перемещается в стеке окон наверх.

См. также

`Window.blur()`

Window.getComputedStyle() DOM Level 2 CSS

возвращает стили CSS, использованные при отображении элемента

Синтаксис

```
CSS2Properties getComputedStyle(Element элемент,
                                String псевдоЭлемент);
```

Аргументы

элемент

Элемент документа, для которого запрашивается информация о стилях.

псевдоЭлемент

Псевдоэлемент CSS в виде строки, например `":before"` или `":first-line"`, либо `null`, если стили не определены.

Возвращаемое значение

Доступный только для чтения объект `CSS2Properties`, который представляет атрибуты стилей и их значения, используемые при отображении указанного элемента в данном окне. Любые числовые значения длины, содержащиеся в этом элементе, всегда измеряются в пикселах или абсолютных единицах, но не в относительных единицах и не в процентах.

Описание

Элемент документа может обретать информацию о стиле из встроенного атрибута `style` и из произвольного числа каскадных таблиц стилей. Прежде чем элемент будет отображен в окне, информация о стилях для этого элемента должна быть извлечена из каскадных таблиц стилей, а величины, выражаемые в относительных единицах (таких как проценты, или «ems»), должны быть «вычислены» и преобразованы в абсолютные значения.

Данный метод возвращает доступный только для чтения объект `CSS2Properties`, который представляет эти вычисленные CSS-стили. Спецификация DOM требует, чтобы

любые стили, представляющие длину, выражались в абсолютных единицах измерения, таких как дюймы или миллиметры. Однако на практике обычно все такие значения измеряются в пикселах; тем не менее нет никакой гарантии, что такое положение вещей сохранится в реализациях.

Сравните метод `getComputedStyle()` со свойством `style` объекта `HTMLElement`, которое предоставляет доступ только к встроенным стилям элемента. Величины в этом свойстве могут измеряться в любых единицах, но оно ничего не сообщает о стилях из таблиц стилей, которые применяются к элементу.

В Internet Explorer аналогичная функциональность доступна через нестандартное свойство `currentStyle`, имеющееся у каждого HTML-элемента.

См. также `CSS2Properties`, `HTMLElement`

Window.moveBy() JavaScript 1.2

перемещает окно на относительную позицию

Синтаксис

window.moveBy(*dx*, *dy*)

Аргументы

dx Количество пикселов, на которые окно должно переместиться вправо.

dy Количество пикселов, на которые окно должно переместиться вниз.

Описание

Метод `moveBy()` перемещает окно *window* на относительную позицию, заданную аргументами *dx* и *dy*. Соображения, касающиеся безопасности, приведены в справочной статье об объекте `Window.moveTo()`.

Window.moveTo() JavaScript 1.2

перемещает окно в абсолютную позицию

Синтаксис

window.moveTo(*x*, *y*)

Аргументы

x Координата X новой позиции окна.

y Координата Y новой позиции окна.

Описание

Метод `moveTo()` перемещает окно *window* таким образом, чтобы его верхний левый угол находился в позиции, заданной аргументами *x* и *y*. По соображениям безопасности броузеры могут ограничивать действие этого метода, запрещая ему перемещать окно за пределы экрана. Обычно считается дурным тоном перемещать окно броузера без явного требования пользователя. Как правило, сценарии применяют этот метод только для перемещения тех окон, которые были открыты самими сценариями методом `Window.open()`.

Window.onblur JavaScript 1.1

обработчик, вызываемый при потере окном фокуса ввода

Синтаксис

```
Function onblur
```

Описание

Свойство `onblur` объекта `Window` определяет функцию-обработчик события, которая вызывается, когда окно теряет фокус ввода.

Начальное значение этого свойства представляет собой функцию, содержащую JavaScript-инструкции, указанные через точку с запятой в атрибуте `onblur` тега `<body>` или `<frameset>`.

Начальное значение этого свойства представляет собой функцию, содержащую JavaScript-инструкции, указанные через точку с запятой в атрибуте

Порядок использования

Обработчик события `onblur` может использоваться для остановки анимации, когда окно не имеет фокуса ввода. Если окно не имеет фокуса, то теоретически это скорее всего означает, что пользователь не может видеть анимацию или не обращает на нее внимания.

См. также `Window.blur()`, `Window.focus()`, `Window.onfocus`; глава 17

Window.onerror JavaScript 1.1

обработчик, вызываемый при возникновении ошибки JavaScript

Синтаксис

Обработчик события `onerror` можно зарегистрировать следующим образом:

```
window.onerror=функция-обработчик
```

Броузер вызывает обработчик так:

```
window.onerror(сообщение, url, строка)
```

Аргументы

сообщение

Строка сообщения о произошедшей ошибке.

url

Строка, задающая URL-адрес документа, в котором произошла ошибка.

строка

Число, задающее номер строки, в которой произошла ошибка.

Возвращаемое значение

Значение `true`, если обработчик обеспечил обработку ошибки и JavaScript-коду не нужно предпринимать никаких дальнейших действий; значение `false`, если JavaScript-код должен вывести для этой ошибки стандартное диалоговое окно.

Описание

Свойство onerror объекта Window задает функцию обработки события, вызываемую, когда в коде, исполняемом в этом окне, возникает JavaScript-ошибка, не перехваченная инструкцией catch. Программист может сам реализовать обработку ошибок, написав собственный обработчик onerror.

Можно определить обработчик ошибок onerror для окна, установив свойство onerror объекта Window равным соответствующей функции. Обратите внимание: в отличие от других обработчиков JavaScript-ошибок, обработчик onerror не может быть определен в атрибуте onerror тега <body>.

Вызываемому обработчику onerror передаются три аргумента: строка с текстом сообщения об ошибке; строка, содержащая URL-адрес документа, в котором произошла ошибка, и число – номер строки, где произошла ошибка. Функция обработки ошибки может делать с этими аргументами все, что угодно; может, например, вывести собственное сообщение об ошибке или зарегистрировать ее любым другим способом. Завершившись, обработчик ошибок должен вернуть true, если обработка ошибки выполнена и от JavaScript не требуется каких-либо дальнейших действий, или false, если обработчик только заметил или как-то зарегистрировал ошибку, но броузер должен продолжить обработку ошибки.

Window.onfocus JavaScript 1.1

обработчик, вызываемый при получении окном фокуса ввода

Синтаксис

```
Function onfocus
```

Описание

Свойство onfocus объекта Window задает функцию обработки события, вызываемую, когда окно получает фокус ввода.

Начальное значение этого свойства представляет собой функцию, содержащую JavaScript-инструкции, указанные через точку с запятой в атрибуте onfocus тега <body> или <frameset>.

Порядок использования

Обработчик события onfocus можно использовать для запуска анимации, а обработчик onblur – для ее остановки, чтобы анимация работала, только когда пользователь обращает внимание на окно.

См. также Window.blur(), Window.focus(), Window.onblur; глава 17

Window.onload JavaScript 1.0

обработчик, вызываемый по завершении загрузки документа

Синтаксис

```
Function onload
```

Описание

Свойство onload объекта Window задает функцию обработки события, которая вызывается после полной загрузки документа или набора фреймов в свое окно или фрейм.

Начальное значение этого свойства представляет собой функцию, содержащую JavaScript-инструкции, перечисленные через точку с запятой в атрибуте onload тега <body> или <frameset>.

Если вызван обработчик события onload, можно быть уверенным, что документ загружен целиком и, следовательно, все сценарии в документе выполнены, все функции в сценариях определены, а все формы и другие элементы документа уже проанализированы и доступны через объект Document.

Для регистрации нескольких функций-обработчиков события onload можно использовать метод Window.addEventListener() или Window.attachEvent().

См. также Window.onunload; раздел 13.5.7; пример 17.7; глава 17

Window.onresize JavaScript 1.2

обработчик, вызываемый при изменении размеров окна

Синтаксис

```
Function onresize
```

Описание

Свойство onresize объекта Window задает функцию обработки ошибок, вызываемую, когда пользователь изменяет размер окна или фрейма.

Начальное значение этого свойства представляет собой функцию, содержащую JavaScript-инструкции, перечисленные через точку с запятой в атрибуте onresize тега <body> или <frameset>, который определяет окно.

Window.onunload DOM Level 0

обработчик, вызываемый, когда броузер покидает страницу

Синтаксис

```
Function onunload
```

Описание

Свойство onunload объекта Window задает функцию обработки событий, вызываемую, когда броузер «выгружает» документ или набор фреймов, готовясь к загрузке нового документа.

Начальное значение этого свойства представляет собой функцию, содержащую JavaScript-инструкции, перечисленные через точку с запятой в атрибуте onunload тега <body> или <frameset>.

Обработчик события onunload дает возможность полностью обновить состояние броузера перед загрузкой нового документа.

Обработчик onunload() вызывается, когда пользователь дает броузеру команду покинуть текущую страницу и переместиться на другую. Следовательно, в обычных обстоятельствах вряд ли имеет смысл задерживать загрузку запрошенной страницы, генерируя всплывающие диалоговые окна (например, методом Window.confirm() или Window.prompt()) из обработчика события onunload.

См. также Window.onload; глава 17

Window.open() JavaScript 1.0

открывает новое окно броузера или находит окно с указанным именем

Синтаксис

window.open(*url*, *имя*, *элементы*, *замена*)

Аргументы

url

Необязательная строка, задающая URL-адрес документа, который должен отображаться в новом окне. Если этот аргумент опущен или задана пустая строка, в новом окне никакой документ не появится.

имя

Необязательная строка алфавитно-цифровых символов и символов подчеркивания, задающая имя нового окна. Это имя может выступать в качестве значения атрибута `target` HTML-тегов `<a>` и `<form>`. Если этот аргумент указывает на уже существующее окно, метод `open()` не создает новое окно, а просто возвращает ссылку на окно с этим именем. В этом случае аргумент *элементы* игнорируется.

элементы

Строка, указывающая, какие элементы стандартного окна броузера должны присутствовать в новом окне. Формат этой строки приведен далее в подразделе «Элементы окна». Данный аргумент может отсутствовать, и тогда новое окно будет иметь все стандартные графические элементы.

замена

Необязательный логический аргумент, указывающий, должен ли URL-адрес документа, загруженного в новое окно, появиться в списке истории просмотра в виде новой записи или заменить текущую запись. Если этот аргумент равен `true`, новая запись в списке истории просмотра не создается. Обратите внимание: этот аргумент используется только в случае изменения содержимого существующего окна.

Возвращаемое значение

Ссылка на объект `Window`, который может быть либо только что созданным, либо новым окном, в зависимости от аргумента *имя*.

Описание

Метод `open()` отыскивает существующее или открывает новое окно броузера. Если аргумент *имя* задает имя существующего окна, то возвращается ссылка на это окно. Возвращаемое окно содержит документ, URL-адрес которого указан в аргументе *url*, а аргумент *элементы* игнорируется. В JavaScript это единственный метод, позволяющий получить ссылку на окно, для которого известно только имя.

Если аргумент *имя* не указан или окно с этим именем еще не существует, метод `open()` создает новое окно броузера. Созданное окно содержит документ, URL-адрес которого задан аргументом *url*, имеет имя, заданное аргументом *имя*, а также размеры и управляющие элементы, заданные аргументом *элементы* (формат этого аргумента описан в следующем подразделе). Если *url* – это пустая строка, метод `open()` открывает пустое окно.

Аргумент *имя* задает имя нового окна. Это имя может содержать только алфавитно-цифровые символы и символы подчеркивания. Оно может выступать в качестве значения атрибута `target` тега `<a>` или `<form>` в HTML-коде для принудительного вывода документов в это окно.

В случае использования метода Window.open() для загрузки нового документа в окно с указанным именем методу можно передать аргумент *замена*, указывающий, должна ли для нового документа создаваться новая запись в истории просмотра окна или URL-адрес документа должен заменить текущую запись в истории окна. Если аргумент *замена* равен true, то URL-адрес нового документа заменяет старый. Если этот аргумент равен false или не указан, то URL-адрес нового документа появляется в истории просмотра окна в качестве новой записи. Этот аргумент обеспечивает методу функциональность, во многом схожую с функциональностью метода Location.replace().

Не путайте вызовы Window.open() и Document.open() — эти два вызова выполняют совершенно разные функции. Для ясности следует отказаться от вызова open() в пользу Window.open(). В обработчиках событий, определенных как HTML-атрибуты, вызов open() обычно интерпретируется как Document.open(), поэтому в таких случаях необходимо вызывать метод Window.open() явно.

Элементы окна

Аргумент *элементы* — это список перечисленных через запятую элементов окна, которые должны появиться в окне. Если этот необязательный аргумент пуст или не указан, то в окне будут присутствовать все элементы. В то же время, если в аргументе *элементы* указан хотя бы один элемент, любые элементы, отсутствующие в списке, будут отсутствовать и в окне. Строка не может содержать пробелы или другие пробельные символы. Каждый элемент в списке имеет следующий формат:

```
элемент[=значение]
```

Для большинства элементов *значение* может быть равно yes или no. Для этих элементов знак равенства и *значение* могут опускаться: если элемент указан, подразумевается yes, если не указан — no. Для элементов width и height *значение* является обязательным, задавая размер окна в пикселах.

Далее перечислены наиболее часто употребляемые элементы и их назначение:

height

Высота области отображения окна в пикселах.

left

Координата X окна в пикселах.

location

Поле ввода для набора URL-адреса непосредственно в броузере.

menubar

Строка меню.

resizable

Если этот элемент не указан или равен no, окно не имеет рамки, «ухватившись» за которую мышью можно изменить его размеры. (В зависимости от платформы у пользователя все равно остается та или иная возможность изменения размера окна.) Обратите внимание на распространенную ошибку: очень часто этот элемент пишут как «resizeable» — с лишней буквой «e».

scrollbars

Включает режим вывода горизонтальных и вертикальных полос прокрутки, когда они необходимы.

`status`

Строка состояния.

`toolbar`

Панель инструментов броузера с кнопками Назад, Вперед и др.

`top`

Координата Y окна в пикселах.

`width`

Ширина области отображения документа в пикселах.

См. также

`Location.replace()`, `Window.close()`, свойства `closed` и `opener` объекта `Window`

Window.print() JavaScript 1.5

печатает документ

Синтаксис

window`.print()`

Описание

На вызов метода `print()` броузер реагирует так же, как если бы пользователь щелкнул на кнопке Печать. Обычно после этого появляется диалоговое окно, позволяющее пользователю отменить операцию печати или выполнить дополнительную настройку.

Window.prompt() JavaScript 1.0

позволяет получить строку, введенную в диалоговом окне

Синтаксис

window`.prompt(`*сообщение*`, `*по_умолчанию*`)`

Аргументы

сообщение

Строка простого (не HTML) текста, которая должна быть показана в диалоговом окне. Она должна приглашать пользователя ввести необходимую информацию.

по_умолчанию

Строка, отображаемая в диалоговом окне по умолчанию. Чтобы метод `prompt()` показал пустое окно, надо передать пустую строку (`""`).

Возвращаемое значение

Строка, введенная пользователем, пустая строка, если пользователь ничего не ввел, или `null`, если пользователь щелкнул на кнопке Отмена.

Описание

Метод `prompt()` выводит указанное *сообщение* в диалоговом окне, которое также содержит поле ввода и кнопки OK и Отмена. Зависящее от платформы графическое изображение в диалоговом окне указывает пользователю, что от него требуется выполнить ввод данных.

Если пользователь щелкнет на кнопке Отмена, метод prompt() вернет null. Если пользователь щелкнет на кнопке OK, метод prompt() вернет значение, указанное в этот момент в поле ввода.

Метод prompt() выводит модальное диалоговое окно, блокирующее любой пользовательский ввод в главном окне броузера до тех пор, пока пользователь не избавится от диалогового окна, щелкнув на кнопке OK или Отмена. Данный метод возвращает значение, зависящее от выбора пользователя в диалоговом окне, поэтому на время вызова метода исполнение JavaScript-кода приостанавливается, и последующие инструкции не исполняются до тех пор, пока пользователь не щелкнет на кнопке в диалоговом окне.

См. также Window.alert(), Window.confirm()

Window.removeEventListener()

См. описание метода Element.removeEventListener()

Window.resizeBy() JavaScript 1.2

изменяет размер окна на относительное значение

Синтаксис

window.resizeBy(*dw*, *dh*)

Аргументы

dw Количество пикселов, на которые должна быть изменена ширина окна.

dh Количество пикселов, на которые должна быть изменена высота окна.

Описание

Метод resizeBy() изменяет размер окна *window* на относительное значение, заданное аргументами *dh* и *dw*. Соображения, касающиеся безопасности и правил использования, приводятся в справочной статье о методе Window.resizeTo().

Window.resizeTo() JavaScript 1.2

изменяет размер окна

Синтаксис

window.resizeTo(*ширина*, *высота*)

Аргументы

ширина Желаемое значение ширины окна.

высота Желаемое значение высоты окна.

Описание

Метод resizeTo() изменяет размер окна *window* так, чтобы оно имело указанные в пикселах *ширину* и *высоту*. По соображениям безопасности броузер может не дать этому методу сделать высоту или ширину окна слишком маленькими. По соображениям удобства обычно считается дурным тоном изменять размер окна, заданный пользователем непосредственно. Если окно было создано сценарием, тогда сценарий может из-

менять его размеры, но изменять из сценария размеры окна, в которое он был загружен, – это неуважение к пользователю.

Window.scrollBy() JavaScript 1.2

прокручивает документ на относительную величину

Синтаксис

window.scrollBy(*dx*, *dy*)

Аргументы

dx Количество пикселов, на которые документ должен быть прокручен вправо.

dy Количество пикселов, на которые документ должен быть прокручен вниз.

Описание

Метод scrollBy() прокручивает документ, отображаемый в окне, на относительную величину, заданную аргументами *dx* и *dy*.

Window.scrollTo() JavaScript 1.2

прокручивает документ

Синтаксис

window.scrollTo(*x*, *y*)

Аргументы

x Координата X, в которой должен оказаться левый край области отображения документа в окне.

y Координата Y, в которой должен оказаться верхний край области отображения документа в окне.

Описание

Метод scrollTo() прокручивает документ, отображаемый в окне *window*, так, чтобы точка в документе, заданная координатами *x* и *y*, если это возможно, оказалась в левом верхнем углу.

Window.setInterval() JavaScript 1.2

периодически выполняет указанный код

Синтаксис

window.setInterval(*код*, *интервал*)

Аргументы

код

JavaScript-функция или строка JavaScript-кода, который периодически должен исполняться. Если строка содержит несколько инструкций, они должны отделяться друг от друга точками с запятой. В IE 4 (но не в более поздних версиях) этот аргумент может быть только строкой.

интервал

Целое значение, задающее интервал в миллисекундах между вызовами *кода*.

Возвращаемое значение

Значение, которое может быть передано в метод `Window.clearInterval()` для отмены периодического исполнения *кода*.

Описание

Метод `setInterval()` периодически исполняет JavaScript-функцию или JavaScript-инструкции в строке, заданные в аргументе *код*, через *интервал* миллисекунд.

Возвращаемое значение метода `setInterval()` может быть передано методу `Window.clearInterval()` для прекращения периодического исполнения *кода*.

Метод `setInterval()` напоминает `setTimeout()`. Последний используется тогда, когда надо лишь отложить исполнение кода на определенное время и его не требуется периодически запускать.

См. также `Window.clearInterval()`, `Window.setTimeout()`

Window.setTimeout() JavaScript 1.0

отложить исполнение кода

Синтаксис

window`.setTimeout(`*код*`, `*задержка*`)`

Аргументы

код

JavaScript-функция или строка JavaScript-кода, который должен быть исполнен по истечении указанной *задержки*. Если данный аргумент является строкой, содержащей несколько инструкций, они должны отделяться друг от друга точками с запятой. В IE 4 этот аргумент может быть только строкой – передача методу имени функции в этой версии броузера не поддерживается.

задержка

Пауза в миллисекундах перед выполнением JavaScript-инструкций в строке аргумента *код*.

Возвращаемое значение

Загадочное значение («идентификатор времени ожидания»), которое может быть передано в метод `clearTimeout()` для отмены исполнения кода.

Описание

Метод `setTimeout()` откладывает исполнение JavaScript-функции или JavaScript-инструкций в строке *код* на количество миллисекунд, указанное в аргументе *задержка*. Обратите внимание: аргумент *код* будет исполнен только один раз. Для периодического исполнения *кода* сам этот *код* должен содержать вызов метода `setTimeout()`, регистрирующий повторное исполнение кода.

Инструкции в строке *код* исполняются в контексте объекта `Window`. Если *код* – это функция, объект `Window` становится значением ключевого слова `this`. Если *код* – это строка, он исполняется в глобальной области видимости, где объект `Window` является единственным объектом в цепочке областей видимости. Это верно, даже если вызов `setTimeout()` выполняется из функции, обладающей более длинной цепочкой областей видимости.

См. также Window.clearTimeout(), Window.setInterval()

Window.status

JavaScript 1.0

временное сообщение в строке состояния

Синтаксис

```
String status
```

Описание

Строковое свойство status доступно для чтения и записи; оно задает сообщение, которое должно появиться в строке состояния окна. Как правило, сообщение появляется только на ограниченное время – пока, например, оно не будет заменено другим сообщением или пока пользователь не переместит указатель мыши в другую область окна. Когда сообщение, заданное в свойстве status, удаляется, строка состояния возвращается к своему состоянию, заданному по умолчанию – это может быть пустая строка или сообщение, указанное в свойстве defaultStatus.

К моменту написания этих строк в большинстве броузеров был введен запрет на изменение строки состояния. Это было сделано из соображений безопасности во избежание попыток подлога, когда вместо истинного адреса гиперссылки в строке состояния отображается подложный адрес.

См. также Window.defaultStatus

XMLHttpRequest

Firefox 1.0, Internet Explorer 7.0, Safari 1.2, Opera 7.60

позволяет выполнять HTTP-запросы и получать ответы **Object→XMLHttpRequest**

Конструктор

```
new XMLHttpRequest()                    // Все броузеры, за исключением IE 5 и IE 6
new ActiveXObject("Msxml2.XMLHTTP")     // IE
new ActiveXObject("Microsoft.XMLHTTP") // IE с устаревшими системными библиотеками
```

Свойства

```
readonly short readyState
```

Состояние HTTP-запроса. В момент создания объекта XMLHttpRequest это свойство приобретает значение 0, а в процессе исполнения запроса и вплоть до получения ответа это значение увеличивается до 4. Каждому из пяти значений состояния присвоено неформальное имя – эти имена перечислены в следующей таблице вместе с их описаниями:

Состояние	Имя	Описание
0	Uninitialized	Начальное состояние. Объект XMLHttpRequest только что создан или возвращен в исходное состояние вызовом метода abort()
1	Open	Метод open() уже вызван, но обращения к методу send() еще не было. Запрос еще не отправлен
2	Sent	Вызван метод send() и HTTP-запрос отправлен веб-серверу. Ответ еще не получен

Состояние	Имя	Описание
3	Receiving	Все заголовки ответа уже приняты. Продолжается прием тела ответа, но прием еще не завершился
4	Loaded	Ответ на HTTP-запрос получен полностью

Значение свойства readyState никогда не уменьшается, если в процессе приема ответа не был вызван метод abort() или open(). Каждый раз, когда увеличивается значение этого свойства, вызывается обработчик события onreadystatechange.

readonly String responseText

Тело ответа (не включая заголовки), принятое от сервера к текущему моменту, или пустая строка, если данные еще не приняты. Если значение свойства readyState меньше 3, данное свойство возвращает часть ответа, которая была принята к текущему моменту. Если значение свойства readyState равно 4, это свойство содержит полное тело ответа.

Если в ответе имеется заголовок, определяющий кодировку символов в теле ответа, используется эта кодировка, в противном случае предполагается кодировка UTF-8.

readonly Document responseXML

Ответ на запрос, который интерпретируется как XML-документ и возвращается в виде объекта Document. Это свойство содержит значение null, если не соблюдены все три следующих условия:

- значение readyState равно 4;
- ответ включает заголовок Content-Type со значением "text/xml", "application/xml" или любым другим значением, оканчивающимся на «+xml», что свидетельствует о том, что ответ представляет собой XML-документ;
- тело ответа содержит корректно оформленный XML-текст, при синтаксическом разборе которого не возникает ошибок.

readonly short status

HTTP-код состояния, полученный от сервера, например, 200 – в случае успеха и 404 – в случае ошибки отсутствия документа. Попытка чтения этого свойства, пока значение свойства readyState меньше 3, вызывает исключение.

readonly String satusText

Это свойство содержит текст, соответствующий HTTP-коду состояния в ответе. То есть когда свойство status имеет значение 200, это свойство содержит строку "OK", а когда 404 – строку "Not Found". Как и в случае со свойством status, попытка чтения этого свойства, пока значение свойства readyState меньше 3, вызывает исключение.

Методы

abort()

Отменяет исполнение текущего запроса, закрывает соединение и останавливает сетевую активность.

getAllResponseHeaders()

Возвращает HTTP-заголовки ответа в виде неразобранной строки.

getResponseHeader()

Возвращает HTTP-заголовок с заданным именем.

`open()`

Инициализирует параметры HTTP-запроса, такие как URL-адрес, HTTP-метод, но не отправляет запрос.

`send()`

Отправляет HTTP-запрос с использованием параметров, указанных при вызове метода `open()`, и необязательного тела запроса, которое может передаваться данному методу.

`setRequestHeader()`

Устанавливает или добавляет заголовок HTTP-запроса в открытый, но еще не отправленный запрос.

Обработчики событий

`onreadystatechange`

Функция-обработчик события, вызываемая всякий раз, когда изменяется значение свойства `readyState`. Может многократно вызываться для значения 3 в свойстве `readyState`.

Описание

Объект `XMLHttpRequest` позволяет из клиентских JavaScript-сценариев запускать HTTP-запросы и получать от веб-сервера ответы (которые не обязательно должны быть в формате XML). Объект `XMLHttpRequest` подробно рассматривается в главе 20, там же можно найти множество примеров применения этого объекта.

Объект `XMLHttpRequest` полностью переносим и хорошо поддерживается всеми современными броузерами. Единственное, в чем наблюдаются отличия между броузерами, – это порядок создания объекта `XMLHttpRequest`. В Internet Explorer для этих целей необходимо использовать конструктор `ActiveXObject()`, как было отмечено в подразделе «Синтаксис».

После создания объекта `XMLHttpRequest` он обычно используется следующим образом:

1. Вызывается метод `open()`, с помощью которого задаются URL-адрес и метод передачи запроса (обычно GET или POST). Как правило, при вызове метода `open()` указывается тип запроса – синхронный или асинхронный.
2. Если был выбран асинхронный режим отправки запроса, в свойство `onreadystatechange` записывается ссылка на функцию, которая будет вызываться в процессе исполнения запроса.
3. Вызывается метод `setRequestHeader()`, если необходимо указать дополнительные параметры запроса.
4. Вызовом метода `send()` выполняется отправка запроса веб-серверу. Если был выбран метод отправки POST, этому методу можно дополнительно передать тело запроса. Если был выбран синхронный режим исполнения запроса, метод `send()` блокирует исполнение сценария, пока полностью не будет получен ответ и в свойстве `readyState` не появится значение 4. В противном случае функция-обработчик события `onreadystatechange` должна дождаться, пока значение свойства `readyState` не будет равно 4 (или хотя бы 3).
5. После того как метод `send()` вернет управление в случае синхронных запросов или свойство `readyState` достигнет значения 4 в случае асинхронных запросов, можно проанализировать ответ, полученный от сервера. В первую очередь следует проверить код в свойстве `status`, чтобы убедиться, что запрос завершился успехом.

В этом случае методом getResponseHeader() или getesponseHeaders() следует извлечь значения из заголовка ответа и с помощью свойства responseText или responseXML получить тело ответа.

К моменту написания этих строк объект XMLHttpRequest еще не был не стандартизован, но работы по его стандартизации консорциум W3C уже начал. Данное описание базируется на проектах стандартов. Существующие ныне реализации XMLHttpRequest вполне работоспособны, хотя в них и наблюдаются незначительные отступления от стандарта. Например, реализация может вернуть значение null, там где стандарт требует возвращать пустую строку, или устанавливать значение свойства readyState равным 3, не гарантируя доступность всех заголовков ответа.

См. также Глава 20

XMLHttpRequest.abort()

отменяет исполнение HTTP-запроса

Синтаксис

```
void abort()
```

Описание

Данный метод возвращает объект XMLHttpRequest в исходное состояние, соответствующее значению 0 в свойстве readyState, и отменяет любые запланированные сетевые взаимодействия. Этот метод может потребоваться, например, если запрос исполняется слишком долго и надобность в получении ответа уже отпала.

XMLHttpRequest.getAllResponseHeaders()

возвращает HTTP-заголовки ответа в виде неразобранной строки

Синтаксис

```
String getAllResponseHeaders()
```

Возвращаемое значение

Если свойство readyState имеет значение меньше 3, этот метод возвращает null. В противном случае возвращаются все HTTP-заголовки ответа (но без строки состояния), полученные от сервера. Заголовки возвращаются в виде единственной строки и отделяются друг от друга комбинацией символов \r\n.

XMLHttpRequest.getResponseHeader()

возвращает значение HTTP-заголовка с заданным именем

Синтаксис

```
String getResponseHeader(заголовок)
```

Аргументы

заголовок

Имя HTTP-заголовка ответа, значение которого должно быть получено. Символы в *заголовке* могут указываться в любом регистре – сравнение, выполняемое при поиске требуемого заголовка, выполняется без учета регистра символов.

Возвращаемое значение

Значение заданного HTTP-заголовка ответа или пустая строка, если требуемый заголовок не был получен или значение свойства readyState меньше 3. Если было принято несколько заголовков с указанным именем, значения этих заголовков объединяются в одну строку через запятую и пробел.

XMLHttpRequest.onreadystatechange

обработчик события, вызываемый при изменении значения свойства readyState

Синтаксис

```
Function onreadystatechange
```

Описание

Это свойство задает функцию-обработчик, которая вызывается всякий раз, когда изменяется значение свойства readyState. Она может вызываться (хотя это и не обязательно) несколько раз, пока свойство readyState имеет значение 3, чтобы обеспечить возможность обратной связи в процессе загрузки.

Обычно обработчик onreadystatechange проверяет свойство readyState объекта XMLHttpRequest, чтобы не пропустить момент, когда его значение достигнет 4. После этого обработчик выполняет некоторые действия со свойством responseText или responseXML.

Стандартом не указывается, может ли функция принимать какие-либо аргументы. В частности, отсутствует какая-либо стандартная возможность получить ссылку на объект XMLHttpRequest, в котором был зарегистрирован обработчик. Это означает, что нет стандартного способа написать универсальную функцию-обработчик, которая могла бы использоваться для одновременного обслуживания нескольких запросов.

Объект XMLHttpRequest должен следовать модели событий DOM и реализовывать метод addEventListener() для регистрации обработчиков события onreadystatechange. (См., например, справочную статью о методе Event.addEventListener().) Но поскольку IE не поддерживает модель событий DOM и надобность в нескольких обработчиках событий для одного запроса возникает очень редко, гораздо безопаснее просто присваивать функцию-обработчик свойству onreadystatechange.

XMLHttpRequest.open()

инициализирует параметры HTTP-запроса

Синтаксис

```
void open(String метод,
          String url,
          boolean асинхронный,
          String пользователь, String пароль)
```

Аргументы

метод

HTTP-метод, используемый для отправки запроса. Среди наиболее устоявшихся методов можно назвать GET, POST и HEAD. Реализации могут также поддерживать дополнительные методы.

url

URL-адрес, который является предметом запроса. Большинство броузеров следуют политике общего происхождения (см. раздел 13.8.2) и требуют, чтобы данный URL-адрес содержал те же самые имя хоста и номер порта, откуда был получен документ со сценарием. Разрешение относительных URL-адресов производится обычным образом с использованием URL-адреса документа со сценарием.

асинхронный

Определяет режим исполнения запроса: синхронный или асинхронный. Если этот аргумент имеет значение `false`, запрос исполняется в синхронном режиме и последующий вызов `send()` блокируется, пока ответ не получен полностью. Если этот аргумент имеет значение `true` или опущен, запрос исполняется в асинхронном режиме и в этом случае обычно требуется определить обработчик события `onreadystatechange`.

пользователь, *пароль*

Эти необязательные аргументы требуются при использовании URL-адреса документа, для доступа к которому необходимо проходить процедуру авторизации. Если эти аргументы указаны, они переопределяют информацию об авторизации, содержащуюся в самом URL-адресе.

Описание

Этот метод инициализирует параметры запроса для последующего использования методом `send()`. Он устанавливает свойство `readyState` в значение 1, удаляет заголовки запроса, определенные ранее, а также заголовки ранее полученного ответа и устанавливает свойства `responseText`, `responseXML`, `status` и `statusText` в значения, предлагаемые по умолчанию. Этот метод безопасно может вызываться, когда свойство `readyState` имеет значение 0 (сразу после создания объекта `XMLHttpRequest` или после вызова метода `abort()`), а также, когда свойство `readyState` имеет значение 4 (после того как получен ответ). Для других состояний объекта поведение метода `open()` не определено.

Метод `open()` ничего другого не делает, кроме как сохраняет параметры для использования методом `send()` и инициализирует объект `XMLHttpRequest`. При этом следует отметить, что реализации при обращении к этому методу обычно не устанавливают сетевое соединение с веб-сервером.

См. также `XMLHttpRequest.send()`; глава 20

XMLHttpRequest.send()

отправляет HTTP-запрос

Синтаксис

```
void send(Object тело)
```

Аргументы

тело

Если с помощью метода `open()` в качестве HTTP-метода передачи был определен метод POST или PUT, данный аргумент определяет тело запроса в виде строки, объекта `Document` или, если тело запроса не требуется, `null`. В случае всех остальных методов данный аргумент не используется и должен иметь значение `null`. (Некоторые реализации не допускают отсутствие этого аргумента.)

Описание

Этот метод инициирует исполнение HTTP-запроса. Если перед этим не вызывался метод `open()` или, если быть более точным, если значение свойства `readyState` не равно 1, метод `send()` возбуждает исключение. В противном случае он начинает исполнение HTTP-запроса, который состоит из:

- HTTP-метода, URL-адреса и информации об авторизации (в случае необходимости), определенные предшествующим вызовом метода `open()`;
- заголовков запроса, если они были определены предшествующим вызовом метода `setRequestHeader()`;
- значения аргумента *тело*, переданного данному методу.

Как только запрос начинает исполняться, метод `send()` устанавливает свойство `readyState` в значение 2 и вызывает обработчик события `onreadystatechange`.

Если в предшествующем вызове метода `open()` аргумент *асинхронный* имел значение `false`, данный метод блокируется и не возвращает управление, пока значение свойства `readyState` не станет равно 4 и ответ сервера не будет получен полностью. В противном случае, если аргумент *асинхронный* имеет значение `true` или если этот аргумент опущен, метод `send()` немедленно возвращает управление, а ответ сервера обрабатывается в фоновом потоке исполнения, как это описывается далее.

Если сервер сообщает о необходимости перенаправления, метод `send()` или фоновый поток исполнения автоматически реализует перенаправление. Когда будут получены HTTP-заголовки ответа, метод `send()` или фоновый поток исполнения переустановит свойство `readyState` в значение 3 и вызовет обработчик события `onreadystatechange`. Если ответ слишком длинный, `send()` или фоновый поток исполнения могут вызывать обработчик события `onreadystatechange` для состояния 3 неоднократно – это обстоятельство можно использовать для вывода информации о ходе загрузки. Наконец, когда завершится прием ответа, `send()` или фоновый поток исполнения переустановит свойство `readyState` в значение 4 и в последний раз вызовет обработчик события.

См. также `XMLHttpRequest.open()`; глава 20

XMLHttpRequest.setRequestHeader()

добавляет HTTP-заголовок в запрос

Синтаксис

```
void setRequestHeader(String имя, String значение)
```

Аргументы

имя

Имя устанавливаемого заголовка. Этот аргумент не должен содержать пробелы, двоеточия, символы перевода строки или листа.

значение

Значение заголовка. Данный аргумент не должен содержать символы перевода строки или листа.

Описание

Метод `setRequestHeader()` определяет HTTP-заголовок, который должен быть включен в запрос, передаваемый последующим вызовом метода `send()`. Этот метод может

вызываться, только когда свойство readyState имеет значение 1, т. е. после вызова метода open(), но перед вызовом метода send().

Если заголовок с заданным *именем* уже был определен, новым значением заголовка станет прежнее значение заголовка плюс запятая с пробелом и новое *значение*, переданное методу.

Если методу open() была передана информация об авторизации, объект XMLHttpRequest автоматически добавит заголовок Authorization. Этот заголовок может быть также добавлен методом setRequestHeader(). Аналогичным образом, если броузер сохранял cookies, ассоциированные с URL-адресом, переданным методу open(), соответствующие заголовки Cookie или Cookie2 автоматически будут добавлены в запрос. Методом setRequestHeaders() можно добавить в эти заголовки дополнительные cookie-файлы. Кроме того, объект XMLHttpRequest может предоставлять значение по умолчанию для заголовка User-Agent. Если этот заголовок устанавливается с помощью этого метода, любое значение будет добавлено к значению, предлагаемому по умолчанию.

Некоторые заголовки запроса устанавливаются объектом XMLHttpRequest автоматически с целью соблюдения требований протокола HTTP и не могут быть установлены с помощью описываемого метода. Помимо других, сюда относятся заголовки, имеющие отношение к представителю (proxy):

```
Host
Connection
Keep-Alive
Accept-Charset
Accept-Encoding
If-Modified-Since
If-None-Match
If-Range
Range
```

См. также XMLHttpRequest.getResponseHeader()

XMLSerializer

Firefox 1.0, Safari 2.01, Opera 7.60

сериализует XML-документы и узлы **Object→XMLSerializer**

Конструктор

```
new XMLSerializer()
```

Методы

```
serializeToString()
```

Этот метод экземпляра выполняет фактическую сериализацию.

Описание

Объект XMLSerializer позволяет преобразовать, или «сериализовать», XML-документ или объект Node в строку разметки XML. Прежде чем использовать объект XMLSerializer, необходимо создать его экземпляр вызовом конструктора без аргументов, а затем обратиться к методу serializeToString():

```
var text = (new XMLSerializer()).serializeToString(элемент);
```

Internet Explorer не поддерживает объект XMLSerializer. Вместо него XML-текст можно получить с помощью свойства xml объекта Node.

См. также DOMParser, Node; глава 21

XMLSerializer.serializeToString()

преобразует XML-документ или узел в строку

Синтаксис

```
String serializeToString(Node узел)
```

Аргументы

узел

Узел XML-документа, который требуется преобразовать в строку. Это может быть объект Document или любой элемент внутри документа.

Возвращаемое значение

Строка в формате XML, которая после сериализации представляет указанный *узел* со всеми его потомками.

XPathExpression

Firefox 1.0, Safari 2.01, Opera 9

скомпилированный XPath-запрос **Object→XPathExpression**

Методы

evaluate()

Вычисляет выражение в контексте заданного узла.

Описание

Объект XPathExpression создается вызовом метода Document.createExpression() и является скомпилированным представлением XPath-запроса. Вычисление значения выражения производится в контексте конкретного узла документа с помощью метода evaluate(). Если выражение требуется вычислить всего один раз, можно воспользоваться методом Document.evaluate(), который компилирует и вычисляет выражение за один шаг.

Internet Explorer не поддерживает объект XPathExpression. Альтернативные методы, реализованные в IE, описываются в справочных статьях о методах Node.selectNodes() и Node.selectSingleNode().

См. также

Document.createExpression(), Document.evaluate(), Node.selectNodes(), Node.selectSingleNode(); глава 21

XPathExpression.evaluate()

вычисляет скомпилированный XPath-запрос

Синтаксис

```
XPathResult evaluate(Node контекстныйУзел,
                     short тип,
                     XPathResult результат)
```

Аргументы

контекстныйУзел

Узел (или документ), в контексте которого должен быть выполнен запрос.

тип

Желаемый тип результата. Этот аргумент должен содержать значение одной из констант, определяемых объектом `XPathResult`.

результат

Объект `XPathResult`, в который должны быть помещены результаты запроса, или `null`, если необходимо, чтобы метод `evaluate()` создал и вернул новый объект `XPathResult`.

Возвращаемое значение

Объект `XPathResult`, в котором хранятся результаты запроса. Этот объект передается в аргументе *результат* или создается заново, если в аргументе *результат* было передано значение `null`.

Описание

Метод вычисляет выражение `XPathExpression` в контексте заданного узла или документа и возвращает результат в объекте `XPathResult`. Порядок извлечения значений из возвращаемого объекта приводится в справочной статье об объекте `XPathResult`.

См. также `Document.evaluate()`, `Node.selectNodes()`, `XPathResult`; главу 21

XPathResult

Firefox 1.0, Safari 2.01, Opera 9

результат XPath-запроса **Object→XPathResult**

Константы

Следующие константы определяют возможные типы результатов, которые может вернуть XPath-запрос. Одно из этих значений хранится в свойстве `resultType` объекта `XPathResult` — это свойство указывает на тип результата, который содержит объект. Эти же константы используются при вызове методов `Document.evaluate()` и `XPathExpression.evaluate()`, чтобы указать желаемый тип результата. Перечень констант и их описание приводятся далее:

`ANY_TYPE`

Передав это значение методу `Document.evaluate()` или `XPathExpression.evaluate()`, можно указать, что допустимым будет считаться результат любого типа. Свойство `resultType` никогда не принимает это значение.

`NUMBER_TYPE`

Результат хранится в свойстве `numberValue`.

`STRING_TYPE`

Результат хранится в свойстве `stringValue`.

`BOOLEAN_TYPE`

Результат хранится в свойстве `booleanValue`.

`UNORDERED_NODE_ITERATOR_TYPE`

Результат представляет собой неупорядоченное множество узлов, последовательный доступ к которым осуществляется в цикле обращением к методу `iterate-`

Next(), пока он не вернет значение null. В процессе исполнения этого цикла документ не должен подвергаться модификациям.

ORDERED_NODE_ITERATOR_TYPE

Результат представляет собой список узлов, следующих в том порядке, в котором они встречаются в документе. Последовательный доступ к узлам в списке осуществляется в цикле обращением к методу iterateNext(), пока он не вернет значение null. В процессе исполнения этого цикла документ не должен подвергаться модификациям.

UNORDERED_NODE_SNAPSHOT_TYPE

Результат представляет собой список узлов с произвольным доступом. Свойство snapshotLength определяет длину списка, а метод snapshotItem() возвращает узел с заданным индексом. Порядок следования узлов в списке может отличаться от порядка следования узлов в документе. Поскольку результат этого типа представляет собой «снимок экрана» (snapshot), он остается действительным, даже если документ подвергается изменениям.

ORDERED_NODE_SNAPSHOT_TYPE

Результат представляет собой список узлов с произвольным доступом, так же как и в случае UNDERED_NODE_SNAPSHOT_TYPE, за исключением того, что порядок следования узлов в списке совпадает с порядком следования узлов в документе.

ANY_UNODERED_NODE_TYPE

Свойство singleNodeValue ссылается на узел документа, соответствующий запросу, или null, если нет ни одного узла, соответствующего запросу. Если в документе запросу соответствует более одного узла, свойство singleNodeValue может ссылаться на любой из них.

FIRST_ODERED_NODE_TYPE

Свойство singleNodeValue ссылается на первый соответствующий запросу узел в документе или null, если нет ни одного такого узла.

Свойства экземпляров

Большинство из представленных здесь свойств хранят корректное значение только при определенных значениях свойства resultType. Попытка обращения к свойствам, которые не определены для текущего значения resultType, приводит к возбуждению исключения.

readonly boolean booleanValue

Хранит результат, когда свойство resultType равно значению BOOLEAN_TYPE.

readonly boolean invalidIteratorState

Значение true, если свойство resultType равно одной из констант ITERATOR_TYPE и документ претерпел изменения, делающие итератор некорректным, т. к. результат уже был возвращен.

readonly float numberValue

Хранит результат, когда свойство resultType равно значению NUMBER_TYPE.

readonly short resultType

Указывает на тип результата исполнения XPath-запроса. Значением этого свойства всегда будет одна из перечисленных ранее констант. Значение этого свойства определяет перечень других доступных для использования свойств и методов.

`readonly Node singleNodeValue`

Хранит результат, когда свойство `resultType` имеет значение `XPathResult.ANY_UNORDERED_NODE_TYPE` или `XPathResult.FIRST_UNORDERED_NODE_TYPE`.

`snapshotLength`

Определяет число узлов, возвращаемых в результате, когда свойство `resultType` имеет значение `UNORDERED_NODE_SNAPSHOT_TYPE` или `ORDERED_NODE_SNAPSHOT_TYPE`. Это свойство используется в паре с методом `snapshotItem()`.

`stringValue`

Хранит результат, когда свойство `resultType` равно значению `STRING_TYPE`.

Методы

`iterateNext()`

Возвращает следующий узел списка. Этот метод может использоваться, когда свойство `resultType` имеет значение `UNORDERED_NODE_ITERATOR_TYPE` или `ORDERED_NODE_ITERATOR_TYPE`.

`snapshotItem()`

Возвращает узел с заданным индексом из списка узлов результата. Этот метод может использоваться, только когда свойство `resultType` имеет значение `UNORDERED_NODE_SNAPSHOT_TYPE` или `ORDERED_NODE_ SNAPSHOT_TYPE`.

Описание

Объект `XPathResult` является представлением значения XPath-выражения. Объекты этого типа возвращаются методами `Document.evaluate()` и `XPathExpression.evaluate()`. Результатами XPath-выражений могут быть строки, числа, логические значения, узлы и списки узлов. Реализации XPath могут возвращать списки узлов разными способами, по этой причине данный объект определяет довольно сложный программный интерфейс для получения фактических результатов исполнения XPath-запроса.

Прежде чем приступить к работе с объектом `XPathResult`, сначала необходимо проверить свойство `resultType`. Значение этого свойства указывает, какие свойства или методы могут использоваться для извлечения значения результата. Попытки обращения к методам или свойствам, которые не определены для текущего значения свойства `resultType`, приводят к возбуждению исключения.

Internet Explorer не поддерживает интерфейс `XPathResult`. Для исполнения XPath-запросов в IE используются методы `Node.selectNodes()` и `Node.selectSingleNode()`.

См. также `Document.evaluate()`, `XPathExpression.evaluate()`

XPathResult.iterateNext()

возвращает следующий узел, соответствующий XPath-запросу

Синтаксис

```
Node iterateNext()
    throws DOMException
```

Возвращаемое значение

Возвращает следующий узел списка узлов, соответствующих запросу, или `null`, если таковых больше не осталось.

Исключения

Этот метод возбуждает исключение, если документ подвергался изменениям уже после того, как был возвращен объект XPathResult. Кроме того, исключение возбуждается, когда свойство resultType содержит значение, отличное от UNORDERED_NODE_ITERATOR_TYPE или ORDERED_NODE_ITERATOR_TYPE.

Описание

Метод iterateNext() возвращает следующий узел, соответствующий запросу, или null, если все подобные узлы уже возвращены. Этот метод может использоваться, только если свойство resultType имеет значение UNORDERED_NODE_ITERATOR_TYPE или ORDERED_NODE_ITERATOR_TYPE. Если тип результата предполагает упорядоченный список узлов (ORDERED), узлы возвращаются в порядке их следования в документе. В противном случае они могут возвращаться в произвольном порядке.

Если свойство invalidIteratorState имеет значение true, значит, документ подвергался изменениям, и в этом случае обращение к данному методу возбудит исключение.

XPathResult.snapshotItem()

возвращает узел, соответствующий XPath-запросу

Синтаксис

```
Node snapshotItem(индекс)
```

Аргументы

индекс Индекс возвращаемого узла.

Возвращаемое значение

Узел с указанным индексом или null, если *индекс* меньше нуля или больше snapshotLength.

Исключения

Этот метод возбуждает исключение, если свойство resultType содержит значение, отличное от UNORDERED_NODE_SNAPSHOT_TYPE или ORDERED_NODE_SNAPSHOT_TYPE.

XSLTProcessor

Firefox 1.0, Safari 2.01, Opera 9

преобразует XML-документ в соответствии с таблицей стилей XSLT

Object→XSLTProcessor

Конструктор

```
new XSLTProcessor()
```

Методы

clearParameters()

Удаляет любой произвольный набор параметров.

getParameter()

Возвращает значение указанного параметра.

importStyleSheet()

Определяет используемую таблицу стилей XSLT.

`removeParameter()`

Удаляет указанный параметр.

`reset()`

Выполняет сброс объекта `XSLTProcessor` в исходное состояние, очищает все параметры и таблицы стилей.

`setParameter()`

Присваивает заданное значение указанному параметру.

`transformToDocument()`

Выполняет преобразование заданного документа или узла с использованием таблицы стилей, передававшейся методу `importStylesheet()`, и параметров, установленных методом `setParameter()`. В качестве результата возвращается новый объект `Document`.

`transformToFragment()`

Выполняет преобразование заданного документа или узла, а в качестве результата возвращается объект `DocumentFragment`.

Описание

Метод `XSLTProcessor` выполняет преобразование узлов XML-документа с использованием таблиц стилей XSLT. Прежде чем задействовать объект `XSLTProcessor`, его необходимо создать вызовом конструктора без аргументов и затем обращением к методу `importStylesheet()` инициализировать с помощью таблицы стилей. Если таблица стилей имеет параметры, установить их можно методом `setParameter()`. Наконец, фактическое преобазование выполняется вызовом метода `transformToDocument()` или `transformToFragment()`.

Internet Explorer поддерживает XSLT-стили, но не реализует объект `XSLTProcessor`. Методы `Node.transformNode()` и `Node.transformNodeToObject()`, реализующие аналогичную функциональность в IE, описываются в соответствующих справочных статьях. Примеры вспомогательных, кросс-платформенных функций для работы с XSLT приводятся в главе 21.

См. также `Node.transformNode()`, `Node.transformNodeToObject()`; глава 21

XSLTProcessor.clearParameters()

удаляет все значения параметров таблицы стилей

Синтаксис

```
void clearParameters()
```

Описание

Данный метод удаляет все значения параметров, которые были определены с помощью метода `setParameter()`. Если преобразования выполняются без набора параметров, вместо них используются значения из таблицы стилей, предлагаемые по умолчанию.

XSLTProcessor.getParameter()

возвращает значение указанного параметра

Синтаксис

```
String getParameter(String URIпространстваИмен, String локальноеИмя)
```

Аргументы

URIпрстранстваИмен

Уникальный идентификатор пространства имен параметра.

локальноеИмя

Имя параметра.

Возвращаемое значение

Значение параметра или `null`, если параметр с таким именем не был установлен.

XSLTProcessor.importStylesheet()

определяет таблицу стилей XSLT для выполнения преобразований

Синтаксис

```
void importStylesheet(Node таблицаСтилей)
```

Аргументы

таблицаСтилей

Таблица стилей XSLT, используемая для выполнения преобразований. Это может быть собственный объект `Document`, а также элемент `<xsl:stylesheet>` или `<xsl:transform>`.

Описание

Метод `importStylesheet()` определяет таблицу стилей XSLT, которая будет использоваться в последующих вызовах методов `transformToDocument()` и `transformToFragment()`.

XSLTProcessor.removeParameter()

удаляет значение параметра

Синтаксис

```
String removeParameter(String URIпространстваИмен, String локальноеИмя)
```

Аргументы

URIпространстваИмен

Уникальный идентификатор пространства имен параметра.

локальноеИмя

Имя параметра.

Описание

Метод `removeParameter()` удаляет значение указанного параметра, если предварительно он был установлен методом `setParameter()`. Во всех последующих преобразованиях будет использоваться значение параметра из таблицы стилей, предлагаемое по умолчанию.

XSLTProcessor.reset()

переустанавливает объект XSLTProcessor в состояние, предлагаемое по умолчанию

Синтаксис

```
void reset()
```

Описание

Данный метод восстанавливает объект XSLTProcessor в том состоянии, в котором он находился сразу же после создания. После вызова этого метода с объектом XSLTProcessor больше не будут связаны какие-либо таблицы стилей и значения параметров.

XSLTProcessor.setParameter()

устанавливает значение параметра таблицы стилей

Синтаксис

```
void setParameter(String URIпространстваИмен,
                  String локальноеИмя,
                  String значение)
```

Аргументы

URIпрстранстваИмен

Уникальный идентификатор пространства имен параметра.

локальноеИмя

Имя параметра.

значение

Значение параметра.

Описание

Данный метод устанавливает значение указанного параметра таблицы стилей.

XSLTProcessor.transformToDocument()

преобразует узел или документ в новый документ

Синтаксис

```
Document transformToDocument(Node исходныйУзел)
```

Аргументы

исходныйУзел

Преобразуемый документ или узел.

Возвращаемое значение

Объект Document, отражающий результаты преобразований.

Описание

Этот метод выполняет XSLT-преобразование указанного узла и в качестве результата возвращает новый объект Document. В процессе преобразований используются таблица XSLT-стилей, заданная методом importStylesheet(), и значения параметров, указанные методом setParameter().

XSLTProcessor.transformToFragment()

преобразует узел или документ в объект DocumentFragment

Синтаксис

```
Document transformToFragment(Node исходныйУзел,
                             Document родительскийДокумент)
```

Аргументы

исходныйУзел

Преобразуемый документ или узел.

родительскийДокумент

Документ, с помощью которого создается возвращаемый объект `DocumentFragment`. Свойство `ownerDocument` возвращаемого объекта `DocumentFragment` будет ссылаться на этот документ.

Возвращаемое значение

Объект `DocumentFragment`, отражающий результаты преобразований.

Описание

Этот метод выполняет XSLT-преобразование указанного узла и в качестве результата возвращает объект `DocumentFragment`. В процессе преобразований используются таблица XSLT-стилей, заданная методом `importStylesheet()`, и значения параметров, указанные методом `setParameter()`. Возвращаемый фрагмент документа может быть вставлен в документ, указанный в аргументе *родительскийДокумент*.

Алфавитный указатель

Специальные символы

Числа

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X

Z

A

Б

В

Г

Д

К

Л

M

H

O

П

P

C

Т

У

Ф

Х

Ц

Ч

Ш

Э

Я