

John Resig

**Pro JavaScript™
Techniques**

Джон Рейсиг

JavaScript
**Профессиональные приёмы
программирования**

2008

Оглавление

Об авторе	7
О техническом редакторе	7
Слова благодарности	7
Глава 1. Современное программирование на JavaScript	9
Объектно-ориентированный JavaScript.....	9
Тестирование кода.....	10
Создание пакета распространения.....	11
Ненавязчивое создание DOM-сценариев.....	12
Объектная модель документа (DOM).....	13
События.....	14
JavaScript и CSS.....	15
Ajax.....	15
Поддержка со стороны браузеров.....	18
Выводы.....	21
Глава 2. Объектно-ориентированный JavaScript	22
Свойства языка.....	22
Ссылки.....	22
Перегрузка функций и проверка типов.....	24
Область видимости переменных.....	28
Замкнутые выражения.....	29
Контекст.....	32
Объектно-ориентированные основы JavaScript.....	34
Объекты.....	34
Создание объектов.....	35
Публичные методы.....	36
Частные методы.....	37
Привилегированные методы.....	38
Статические методы.....	40
Выводы.....	40
Глава 3. Создание кода широкого применения	41
Стандартизация объектно-ориентированного кода.....	41
Наследование с использованием прототипов.....	41
Классическое наследование.....	42
Библиотека Base.....	45
Библиотека Prototype.....	47
Создание пакета.....	50
Организация пространства имен.....	51
Dojo.....	51
YUI.....	53
Очистка программного кода.....	53
Объявление переменных.....	54
Операторы != и == против операторов !== и ===.....	54
Блоки и скобки.....	55
Точки с запятой.....	55
Сжатие.....	56
JSMIn.....	56
Packer.....	57
Распространение.....	58
Выводы.....	59
Глава 4. Инструменты для отладки и тестирования	61
Отладка.....	61
Консоль ошибок.....	61
Internet Explorer.....	61
Firefox.....	62
Safari.....	63
Opera.....	64
Инспекторы DOM.....	65
Инспектор DOM, имеющийся в Firefox.....	65
DOM-инспектор, имеющийся в Safari.....	66
Наглядное представление источника.....	67
Firebug.....	68
Venkman.....	69
Тестирование.....	70
JUnit.....	70
J3Unit.....	72
Test.Simple.....	73
Вывод.....	74
Глава 5. Объектная модель документа	76
Введение в объектную модель документа.....	76
Переходы по DOM.....	76
Как справиться в DOM с пустым пространством.....	79

Простое перемещение по DOM-структуре	81
Привязка к каждому HTML-элементу	82
Стандартные методы DOM	83
Ожидание загрузки HTML DOM	85
Ожидание загрузки страницы	85
Ожидание загрузки основной части DOM	86
Вычисление окончания загрузки DOM	86
Обнаружение элементов в документе HTML	89
Обнаружение элементов по имени класса	89
Обнаружение элементов по селектору CSS	90
cssQuery	91
jQuery	91
XPath	92
Получение содержимого элемента	93
Получение текста, находящегося внутри элемента	94
Получение HTML, находящегося внутри элемента	95
Работа с атрибутами элементов	96
Получение и установка значений атрибута	97
Модификация DOM	100
Создание узлов с использованием DOM	100
Вставка в DOM	101
Вставка в DOM кода HTML	103
Удаление узлов из DOM	105
Вывод	107
Глава 6. События	108
Введение в события JavaScript	108
Асинхронные события против потоков	108
Потоки JavaScript	108
Асинхронные обратные вызовы	109
Фазы события	110
Общие свойства событий	113
Объект события	113
Ключевое слово this	113
Прекращение всплытия событий	114
Подмена исходных действий браузера	116
Привязка перехватчиков событий	118
Традиционная привязка	119
Преимущества традиционной привязки	119
Недостатки традиционного способа привязки	120
DOM-привязка: W3C	120
Преимущества W3C-привязки	121
Недостаток W3C-привязки	121
DOM-привязка: IE	121
Преимущество IE-привязки	122
Недостатки IE-привязки	122
addEventListener и removeEvent	122
Преимущества addEvent	124
Недостаток addEvent	125
Виды событий	125
Создание ненавязчивых DOM-сценариев	126
Предупреждение отключения JavaScript	126
Обеспечение независимости ссылок от JavaScript	127
Отслеживание блокировки CSS	127
Доступность события	128
Вывод	129
Глава 7. JavaScript и CSS	130
Доступ к информации о стилях	130
Динамические элементы	132
Позиция элемента	132
Получение позиции	137
Установка позиции	139
Размер элемента	140
Видимость элемента	143
Анимация	145
Выплывание	145
Проявление	146
Браузер	147
Позиция указателя мыши	147
Область просмотра	148
Размер страницы	148
Позиции полос прокрутки	149
Перемещение полос прокрутки	149
Размер области просмотра	150
Перетаскивание	151
Библиотеки	156
moo.fx и jQuery	157
Scriptaculous	158

Перестроение путем перетаскивания	158
Ползунок для ввода данных	159
Вывод	160
Глава 8. Усовершенствование форм	161
Проверка данных формы	161
Обязательные поля	164
Соответствие шаблону	166
Адреса электронной почты	166
URL	166
Телефонные номера	167
Дата	168
Набор правил	168
Отображение сообщений об ошибках	170
Проверка приемлемости данных	170
Когда следует проводить проверку	174
Проверка, предшествующая отправке данных формы	174
Проверка после внесения в поле изменений	175
Проверка после загрузки страницы	176
Повышение качества работы с формами	176
Накладные надписи	176
Пометка обязательных полей	179
Выводы	180
Глава 9. Создание галерей изображений	182
Примеры галерей	182
Lightbox	182
ThickBox	184
Создание галереи	186
Ненавязчивая загрузка	188
Наложение затемнения	190
Позиционируемый контейнер	192
Переходы	196
Демонстрация изображений	198
Вывод	201
Глава 10. Введение в Ajax	201
Использование Ajax	202
HTTP-запросы	202
Установка соединения	203
Преобразование данных в последовательную форму	204
Создание GET-запроса	206
Создание POST-запроса	206
HTTP ответ	208
Обработка ошибок	209
Проверка истечения времени запроса	210
Обработка ответных данных	211
Полноценный Ajax-пакет	212
Примеры различного использования данных	215
RSS-поток, основанный на формате XML	215
Вставка HTML	217
JSON и JavaScript: Удаленное выполнение	218
Вывод	219
Глава 11. Усовершенствование блогов с помощью Ajax	220
Бесконечный блог	220
Шаблон блога	220
Источник данных	223
Определение наступления событий	224
Запрос	225
Результат	226
Наблюдение за ведением блога в режиме реального времени	229
Вывод	232
Глава 12. Поиск автозаполнения	233
Примеры поиска автозаполнения	233
Построение страницы	235
Отслеживание ввода с клавиатуры	236
Извлечение результатов	240
Переход по списку результатов	243
Перемещения с помощью клавиатуры	243
Перемещение с помощью мыши	244
Окончательный результат	244
Вывод	250
Глава 13. Ajax Wiki	251
Что такое Wiki?	251
Обращение к базе данных	252
Ajax-запрос	253

Код на стороне сервера	254
Обработка запроса	254
Выполнение и форматирование SQL	256
Обработка JSON-ответа	258
Дополнительный учебный пример: JavaScript блог	259
Код приложения	260
Основной код JavaScript	261
JavaScript SQL-библиотека	264
Ruby-код на стороне сервера	265
Вывод	268
Глава 14. В каком направлении движется JavaScript?	269
JavaScript 1.6 и 1.7	269
JavaScript 1.6	269
ECMAScript для XML (E4X)	269
Дополнительные возможности по работе с массивами	271
JavaScript 1.7	272
Включения в массив	272
Управление областью видимости переменных (Let Scoping)	273
Деструктуризация	274
Web Applications 1.0	275
Создание часов	276
Простая модель планет	280
Comet	282
Вывод	285
Приложение А. Справочник по DOM	287
Resources	287
Терминология	287
Предок	288
Атрибут	288
Дочерний элемент	288
Элемент Document	288
Потомки	288
Элемент	288
Узел	288
Родитель	288
Сестры	289
Текстовые узлы	289
Глобальные переменные	289
document	289
HTMLElement	289
Перемещение по DOM	290
body	290
childNodes	290
documentElement	290
firstChild	291
getElementById(элемID)	291
getElementsByName(имяТега)	291
lastChild	292
nextSibling	292
parentNode	293
previousSibling	293
Информация об узле	293
innerText	293
nodeName	294
nodeType	294
nodeValue	295
Атрибуты	296
className	296
getAttribute(имяАтрибута)	296
removeAttribute(имяАтрибута)	297
setAttribute(attrName, attrValue)	297
Модификация DOM	298
appendChild(добавляемыйУзел)	298
cloneNode(true false)	298
createElement(имяТега)	299
createElementNS(пространство_имен, имяТега)	299
createTextNode(текстоваяСтрока)	300
innerHTML	300
insertBefore(узелДляВставки, узелПередКоторымВставлять)	301
removeChild(удаляемыйУзел)	301
replaceChild(вставляемыйУзел, заменяемыйУзел)	302
Приложение Б. Справочник по событиям	303
Источники информации	303
Терминология	303
Асинхронный	303
Прикрепление / Привязка / Регистрация обратного вызова	303

Всплытие.....	303
Захват.....	303
Исходное действие (или действие по умолчанию).....	304
Событие.....	304
Обработчик события.....	304
Потоковый.....	304
Объект события.....	304
Общие свойства.....	304
type.....	304
target / srcElement.....	305
stopPropagation() / cancelBubble.....	305
preventDefault() / returnValue = false.....	306
Свойства мыши.....	307
clientX / clientY.....	307
pageX / pageY.....	307
layerX / layerY и offsetX / offsetY.....	307
button.....	307
relatedTarget.....	308
Свойства клавиатуры.....	309
ctrlKey.....	309
keyCode.....	309
shiftKey.....	311
События страницы.....	311
load.....	311
beforeunload.....	312
error.....	312
resize.....	313
scroll.....	313
unload.....	313
События пользовательского интерфейса (UI).....	314
focus.....	314
blur.....	314
События мыши.....	314
click.....	314
dblclick.....	315
mousedown.....	315
mouseup.....	315
mousemove.....	315
mouseover.....	316
mouseout.....	316
События клавиатуры.....	317
keydown / keypress.....	317
keyup.....	318
События форма.....	318
select.....	318
change.....	318
submit.....	319
reset.....	319
Приложение В. Браузеры.....	321
Современные браузеры.....	321
Internet Explorer.....	321
Версии 5.5 и 6.0.....	321
Версия 7.....	321
Mozilla.....	321
Firefox 1.0, Netscape 8 и Mozilla 1.7.....	322
Firefox 1.5 и 2.0.....	322
Safari.....	322
Opera.....	322
Версия 8.5.....	322
Версия 9.0.....	322

Об авторе



Джон Ресиг (John Resig) — программист и предприниматель, испытывающий пристрастие к языку программирования JavaScript. Он является создателем и ведущим разработчиком JavaScript-библиотеки jQuery, а также ведущим разработчиком многих других веб-проектов. В свободное от программирования время он любит смотреть кино, вести записи в своем блог-журнале (<http://ejohn.org/>), и общаться со своей подружкой Джулией.

О техническом редакторе

Дэн Уэбб (Dan Webb) — свободный разработчик веб-приложений, в последнее время сотрудничающий с Vivabit, где он разрабатывает Event Wax, веб-систему управления событиями. Также недавно он стал соавтором внешнего модуля для Rails — Unobtrusive JavaScript, и расширения Low Pro для Prototype.

Дэн является общепризнанным специалистом по JavaScript, выступавшим на @media 2006, RailsConf и The Ajax Experience. Он написал ряд статей для «A List Apart», «HTML Dog» и «SitePoint», и является членом группы UK web design group the Brit Pack. Он регулярно публикует на своем веб-сайте <http://www.danwebb.net/> заметки о Ruby, Rails и JavaScript. Недавно он стал членом воссозданной команды разработчиков Prototype Core Team.

Слова благодарности

Хочу воспользоваться возможностью поблагодарить всех, кто содействовал выходу этой книги. Нами проделан огромный объем работы, и я благодарен всем за помощь и подсказку, полученные мной на всем протяжении работы над книгой.

Спасибо моему редактору, Крису Миллзу (Chris Mills), за то, что он нашел и вдохновил меня на написание этой книги. Именно он сформировал представление о большей части структуры, развития и канвы повествования этой книги, без него этот проект просто бы не состоялся.

Также хочу поблагодарить моего технического редактора, Дена Уэбба (Dan Webb), за тщательную проверку моего кода и напоминания о более тонких особенностях языка JavaScript. Благодаря его усилиям программный код этой книги работает вполне предсказуемо и представлен в правильном и понятном виде.

Я благодарен литературному редактору, Дженнифер Уайппл (Jennifer Whipple), и выпускающему редактору, Лауре Эстерман (Laura Esterman), которые помогли сделать книгу более легкой для чтения и понимания, и устранили многие допущенные мной недосказанности и непоследовательности в изложении материала.

Я также благодарен моему руководителю проекта, Трейси Браун Коллинзу (Tracy Brown Collins), за то что он был направляющей и организующей силой всей моей работы, и в целом заставлял меня работать с полной отдачей.

Хочу поблагодарить Джулию Вест (Julia West) и Джоша Кинга (Josh King) за поддержку в течение долгих дней и недель работы над книгой, когда я уклонялся от всех других своих обязанностей. Джулия ежедневно была рядом со мной, постоянно убеждаясь в том, что я выполнил свою норму, придавая мне силы и вдохновляя на упорный труд.

И в завершение я хочу поблагодарить за многолетнюю поддержку и содействие свою семью и друзей.

Глава 1 Современное программирование на JavaScript

Развитие JavaScript шло постепенно, но имело постоянный характер. За прошедшее десятилетие восприятие JavaScript изменилось от простого, игрушечного до вполне уважаемого языка программирования, используемого по всему миру корпорациями и разработчиками для создания великолепных приложений. Современный JavaScript такой же, каким был всегда — цельный, надежный и невероятно мощный язык программирования. Многие из того, что рассматривается в этой книге, станет демонстрацией всего, что отличает современные JavaScript-приложения от их предшественников. Многие из представленных в этой главе идей при всем своем развитии уже не новы, но то признание, которое они завоевали у тысяч способных программистов, помогло усовершенствовать методы их применения и сделать их более современными. Итак, давайте без лишних слов приступим к рассмотрению современного программирования на JavaScript.

Объектно-ориентированный JavaScript

С точки зрения языка, вы не найдете здесь абсолютно ничего нового ни об объектно-ориентированном программировании, ни об объектно-ориентированном JavaScript, поскольку этот язык с самого начала создавался как полностью объектно-ориентированный. Тем не менее, по мере того, как JavaScript «развивается» в своем применении и признании, программисты, работающие на других языках (таких как Ruby, Python и Perl) его заметили и стали переносить на него свои приемы программирования.

Объектно-ориентированный код JavaScript по внешнему виду и поведению отличается от кода, написанного на других языках, поддерживающих объектное программирование. Углубленное рассмотрение этого вопроса и различных аспектов, составляющих уникальность языка, я планирую во второй главе, а теперь, чтобы получить представление о том, как пишется современный код JavaScript, давайте обратимся к некоторым основам. В листинге 1.1 приведены примеры двух объектных конструкторов, показывающих образование пары объектов, пригодной для использования в учебном процессе.

Листинг 1.1. Объектно-ориентированный JavaScript, представляющий лекции и расписание их проведения

```
// Конструктор для нашей лекции — 'Lecture'
// принимает две строки — name и teacher
function Lecture( name, teacher ) {
    // Сохранение строк в качестве локальных свойств объекта
    this.name = name;
    this.teacher = teacher;
}

// Метод класса Lecture, используемый для генерации
// строки, которую можно использовать для отображения информации о лекции
Lecture.prototype.display = function(){
    return this.teacher + " преподает " + this.name;
};

// Конструктор расписания лекций, принимающий
// массив лекций
function Schedule( lectures ) {
    this.lectures = lectures;
}

// Метод, предназначенный для построения строки, представляющей
```

```
// расписание лекций
Schedule.prototype.display = function(){
    var str = "";

    // Перебор всех лекций, построение
    // информационной строки
    for ( var i = 0; i < this.lectures.length; i++ )
        str += this.lectures[i].display() + " ";

    return str;
};
```

При просмотре Листинга 1.1 можно заметить, что в его коде присутствует большинство основных элементов объектно-ориентированного программирования, которые по сравнению с элементами других, более распространенных языков объектно-ориентированного программирования структурированы несколько по-другому. Вы можете создавать конструкторы объектов, методы, обращаться к свойствам объектов и извлекать их значения. Пример использования в приложении двух классов показан в Листинге 1.2.

Листинг 1.2. Предоставление пользователю списка классов

```
// Создание нового объекта расписания – Schedule и его сохранение в
// переменной 'mySchedule'
var mySchedule = new Schedule([
    // Создание массива объектов Lecture, который передается
    // объекту Lecture как единое свойство
    new Lecture( "Gym", "Mr. Smith" ),
    new Lecture( "Math", "Mrs. Jones" ),
    new Lecture( "English", "TBD" )
]);

// Отображение информации о расписании в виде всплывающего уведомления
alert( mySchedule.display() );
```

С завоеванием признания JavaScript в программистской среде, стало более популярным использование качественно спроектированного объектно-ориентированного кода. Я попытаюсь наполнить всю книгу различными показательными фрагментами объектно-ориентированного кода JavaScript, которые, как я считаю, лучше всего проиллюстрируют разработку и реализацию кода.

Тестирование кода

После определения качественной объектно-ориентированной базы программного кода, вторым аспектом разработки профессионального кода JavaScript является обеспечение надежной среды тестирования. Потребность в качественном тестировании становится особенно очевидной при разработке кода, который будет активно использоваться или поддерживаться другими разработчиками. Проведение тестирования с целью закладки надежной базы для других разработчиков — неотъемлемая составляющая практики разработки поддерживаемого программного кода.

В главе 4 будут рассмотрены различные инструменты, пригодные для создания нужных режимов тестирования и эксплуатации, а также простая отладка сложных приложений. Один из таких инструментов — дополнительный модуль Firebug для Firefox. Этот модуль предоставляет целый ряд полезных инструментов,

среди которых консоль ошибок, регистрация HTTP-запросов, отладка и обследование элементов. На рис. 1.1 показана реальная копия экрана дополнительного модуля Firefox Firebug в режиме отладки кодового фрагмента.



Рис. 1.1. Копия экрана, иллюстрирующая работу дополнительного модуля Firefox Firebug

Важность разработки свободного от ошибок и хорошо тестируемого кода невозможно переоценить. Как только вы приступите к разработке высококачественного объектно-ориентированного кода и подберете для него надлежащий набор программ для тестирования, я уверен, что вы согласитесь с этим утверждением.

Создание пакета распространения

Завершающим аспектом разработки современного, профессионального кода JavaScript является создание пакета программного кода с целью его распространения или реального применения. Поскольку разработчики стали применять на своих страницах все больше кода JavaScript, возросла вероятность возникновения конфликтных ситуаций. Если в каждой из двух JavaScript-библиотек есть переменная по имени `data`, или в обеих из них решено добавить различную обработку одного и того же события, то могут возникнуть фатальные конфликты и трудноопределимые ошибки.

Идеальный вариант создания качественной JavaScript-библиотеки дает разработчикам возможность просто указать на нее, используя `<script>`-тег, и быть уверенным, что она заработает, не требуя никаких изменений. Чтобы обеспечить качество и универсальную совместимость своего кода, разработчики используют ряд технологий и решений.

Наиболее распространенной технологией устранения помех и влияния на код со стороны другого кода JavaScript, является использование пространства имен. Элементарным (но не обязательно лучшим или наиболее полезным) примером его применения является общедоступная библиотека пользовательского интерфейса, разработанная компанией Yahoo. Пример использования этой библиотеки показан в листинге 1.3.

Листинг 1.3. Добавление обработчика события, используя библиотеку Yahoo UI, обладающую развитым пространством имен

```
// Добавление обработчика события mouseover для элемента, имеющего
// в качестве ID значение 'body'
YAHOO.util.Event.addListener('body', 'mouseover', function() {

    // и изменение фонового цвета элемента на красный
    this.style.backgroundColor = 'red';

});
```

Но у такого метода использования пространства имен есть одна проблема, заключающаяся в отсутствии какой-либо внутренней согласованности между библиотеками относительно порядка или структуры их использования. И здесь особую роль приобретают централизованные хранилища кода, такие как JSAN (JavaScript Archive Network). Это хранилище предоставляет согласованный набор правил для структурируемых библиотек, а также способ быстрого и легкого импорта других библиотек, от которых зависит работа вашего кода. Копия экрана основного центра распространения JSAN показана на рис. 1.2. Обсуждение всех хитросплетений разработки свободного от ошибок, пригодного для создания пакетов распространения кода предстоит в главе 3. В дополнение к этому внимательное отношение к другим, часто встречающимся «камням преткновения», среди которых противоречия, возникающие при обработке событий, будет рассмотрено в главе 6.

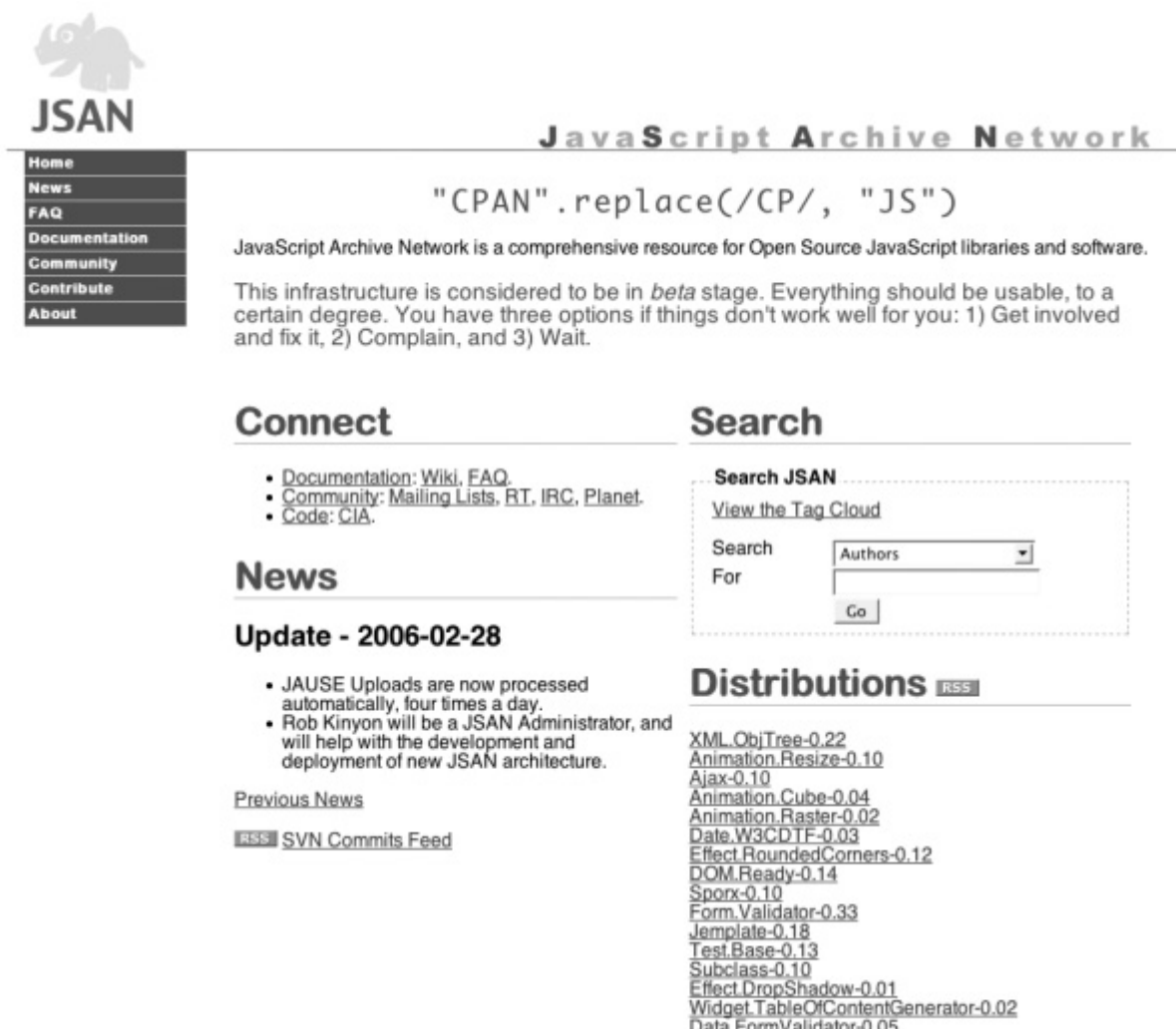


Рис. 1.2. Копия экрана общедоступного хранилища кода JSAN

Ненавязчивое создание DOM-сценариев

Построение продукта на основе качественного, тестируемого кода и совместимого пакета распространения является концепцией ненавязчивого создания DOM-сценария. Написание ненавязчивого кода предполагает полное отделение от вашего HTML-содержимого: от данных, приходящих с сервера, и от кода JavaScript, используемого для придания им динамичности. Наиболее важный сопутствующий эффект от такого полного отделения заключается в том, что теперь вы будете располагать кодом, который всецело поддается упрощению или усложнению в зависимости от используемого браузера. Этим можно воспользоваться, предлагая более современное содержимое на тех браузерах, которые его поддерживают, и, в то же время, элегантно снижая уровень содержимого для браузеров с более скромными возможностями.

Создание современного, ненавязчивого кода имеет две составляющие: объектную модель документа — Document Object Model (DOM), и JavaScript-события. В этой книге предстоит глубокое рассмотрение их обеих.

Объектная модель документа (DOM)

DOM является весьма распространенным способом представления XML-документов. Он не самый быстрый, легкий или простой в использовании, но это и не обязательно, зато его отличают наибольшая распространенность и реализация на большинстве языков, используемых разработчиками веб-приложений (среди которых Java, Perl, PHP, Ruby, Python и JavaScript). DOM был сконструирован с прицелом на интуитивно понятный для разработчиков способ перемещения по XML-иерархии.

Поскольку действующий HTML — это просто подмножество XML, наличие эффективного способа синтаксического разбора и просмотра DOM-документа абсолютно необходимо для облегчения разработки кода на JavaScript. В конце концов в основном взаимодействие в JavaScript происходит между кодом JavaScript и различными HTML-элементами, имеющимися на веб-странице, поэтому DOM является великолепным инструментом для упрощения этого процесса. В листинге 1.4. приводится ряд примеров использования DOM для навигации по различным элементам страницы, для их поиска и последующих манипуляций.

Листинг 1.4. Использование объектной модели документа для определения местоположения различных элементов DOM и манипуляций с ними

```
<html>
<head>
  <title>Введение в DOM</title>
  <script>
    // Мы не можем применять DOM,
    // пока не загружен весь документ
    window.onload = function(){

      // Поиск всех элементов <li>, имеющихся в документе
      var li = document.getElementsByTagName("li");

      // и добавление к ним красного обрамления
      for ( var j = 0; j < li.length; j++ ) {
        li[j].style.border = "1px solid #000";
      }

      // Определение местоположения элемента, имеющего ID
      // со значением 'everywhere'
      var every = document.getElementById( "everywhere" );

      // и удаление его из документа
      every.parentNode.removeChild( every );
    };

  </script>
</head>
<body>
  <h1>Введение в DOM</h1>
  <p class="test">Существует ряд причин по которым DOM можно
    считать превосходной моделью, и вот некоторые из них:</p>
  <ul>
    <li id="everywhere">Ее повсюду можно найти.</li>
```

```

    <li class="test">Она проста в использовании.</li>
<li class="test">Она способна помочь вам найти что угодно,
        и с завидной быстротой.</li>
</ul>
</body>
</html>

```

DOM является первым шагом на пути разработки ненавязчивого кода JavaScript. Быстрота и простота навигации по HTML-документу, приводит к значительному упрощению взаимодействия между JavaScript и HTML.

События

События являются тем самым связующим элементом, который скрепляет все пользовательское взаимодействие с приложением. В хорошо сконструированном приложении JavaScript, вы стремитесь получить доступ к источнику данных и его визуальному представлению (изнутри HTML DOM). Чтобы синхронизировать эти два аспекта, вы стараетесь получить возможность отслеживать пользовательские действия, пытаться в соответствии с ними обновлять пользовательский интерфейс. Сочетание использования DOM и событий JavaScript — это фундаментальный союз, придающий всем современным веб-приложениям присущие им свойства.

Все современные браузеры предоставляют ряд событий, которые заявляют о себе, когда происходят какие-то определенные моменты взаимодействия, в частности, перемещение пользователем указателя мыши, нажатие клавиш или выход из страницы. Используя эти события, вы можете указать код, который будет выполнен при их наступлении. Пример такого взаимодействия показан в листинге 1.5, где фоновый цвет элементов изменяется, когда пользователь проводит над ними указателем мыши.

Листинг 1.5. Использование DOM и событий для обеспечения визуальных эффектов

```

<html>
<head>
    <title>Введение в DOM</title>
    <script>
        // Мы не можем применять DOM,
        // пока не загружен весь документ
        window.onload = function() {

            // Поиск всех элементов <li>, для прикрепления к ним
            // обработчиков событий
            var li = document.getElementsByTagName("li");
            for ( var i = 0; i < li.length; i++ ) {

                // Прикрепление обработчика события наложения указателя мыши
                // на элемент <li>, изменяющего фоновый цвет
                // элемента <li> на синий.
                li[i].onmouseover = function() {
                    this.style.backgroundColor = 'blue';
                };

                // Прикрепление обработчика события выхода указателя мыши
                // за пределы элемента <li>, возвращающего фоновому цвету
                // элемента <li> его исходное белое значение.
            }
        }
    </script>

```

```

        li[i].onmouseout = function() {
            this.style.backgroundColor = 'white';
        };
    }
};
</script>
</head>
<body>
    <h1>Введение в DOM</h1>
    <p class="test">Существует ряд причин по которым DOM можно
        считать превосходной моделью, и вот некоторые из них:</p>
    <ul>
        <li id="everywhere">Ее повсюду можно найти.</li>
        <li class="test">Она проста в использовании.</li>
        <li class="test">Она способна помочь вам найти что угодно,
            и с завидной быстротой.</li>
    </ul>
</body>
</html>

```

События JavaScript сложны и разнообразны. В том или ином виде они используются в большей части кода и приложений, представленных в этой книге. Глава 6 и приложение Б полностью посвящены событиям и их взаимодействию.

JavaScript и CSS

Создание приложений на основе DOM и организация взаимодействия с использованием событий представляют собой динамический HTML. В своей основе динамический HTML является взаимодействием, осуществляемым между JavaScript и CSS-информацией, прикрепленной к элементам DOM.

Каскадные таблицы стилей (Cascading style sheets, CSS) служат стандартом разметки простых, обычных веб-страниц, который до сих пор предоставляет разработчикам наиболее мощный инструмент созидания, доставляя пользователям меньше всего проблем совместимости. В конечном счете, динамический HTML занимается исследованием возможностей организации взаимодействия JavaScript и CSS, и поиском наилучших способов использования их сочетания для получения наиболее впечатляющих результатов.

Некоторые примеры расширенного взаимодействия, в частности, перетаскивание элементов и мультипликацию, можно увидеть в главе 7, где они подробно рассматриваются.

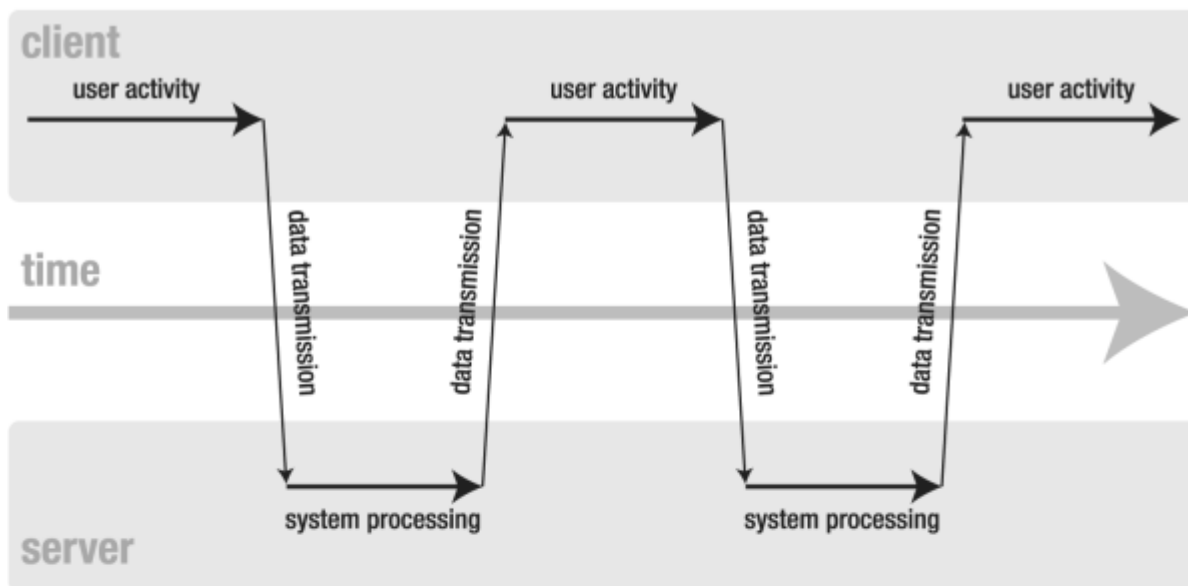
Аjax

Асинхронный JavaScript и XML, Ajax — это термин, придуманный Джесси Джеймсом Гарретом (Jesse James Garrett), соучредителем и президентом Adaptive Path, фирмы по разработке архитектуры информационных систем, в статье «Ajax: A New Approach to Web Applications» («Ajax: новый подход к веб-приложениям») (<http://www.adaptivepath.com/publications/essays/archives/000385.php>). В ней описывается расширенное взаимодействие, происходящее между клиентом и сервером при запросе и передаче дополнительной информации.

Термин Ajax охватывает сотни комбинаций обмена данными, но все они концентрируются вокруг основного исходного условия: дополнительные запросы делаются от клиента к серверу даже после того как страница полностью загружена. Это дает разработчикам приложений дополнительные возможности по организации взаимодействия, избавляющие пользователя от медленного, традиционного хода работы приложения. На рис. 1.3 представлена диаграмма из статьи Гарретта об Ajax, на которой показано, как изменяется поток взаимодействия в

пределах приложения благодаря дополнительным запросам, осуществляемым в фоновом режиме (зачастую, без ведома пользователя).

classic web application model (synchronous)



Ajax web application model (asynchronous)

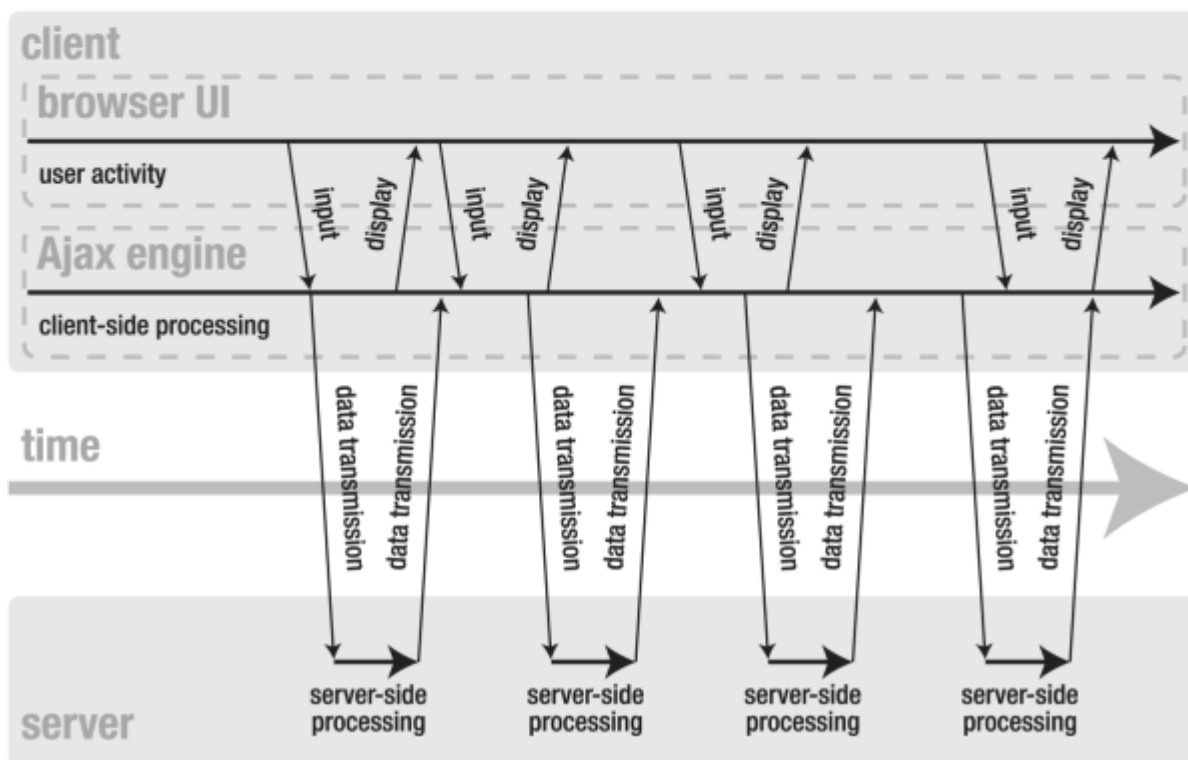


Рис. 1.3. Диаграмма из статьи «Ajax: A New Approach to Web Applications», показывающая расширенное, асинхронное взаимодействие, осуществляемое между клиентом и сервером

Первый выход статьи Гарретта возбудил интерес пользователей, разработчиков, конструкторов и управленцев, обусловив бурный рост новых приложений, использовавших существенно расширившийся уровень взаимодействия. Как ни странно, несмотря на этот всплеск интереса, технологические приемы, положенные в основу Ajax не отличались новизной (и использовались в коммерческих продуктах примерно с 2000 года). Тем не менее, главным отличием было использование в прежних приложениях средств связи с сервером, присущих

конкретному браузеру (например, свойств, имевшихся только в Internet Explorer). С тех пор, как все современные браузеры стали поддерживать XMLHttpRequest (основной метод отправки или получения XML-данных с сервера), игровое поле было выровнено, позволяя всем получать удовольствие от преимуществ применения новой технологии.

Если говорить о компании, занимающей передовые позиции по созданию впечатляющих приложений, в которых используется технология Ajax, то в первую очередь следует упомянуть Google. Еще до выхода первой статьи по Ajax эта компания создала чрезвычайно интерактивную демонстрационную версию — Google Suggest, позволяющую вводить запрос и получать в режиме реального времени возможность его автозавершения. Получить это свойство при использовании старой системы перезагрузки страницы было невозможно. Копия экрана, показывающая Google Suggest в действии, показана на рис. 1.4.



Рис. 1.4. Копия экрана приложения Google Suggest, в котором на момент выхода статьи Гарретта об Ajax уже использовались асинхронные XML-технологии

В то же время у Google было и другое революционное приложение — Google Maps, позволявшее пользователю перемещаться по карте и просматривать нужные ему, локализованные результаты, отображаемые в реальном масштабе времени. Уровень скорости и удобства, предоставляемый этим приложением за счет использования технологий Ajax, резко отличался от уровня других доступных приложений, работающих с картами, в результате чего рынок интерактивных карт был полностью реконструирован. Копия экрана Google Maps показана на рис. 1.5.

Хотя за последние несколько лет язык JavaScript не претерпел каких-либо существенных физических изменений, его принятие в качестве полноценной среды программирования такими компаниями как Google и Yahoo, показало, сколь существенно изменилось его восприятие и популярность.

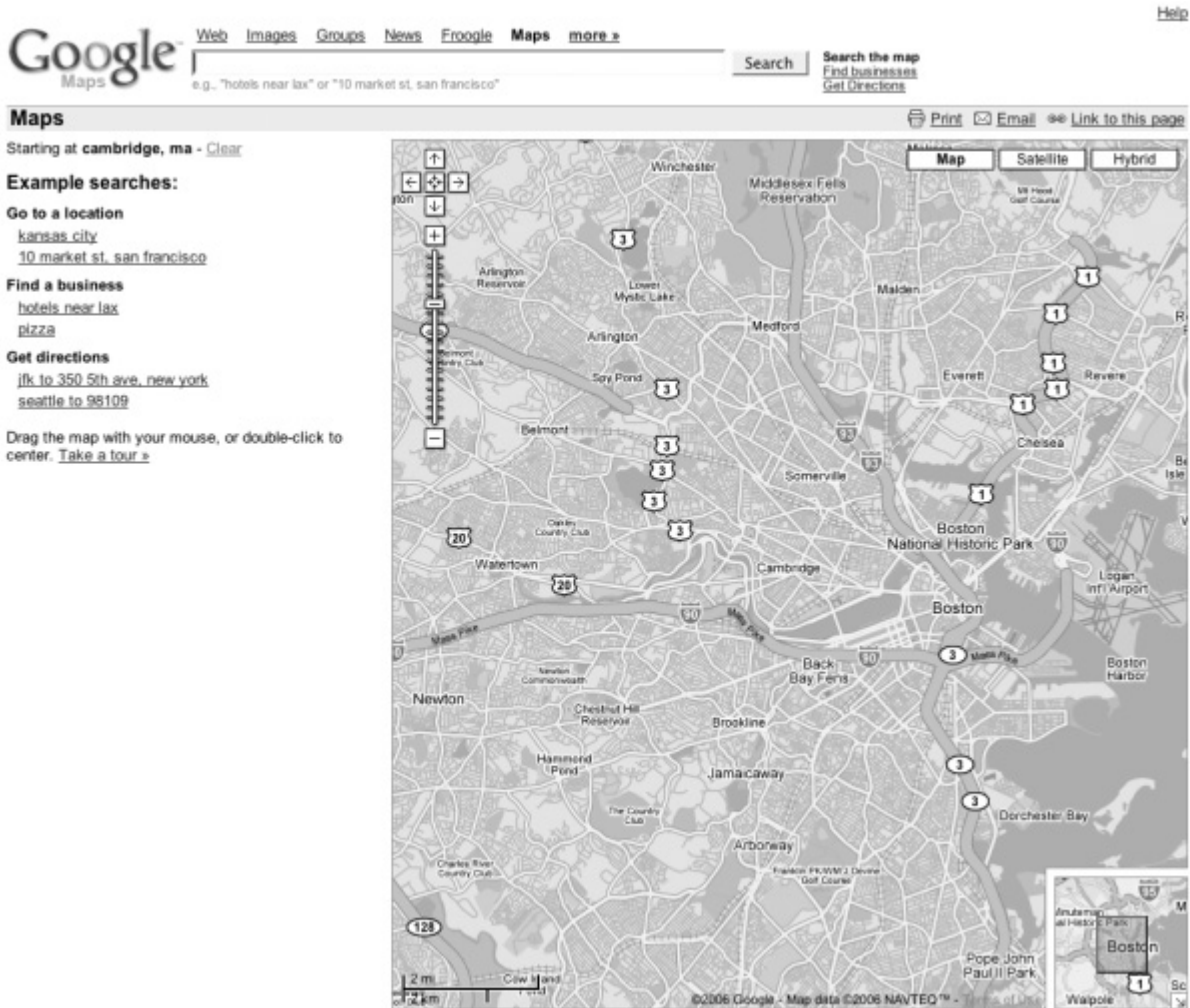


Рис. 1.5. Приложение Google Maps, использующее ряд Ajax-технологий для динамической загрузки локализованной информации

Поддержка со стороны браузеров

Разработка на JavaScript связана с одним грустным обстоятельством — привязанностью к браузерам, осуществляющим ее реализацию и поддержку, получается, что она отдана на откуп тем браузерам, которые на данный момент обладают наибольшей популярностью. Поскольку пользователи не всего используют браузеры, обладающие наилучшей поддержкой JavaScript, мы вынуждены определять, какие свойства считать наиболее важными.

Многие разработчики стали отказываться от поддержки своих программ теми браузерами, которые доставляют слишком много проблем при разработке с учетом их особенностей. Существует тонкое соотношение между поддержкой браузеров на основе контингента их пользователей, и их поддержкой на основе наличия в них требуемых свойств.

Недавно компания Yahoo выпустила JavaScript-библиотеку, которую можно использовать для расширения своих веб-приложений. Вместе с этой библиотекой был выпущен ряд указаний по приемам конструирования, которых должны придерживаться разработчики веб-приложений. На мой взгляд, наиболее важным из выпущенных компанией Yahoo документов является официальный список браузеров, в которых поддерживается и не поддерживается работа библиотеки. Любой человек, и любая корпорация могли бы сделать то же самое, опубликовав документ, который трудно переоценить, на одном из своих самых посещаемых веб-сайтов Интернета.

В Yahoo разработана система категорирования браузерной поддержки, присваивающая браузерам определенную категорию на основе их возможностей. В ней браузерам присваивается одна из трех категорий: А, Х и С:

- Браузеры категории А протестированы на полную поддержку, и приложения Yahoo могут гарантированно работать под их управлением.
- Браузеры категории Х — это те же браузеры категории А, о существовании которых компания Yahoo знает, но еще не имела возможности полностью протестировать их работу, или это новые браузеры, с которыми она еще никогда не имела дело. Браузеры категории Х получают такое же содержимое, что и браузеры категории А, в надежде, что они смогут справиться с этим расширенным содержимым.
- Браузеры категории С известны как «плохие», не поддерживающие свойства, необходимые для полноценной работы приложений Yahoo. Это такие браузеры, которые обслуживают функциональные контенты приложений, в которых не используется JavaScript, поскольку приложения Yahoo абсолютно ненавязчивы в этом вопросе (в том смысле, что они продолжают работать и в отсутствие JavaScript).

Кстати, совершенно случайно тот выбор категорий, который был сделан компанией Yahoo, совпал с моим собственным, что придало ему особую привлекательность. В этой книге я часто употребляю термин *современный браузер*, используя который я подразумеваю любой браузер, который в таблице браузеров Yahoo отнесен к категории А. Наличие постоянного набора свойств, с которыми можно работать, позволяет сделать процесс обучения и разработки намного более интересным и менее тягостным (благодаря игнорированию вопросов несовместимости браузеров).

Я настоятельно рекомендую ознакомиться с документами, относящимися к категорированию браузерной поддержки (которые можно найти по адресу <http://developer.yahoo.com/yui/articles/gbs/gbs.html>), включая и соответствующую таблицу, изображенную на рис. 1.6, чтобы оценить предпринятую Yahoo попытку. Открывая доступ к этой информации широкому кругу веб-разработчиков, Yahoo предоставляет неоценимый «золотой стандарт», к которому следует стремиться и всем остальным, что само по себе уже трудно переоценить.

Дополнительную информацию о поддерживаемых браузерах можно найти в приложении В, где подробно рассмотрены все недостатки и преимущества каждого браузера. Практически всегда те браузеры, которые отнесены к категории А, будут находиться на переднем крае разработки, обеспечивая исчерпывающим набором свойств, необходимых для ее осуществления. Выбор браузеров, от которых вы хотите добиться поддержки своих приложений, в конечном счете обуславливается тем набором свойств, который будет в состоянии поддержать ваше приложение. Если вы (к примеру) хотите добиться поддержки Netscape Navigator 4 или Internet Explorer 5, это сильно сократит то количество свойств, которые можно будет использовать в вашем приложении, в силу того, что эти браузеры ограничены в поддержке современных технологий программирования.

	Win 98	Win 2000	Win XP	Mac 10.0	Mac 10.2	Mac 10.3	Mac 10.3.x	Mac 10.4
IE 7.0	n/a	n/a	A-grade	n/a	n/a	n/a	n/a	n/a
IE 6.0	A-grade	A-grade	A-grade	n/a	n/a	n/a	n/a	n/a
IE 5.5	A-grade	A-grade	n/a	n/a	n/a	n/a	n/a	n/a
IE 5.0	C-grade	C-grade	n/a	C-grade	C-grade	C-grade	C-grade	C-grade
Netscape 8.0	X-grade	X-grade	A-grade	n/a	n/a	n/a	n/a	n/a
Firefox 1.5	A-grade	A-grade	A-grade	A-grade	A-grade	A-grade	A-grade	A-grade
Firefox 1.0.7	A-grade	A-grade	A-grade	A-grade	A-grade	A-grade	A-grade	A-grade
Mozilla 1.7.12	X-grade	X-grade	A-grade	X-grade	X-grade	X-grade	X-grade	X-grade
Opera 8.5	X-grade	X-grade	A-grade	C-grade	C-grade	C-grade	X-grade	X-grade
Safari 1.0	n/a	n/a	n/a	X-grade	n/a	n/a	n/a	n/a
Safari 1.1	n/a	n/a	n/a	X-grade	X-grade	n/a	n/a	n/a
Safari 1.2	n/a	n/a	n/a	X-grade	X-grade	X-grade	n/a	n/a
Safari 1.3	n/a	n/a	n/a	n/a	n/a	X-grade	A-grade	n/a
Safari 2.0	n/a	n/a	n/a	n/a	n/a	n/a	n/a	A-grade

Рис. 1.6. Таблица категорирования браузерной поддержки, предоставленная компанией Yahoo

Зная, какие из браузеров можно отнести к современным, вы можете воспользоваться их сильными сторонами, и получить надежную базу для дальнейших разработок. Эта база может быть определена наличием следующего набора свойств:

Ядра JavaScript 1.5: Самой последней, широко распространенной версии JavaScript, обладающей всеми свойствами, необходимыми для полной функциональной поддержки объектно-ориентированного языка JavaScript. Internet Explorer 5.0 не обладает полной поддержкой версии 1.5, что является главной причиной нежелания разработчиков заниматься поддержкой этого браузера.

XML Document Object Model (DOM) 2: Стандарта рассмотрения HTML и XML-документов. Это абсолютно необходимое условие для создания быстро работающих приложений.

XMLHttpRequest: Основы Ajax — простого уровня инициирования удаленных http-запросов. Этот объект поддерживается всеми браузерами по умолчанию, за исключением Internet Explorer 5.5–6.0; тем не менее оба они поддерживают инициирование совместимого объекта, используя ActiveX.

CSS: Основного компонента, необходимого для разработки веб-страниц. Это требование может показаться странным, но наличие CSS жизненно важное требование для разработчиков веб-приложений. Поскольку таблицы CSS поддерживаются всеми современными браузерами, это обычно сводится к несоответствиям в представлениях, вызывающим наибольшее количество проблем. Это главная причина, по которой Internet Explorer для Mac поддерживается реже всех браузеров.

Сочетание всех этих свойств браузера и составляет основу разработки приложений на JavaScript. Поскольку ранее перечисленные свойства так или иначе поддерживаются всеми современными браузерами, вы получаете твердую платформу для создания всего, что будет встречаться в остальной части этой книги. Все, что рассматривается в книге будет основано на предположении, что используемый вами браузер по крайней мере поддерживает эти свойства.

Выводы

В этой книге делается попытка всецело охватить все современные, профессиональные технологии программирования на JavaScript, поскольку они используются всеми, от индивидуальных разработчиков до больших корпораций, делая их код более удобным, понятным и интерактивным.

В этой главе мы получили короткий обзор всего, о чем собираемся вести речь в этой книге. Сюда входят основы профессионального программирования на JavaScript: создание объектно-ориентированного кода, его тестирование и создание пакета распространения. Затем вы увидели, в чем заключаются фундаментальные аспекты создания ненавязчивых DOM-сценариев, включая краткий обзор объектной модели документа, событий и вопросов взаимодействия между JavaScript и CSS. И, наконец, вы взглянули на исходные условия, лежащие в основе технологии Ajax и на проблему поддержки JavaScript в современных браузерах. Изучение всех этих тем более чем достаточное условие освоения уровня профессионального программирования на JavaScript.

Глава 2 Объектно-ориентированный JavaScript

Объекты являются основными элементами JavaScript. В сущности все в JavaScript является объектом, со всеми вытекающими из этого преимуществами. Но чтобы заработать репутацию надежного объектно-ориентированного языка, JavaScript включает широкий арсенал свойств, делающих его весьма оригинальным языком, как по возможностям, так и по стилю.

В этой главе я хочу начать с охвата некоторых наиболее важных аспектов языка JavaScript, среди которых ссылки, область видимости, замыкания и контекст, которым слишком мало уделено внимания в других книгах по JavaScript. После того, как будет заложен прочный фундамент, мы перейдем к исследованию других важных аспектов объектно-ориентированного JavaScript, куда будут включены тонкости поведения объектов, способы создания новых объектов и установка методов со специфическими полномочиями. Если отнестись к ней со всей серьезностью, то вполне возможно, что эта глава — самая важная в книге, поскольку она должна полностью изменить ваш взгляд на JavaScript, как на язык программирования.

Свойства языка

В языке JavaScript имеется ряд свойств, составляющих основу его самобытности. Подобными свойствами обладают очень немногие другие языки. Лично я считаю, что сочетание свойств и придает языку ту самую скрытую мощь.

Ссылки

Основным аспектом JavaScript является понятие ссылок. *Ссылка* — это указатель на фактическое местонахождение объекта. Это невероятно мощное свойство, в отношении которого действует предположение, что физический объект никогда не может быть ссылкой. Строка — это всегда строка, а массив — всегда массив. Но на один и тот же объект могут ссылаться многие переменные. Именно на такой системе ссылок построен JavaScript. Поддерживая наборы ссылок на другие объекты, язык предоставляет вам гораздо больше гибкости.

Вдобавок к этому объект может содержать набор свойств, целиком состоящий из простых ссылок на другие объекты (такие как строки, числа, массивы и т.д.). Когда несколько переменных указывают на один и тот же объект, модификация исходного типа этого объекта отразится на всех переменных. Соответствующий пример показан в листинге 2.1, где две переменные указывают на один и тот же объект, а модификация содержимого объекта имеет глобальный эффект.

Листинг 2.1. Пример нескольких переменных, ссылающихся на один и тот же объект

```
// Установка obj на пустой объект
var obj = new Object();

// теперь objRef является ссылкой на другой объект
var objRef = obj;

// Модификация свойства исходного объекта
obj.oneProperty = true;

// Теперь мы видим, что это изменение представлено в обеих переменных
// (Поскольку обе они ссылаются на один и тот же объект)
alert( obj.oneProperty === objRef.oneProperty );
```

Прежде я уже упоминал, что самомодифицирующиеся объекты составляют для JavaScript большую редкость. Рассмотрим один довольно известный случай, когда это все-таки происходит. Объект массива способен

самостоятельно дополняться элементами, используя метод `push()`. Поскольку в ядре объекта `Array` значения сохранены как свойства объекта, в результате возникает ситуация, подобная той, что показана в листинге 2.1, где объект становится глобально модифицированным (приводящая к одновременным изменениям в содержимом множества переменных). При мер подобной ситуации приведен в листинге 2.2.

Листинг 2.2. Пример самомодифицирующегося объекта

```
// Создание массива элементов
var items = new Array( "one", "two", "three" );

// Создание ссылки на массив элементов
var itemsRef = items;

// Добавление элемента к исходному массиву
items.push( "four" );

// Длина обоих массивов должна быть одинаковой,
// поскольку оба они указывают на один и то же объект массива
alert( items.length == itemsRef.length );
```

Важно запомнить, что ссылки всего лишь указывают на конечный объект, на который они ссылаются, и не являются ссылками друг на друга. К примеру, в Perl есть возможность иметь ссылку, указывающую на другую переменную, которая также является ссылкой. Но в JavaScript ссылка опускается по цепочке ссылок, и указывает только на основной объект. Эта ситуация проиллюстрирована в листинге 2.3, где физический объект претерпевает изменения, но ссылка продолжает указывать на старый объект.

Листинг 2.3. Изменение ссылки на объект Changing the Reference of an Object While Maintaining Integrity

```
// Установка items на массив строковых значений (объект)
var items = new Array( "one", "two", "three" );

// Установка itemsRef в качестве ссылки на items
var itemsRef = items;

// Установка items на новый объект
items = new Array( "new", "array" );

// Теперь items и itemsRef указывают на разные объекты.
// items указывает на new Array( "new", "array" )
// itemsRef указывает на new Array( "one", "two", "three" )
alert( items !== itemsRef );
```

И, наконец, взглянем на удивительный случай, возникающий при самомодификации объекта, но приводящий к возникновению нового объекта, ссылки на который отсутствуют. При объединении строк всегда образуется новый объект строки, а не модифицированная версия исходной строки. Этот случай показан в листинге 2.4.

Листинг 2.4. Пример модификации объекта, в результате которой возникает не самомодифицированный, а новый объект

```
// Установка item на новый строковый объект
```

```

var item = "test";

// Теперь itemRef ссылается на тот же самый строковый объект
var itemRef = item;

// Объединение строкового объекта в новом текстом
// Учтите, что при этом создается новый объект,
// а не модифицируется исходный.
item += "ing";

// Значения item и itemRef не равны друг другу, поскольку
// был создан абсолютно новый строковый объект
alert( item !== itemRef );

```

Если вы плохо знакомы со ссылками, то в них не сложно и запутаться. Но понимание порядка их работы является основой для создания высококачественного и свободного от ошибок кода JavaScript. В нескольких следующих разделах мы рассмотрим ряд свойств, которые может быть и не отличаются новизной или особой привлекательностью, но играют важную роль в написании безупречного программного кода.

Перегрузка функций и проверка типов

Распространенным свойством других объектно-ориентированных языков, к которым относятся и язык Java, является возможность «перегрузки» функций, изменяющей ее поведение в зависимости от количества или типов переданных им аргументов. Хотя напрямую это свойство в JavaScript не доступно, ряд имеющихся в нем инструментов предоставляет точно такую же возможность.

Для перегрузки функций требуются две вещи: возможность определения количества предоставленных аргументов, и возможность определения типа аргументов. Начнем с количества предоставленных аргументов.

Внутри каждой функции JavaScript существует контекстно-зависимая переменная под названием *arguments*, которая ведет себя как псевдомассив, содержащий все переданные функции аргументы. Переменная *Arguments* не является настоящим массивом (т.е. вы не можете вносить в нее изменения или вызывать метод *.push()* для добавления новых элементов), но можете обращаться к элементам массива и использовать свойство *.length*. Это подтверждается двумя примерами, показанными в листинге 2.5.

Листинг 2.5. Два примера перегрузки функции в JavaScript

```

// Простая функция отправки сообщения
function sendMessage( msg, obj ) {

    // если предоставлен и текст сообщения и объект
    if ( arguments.length == 2 )
        // Отправить сообщение объекту
        obj.handleMsg( msg );

    // В противном случае предположить, что предоставлен только текст
    // сообщения
    else
        // Поэтому ограничиться выводом задаваемого по умолчанию сообщения
        // об ошибке
        alert( msg );
}

```



```

}

// Вызов функции с одним аргументом приводит к отображения сообщения
// с использованием метода alert
sendMessage( "Hello, World!" );

// Также функции можно передать свой собственный объект, который
// осуществляет другой способ отображения информации

sendMessage( "How are you?", {
    handleMessage: function( msg ) {
        alert( "This is a custom message: " + msg );
    }
});

// Функция, принимающая любое количество аргументов и составляющая из них
// массив

function makeArray() {
    // Временный массив
    var arr = [];

    // Перебор всех переданных аргументов
    for ( var i = 0; i < arguments.length; i++ ) {
        arr.push( arguments[i] );
    }

    // Возвращение получившегося массива
    return arr;
}

```

Существует также и другой метод определения количества аргументов, переданных функции. Но он имеет несколько более изощренную природу использования. В нем применяется тот факт, что любой, не переданный аргумент имеет значение неопределенного типа — *undefined*. В листинге 2.6 показана простая функция, предназначенная для отображения сообщения об ошибке, и предоставляющая сообщение по умолчанию, если никакое сообщение ей не передано.

Листинг 2.6. Отображение сообщения об ошибке и сообщения по умолчанию

```

function displayError( msg ) {
    // Проверка на то, что значение msg не определено (undefined)
    if ( typeof msg == 'undefined' ) {
        // Если оно не определено, установить значение по умолчанию
        msg = "Произошла ошибка.";
    }

    // Отображение сообщения
    alert( msg );
}

```

Использование оператора `typeof` помогает нам перейти к теме проверки типов. Поскольку JavaScript на данный момент является языком с динамическим определением типов, то эта тема может оказаться очень полезной и интересной. Существует несколько различных способов проверки типа переменной, но мы собираемся рассмотреть два самых полезных.

Первый способ проверки типа объекта заключается в использовании интуитивно понятного оператора `typeof`. Этот оператор дает нам строковое имя, представляющее тип содержимого переменной. Применение этого способа было бы великолепным решением, за исключением того, что для переменных типа `object` или `array`, или пользовательского объекта, к примеру, `user`, этот оператор возвращает строку `object`, не позволяя понять, в чем состоит разница между всеми этими объектами. Пример применения этого способа показан в листинге 2.7.

Листинг 2.7. Пример использования `typeof` для определения типа объекта

```
// Проверка, не является ли наше число на самом деле строкой
if ( typeof num == "string" )
    // Если является, выделение из нее численной составляющей
    num = parseInt( num );

// Проверка, не является ли наш массив на самом деле строкой
if ( typeof arr == "string" )
    // Если так оно и есть, создание массива, за счет разбиения строки
    // по элементам, разделенным запятыми
    arr = arr.split(",");
```

Второй способ проверки типа объекта заключается в использовании ссылки на свойство, присущее всем объектам JavaScript, которое называется `constructor`. Это свойство является ссылкой на функцию, используемую для первоначального создания этого объекта. Пример использования этого способа показан в листинге 2.8.

Листинг 2.8. Пример использования свойства `constructor` для определения типа объекта

```
// Проверка, не является ли наше число на самом деле строкой
if ( num.constructor == String )
    // Если является, выделение из нее численной составляющей
    num = parseInt( num );

// Проверка, не является ли наша строка на самом деле массивом
if ( str.constructor == Array )
    // Если так оно и есть, создание строки за счет объединения элементов
    // массива через запятые
    str = str.join(',');
```

В таблице 2.1 показаны результаты проверки типа различных объектов с использованием двух, рассмотренных нами методов. В первом столбце таблицы показан объект, тип которого мы пытаемся определить. Во втором столбце показан результат работы оператора `typeof` Переменная (где *Переменная* — это значение, содержащееся в первом столбце). Все результаты, представленные в этом столбце являются строками. И, наконец, в третьем столбце показан результат работы `Переменная.constructor` по отношению к объектам, содержащимся в первом столбце. Все результаты, представленные в этом столбце, являются объектами.

Таблица 2.1. Проверка типов переменных

Переменная	Тип переменной	Конструктор переменной
------------	----------------	------------------------

{ an: "object" }	object	Object
["an", "array"]	object	Array
function(){}	function	Function
"a string"	string	String
55	number	Number
true	boolean	Boolean
new User()	object	User

Теперь, воспользовавшись информацией из таблицы 2.1, вы можете создать универсальную функцию для проведения внутри нее проверки типа. Теперь уже, наверное, очевидно, что использование конструктора переменной в качестве ссылки на тип объекта — это наиболее надежный и безошибочный способ проверки типов. Строгая проверка типов может помочь в тех случаях, когда вам нужно убедиться, что вашим функциям передано абсолютно точное количество аргументов абсолютно правильного типа. Практический пример такой проверки показан в листинге 2.9.

Листинг 2.9. Функция, которую можно использовать для строгого утверждения всех аргументов, переданных функции

```
// Строгая проверка списка типов переменных по списку аргументов
function strict( types, args ) {

    // Проверка совпадения количества типов и аргументов
    if ( types.length !== args.length ) {

        // Если количество не совпадает, выдача полезного исключения
        throw "Неверное число аргументов. Ожидалось " + types.length +
            ", а вместо этого получено " + args.length + ".";
    }

    // Перебор всех аргументов и проверка их типов
    for ( var i = 0; i < args.length; i++ ) {
        //
        if ( args[i].constructor !== types[i] ) {
            throw "Неверный тип аргумента. Ожидался" + types[i].name +
                ", а получен " + args[i].constructor.name + ".";
        }
    }
}

// Простая функция, предназначенная для распечатки списка пользователей
function userList( prefix, num, users ) {
    // Проверка, что prefix — строка, num — число,
    // а users — массив
    strict( [ String, Number, Array ], arguments );

    // Выполнение итерации до 'num' пользователей
    for ( var i = 0; i < num; i++ ) {
```

```

    // Отображение сообщения о каждом пользователе
    print( prefix + ": " + users[i] );
}
}

```

Проверка типа переменных и проверка длины массива аргументов по сути являются весьма простыми понятиями, но могут использоваться для обеспечения сложных методов, которые могут быть к ним приспособлены, и создавать о вашем коде наилучшие впечатления у разработчиков и пользователей. Далее мы собираемся рассмотреть область видимости, существующую в JavaScript, и как ею лучше всего управлять.

Область видимости переменных

В JavaScript область видимости имеет свои особенности. Те или иные формы проявления области видимости присутствуют во всех объектно-ориентированных языках программирования; а какие именно — зависит от того, в каком контексте сохраняется эта область. В JavaScript область видимости сохраняется внутри функций, а не блоков (образуемых такими операторами, как `while`, `if` и `for`). В итоге может получиться код, имеющий, казалось бы, несколько странный результат работы (если вы ранее работали с языками, имеющими поблочные области видимости). В листинге 2.10 показан пример результата работы кода с областью видимости, проявляемой внутри функции.

Листинг 2.10. Пример работы области видимости переменных в JavaScript

```

// Установка значения 'test' для глобальной переменной foo
var foo = "test";

// Внутри блока if
if ( true ) {
    // Присвоение foo значения 'new test'
    // ПРИМЕЧАНИЕ: все это внутри глобальной области видимости!
    var foo = "new test";
}

// Здесь мы видим, что foo теперь равна 'new test'
alert( foo == "new test" );

// Создание функции, изменяющей значение переменной foo
function test() {
    var foo = "old test";
}

// Тем не менее, когда осуществляется ее вызов, воздействие на 'foo'
// проявляется только в области видимости внутри функции
test();

// что подтверждается тем, что значение foo по-прежнему равно 'new test'
alert( foo == "new test" );

```

Обратите внимание на то, что в листинге 2.10 переменные находятся внутри глобальной области видимости. Интересной особенностью встроенных в браузеры версий JavaScript является то, что переменные, имеющие глобальную область видимости, фактически являются лишь свойствами объекта `window`. Хотя в некоторых устаревших версиях Opera и Safari этого не происходит, все-таки лучше придерживаться того, что

браузер именно так себя и ведет. В листинге 2.11 показан пример такого проявления глобальной области видимости.

Листинг 2.11. Пример глобальной области видимости в JavaScript и в объекте Window

```
// Глобальная переменная, в которой содержится строка 'test'
var test = "test";

// Обратите внимание на то, что наша 'глобальная' переменная и свойство test
// объекта window идентичны друг другу
alert( window.test == test );
```

И, наконец, посмотрим, что произойдет когда объявление переменной не определено. В листинге 2.12 значение переменной foo присвоено в области видимости внутри функции test(). Тем не менее, в этом листинге фактически нигде не объявлена область видимости этой переменной (не использовано объявление var foo). Когда переменная foo не объявлена явным образом, она становится глобальной, даже если она используется только в области видимости внутри функции.

Листинг 2.12. Пример скрытого объявления глобальной переменной

```
// Функция, внутри которой устанавливается значение переменной foo
function test() {
    foo = "test";
}

// Вызов функции для установки значения foo
test();

// Мы видим, что теперь foo имеет глобальную область видимости
alert( window.foo == "test" );
```

Хотя в JavaScript механизм области видимости и не такой строгий, как у языков, где он имеет блочный характер, теперь уже не остается сомнений в том, что он все же обладает достаточной мощностью и выразительностью. А когда это еще и сочетается с понятием замкнутых выражений, которое будет рассмотрено в следующем разделе, JavaScript проявляет себя очень мощным языком сценариев.

Замкнутые выражения

Замкнутые выражения — это средства, через которые внутренние функции могут сослаться на переменные, представленные в их внешних функциях, в которые они были включены, после того, как их родительские функции уже завершили свою работу. Именно эта тема может сильно повлиять на ваши представления и может стать слишком сложной для восприятия. Я настоятельно рекомендую обратиться к веб-сайту, упомянутому в конце этого раздела, поскольку на нем выложена превосходная информация по теме замкнутых выражений.

Начнем с рассмотрения двух простых примеров замкнутых выражений, показанных в листинге 2.13.

Листинг 2.13. Два примера, показывающие, как замкнутые выражения могут сделать ваш код более понятным

```
// Поиск элемента, ID которого равен 'main'
var obj = document.getElementById("main");
```

```
// Изменение стиля его обрамления
obj.style.border = "1px solid red";

// Инициализация обратного вызова, который должен произойти через одну
// секунду
setTimeout(function(){
    // и сделать объект невидимым
    obj.style.display = 'none';
}, 1000);

// Типичная функция для отображения задержанного сообщения
function delayedAlert( msg, time ) {
    // Инициализация вложенного обратного вызова
    setTimeout(function(){
        // в котором используется сообщение, переданное из включающей его
        // функции
        alert( msg );
    }, time );
}

// Вызов функции delayedAlert с двумя аргументами
delayedAlert( "Добро пожаловать!", 2000 );
```

Первый вызов функции `setTimeout` демонстрирует распространенный случай, когда у неопытных разработчиков на JavaScript возникают проблемы. Зачастую в их программах можно увидеть примерно следующий код:

```
setTimeout("otherFunction()", 1000);

// или даже ...;
setTimeout("otherFunction(" + num + "," + num2 + ")", 1000);
```

Используя понятие замкнутых выражений, можно вполне обойтись и без этой мешанины программного кода. В первом примере все очень просто. В нем имеется обратный вызов, выполняемый через 1000 миллисекунд после первого вызова, в котором есть ссылка на переменную `obj` (которая определена как глобальная переменная, содержащая ссылку на элемент с ID равным `main`). Вторая, определенная в примере функция, `delayedAlert`, показывает решение по выводу сообщения с помощью `setTimeout`, а также возможность иметь замкнутые выражения в пределах областей видимости функций.

Вы можете убедиться, что использование простых замкнутых выражений, подобных тому, которое применено в нашем коде, делает содержание программы более понятным, не давая ему превращаться в синтаксический винегрет.

Рассмотрим один интересный побочный эффект, который можно получить при использовании замкнутых выражений. В некоторых функциональных языках программирования существует понятие карринга (*currying*). По своей сути, *карринг* — это способ предварительного заполнения функции некоторым количеством аргументов, путем создания новой, более простой функции. В листинге 2.14 приведен простой пример карринга, создающего новую функцию, преднаполняемую аргументом другой функции.

Листинг 2.14. Пример осуществления карринга функции с использованием замкнутых выражений

```
// Функция, генерирующая новую функцию сложения чисел
function addGenerator( num ) {

    // Возврат простой функции, выполняющей сложение двух чисел, где первое
    // число позаимствовано у генератора
    return function( toAdd ) {
        return num + toAdd
    };
}

// Теперь addFive содержит функцию, которая берет один аргумент прибавляет к
// нему число пять, и возвращает полученный результат
var addFive = addGenerator( 5 );

// Здесь мы можем убедиться, что результат работы функции addFive равен 9,
// когда ей в качестве аргумента передается число 4
alert( addFive( 4 ) == 9 );
```

При программировании на JavaScript возникает еще одна, довольно распространенная проблема, которую можно решить с помощью замкнутых выражений. Дело в том, что неопытные разработчики имеют привычку случайно оставлять в глобальной области видимости большое количество абсолютно ненужных переменных. Эта практика всегда считалась порочной, поскольку эти лишние переменные могут создавать скрытую помеху работе каких-нибудь библиотек, вызывая возникновение запутанных проблем. В листинге 2.15 показано, как используя самовыполняемые, безымянные функции, вы можете фактически скрыть от всего остального кода все переменные, которые обычно получают глобальную область видимости.

Листинг 2.15. Пример использования безымянных функций для скрытия переменных от глобального обозрения

```
// Создание новой безымянной функции, используемой в качестве оболочки
(function(){
    // Переменная, обычно имеющая глобальную область видимости
    var msg = "Спасибо за визит!";

    // Привязка к глобальному объекту новой функции,
    window.onload = function(){
        // которая использует 'скрытую' переменную
        alert( msg );
    };

// Закрытие безымянной функции и ее выполнение
})();
```

В заключение рассмотрим одну проблему, связанную с использованием замкнутых выражений. Вспомним, что замкнутые выражения дают возможность ссылаться на переменные, которые существуют внутри родительской функции. Но они не предоставляют значений этих переменных, присвоенных им в момент создания, а дают самые последние значения этих переменных, которые были получены ими в родительских функциях. Наиболее часто эта проблема проявляется при работе цикла. Представим, что есть переменная, используемая в качестве итератора (например, *i*). Внутри цикла `for` создается новая функция, использующая замкнутое выражение, для ссылки на итератор. Проблема в том, что к тому времени, когда будет происходить вызов новых замкнутых функций, они

будут ссылаться на последнее значение итератора (например, на последнюю позицию массива), а не на то значение, которое, возможно, вами ожидалось. В листинге 2.16 показан пример использования безымянных функций, порождающих область видимости для создания экземпляра, в котором можно использовать ожидаемое замкнутое выражение.

Листинг 2.16. Пример использования безымянной функции для порождения области видимости, необходимой для создания множества функций, использующих замкнутые выражения

```
// Элемент, у которого значение ID равно main
var obj = document.getElementById("main");

// Массив элементов для привязки
var items = [ "click", "keypress" ];

// Итерация по всем элементам
for ( var i = 0; i < items.length; i++ ) {
    // Создание самовыполняемой безымянной функции для порождения области
    // видимости
    (function(){
        // Запоминание значения внутри этой области видимости
        var item = items[i];
        // Привязка функции к элементу
        obj[ "on" + item ] = function() {
            // элемент обращается к родительской переменной, которая была
            // успешно увидена в содержимом этого цикла for
            alert( "Спасибо за ваш " + item );
        };
    }) ();
}
```

Понятие замкнутых выражений не так-то легко усвоить; мне понадобилось потратить массу времени и усилий, что бы по-настоящему осознать ту мощь, которая в них заключена. Хорошо, что нашелся замечательный источник, в котором объясняется работа замкнутых выражений в JavaScript — статья Джима Джея (Jim Jey) «JavaScript Closures» опубликованная по адресу http://jibbering.com/faq/faq_notes/closures.html.

В заключение, мы рассмотрим понятие *контекста*, которое является той самой строительной конструкцией, вокруг которой выстраивается вся объектно-ориентированная функциональность JavaScript.

Контекст

У вашего кода в JavaScript всегда будет какая-то разновидность контекста (объект, внутри которого он работает). Это свойственно и другим объектно-ориентированным языкам, но без тех экстремальных обстоятельств, которые встречаются в JavaScript.

Контекст проявляется посредством переменной *this*. Эта переменная всегда будет ссылаться на объект, внутри которого код в данный момент работает. Следует помнить, что глобальные объекты фактически являются свойствами объекта *window*. Это означает, что даже в глобальном контексте переменная *this* будет ссылаться на объект. Контекст может быть довольно мощным инструментом, одним из самых важных для объектно-ориентированного кода. В листинге 2.17 показаны несколько простых примеров контекста.

Листинг 2.17. Пример использования функций в пределах контекста с последующим переключением их контекста на другую переменную

```
var obj = {
  yes: function(){
    // this == obj
    this.val = true;
  },
  no: function(){
    this.val = false;
  }
};

// Мы видим, что свойство в объекте 'obj' отсутствует
alert( obj.val == null );

// Мы запускаем функцию yes и она вносит изменения в свойство val,
// связанное с объектом 'obj'
obj.yes();
alert( obj.val == true );

// А теперь указываем window.no на метод obj.no, и запускаем его
window.no = obj.no;
window.no();

// Это приводит к тому, что объект obj остается неизменным (поскольку
// контекст был переключен на объект window)
alert( obj.val == true );

// а свойство val объекта window стало обновленным.
alert( window.val == false );
```

Возможно, вы обратили внимание, на то, какой неуклюжий код был использован в листинге 2.17 для переключения контекста метода `obj.no` на переменную `window`. К счастью, в JavaScript есть несколько методов, упрощающих понимание и реализацию этого процесса. В листинге 2.18 показаны два разных метода, `call` и `apply`, способные справиться с этой задачей.

Листинг 2.18. Примеры изменения контекста функций

```
// Простая функция, устанавливающая цветовой стиль своего контекста
function changeColor( color ) {
  this.style.color = color;
}

// Ее вызов для объекта window, заканчивающийся неудачей, поскольку в нем
// нет объекта style
changeColor( "white" );

// Поиск элемента, ID которого равен main
var main = document.getElementById("main");
```

```
// Установка для него черного цвета, используя метод call
// Этот метод устанавливает контекст по первому аргументу, и передает все
// остальные аргументы в качестве аргументов функции
changeColor.call( main, "black" );

// Функция, устанавливающая цвет элемента body
function setBodyColor() {
    // Метод apply устанавливает контекст на элемент body, указанный в
    // качестве первого аргумента, второй аргумент представляет собой массив
    // аргументов, передаваемых функции
    changeColor.apply( document.body, arguments );
}

// Установка для элемента body черного цвета
setBodyColor( "black" );
```

Возможно, польза от применения контекста не покажется сразу же столь очевидной, но она несомненно проявится в следующем разделе, где мы рассмотрим объектно-ориентированные свойства JavaScript.

Объектно-ориентированные основы JavaScript

Фраза «объектно-ориентированный JavaScript» страдает явной избыточностью, поскольку язык JavaScript является полностью объектно-ориентированным, и использовать его как-то по-другому просто невозможно. Тем не менее, недостаток многих неопытных программистов (включая и работающих на JavaScript) проявляется в том, что они пишут свой код с функциональным подходом, не используя контекст или группировку. Чтобы обрести исчерпывающее понятие о том, как пишется оптимальный код JavaScript, нужно понять, как работают объекты JavaScript, в чем состоят их отличия от объектов в других языках программирования, и как этими отличиями можно воспользоваться.

В остальной части этой главы мы изучим основы написания объектно-ориентированного кода в JavaScript, а затем, в следующих главах пройдем практикум создания такого кода.

Объекты

Объекты являются основой JavaScript. Фактически все внутри языка является объектом. Основная составляющая мощности языка базируется именно на этом факте. На своем самом базовом уровне объекты существуют как семейства свойств, чем-то напоминая хэш-конструкции, которые можно увидеть в других языках. В листинге 2.19 показаны два основных примера создания объекта с набором свойств.

Листинг 2.19. Два примера создания простого объекта и установки его свойств

```
// создание нового Object-объекта и сохранение его в 'obj'
var obj = new Object();

// Присвоение некоторым свойствам объекта различных значений
obj.val = 5;
obj.click = function(){
    alert( "hello" );
};
```

```
// Эквивалентный код, использующий для определения свойств
// сокращенную запись {...;} и пар ключ-значение
var obj = {

    // Установка имен свойств и значений с использованием пар
    // ключ-значение
    val: 5,
    click: function(){
        alert( "hello" );
    }

};
```

В действительности для объектов ничего кроме этого и не нужно. Но при создании новых объектов, особенно тех, которые наследуют свойства других объектов все уже не так просто.

Создание объектов

В отличие от многих других объектно-ориентированных языков, понятие классов в JavaScript фактически отсутствует. Большинство других объектно-ориентированных языков дает возможность реализовать экземпляр конкретного класса, но к JavaScript это не относится. В JavaScript объекты могут создавать другие объекты, и объекты могут быть унаследованы от других объектов. В общем смысле это понятие называется *наследованием прототипов* и будет рассмотрено чуть позже в разделе «Публичные методы».

И все-таки в JavaScript должен быть способ создания нового объекта, независимо от того какая объектная схема в нем используется. JavaScript делает так, что любая функция может быть также создана в качестве экземпляра объекта. Это звучит несколько запутаннее, чем есть на самом деле. Все очень похоже на разделку куска теста (заготовки объекта) при помощи формы для печенья (в качестве которой выступает конструктор объекта, использующий его прототип).

Пример того, как это все работает, показан в листинге 2.20.

Листинг 2.20. Создание и использование простого объекта

```
// Простая функция, принимающая имя и сохраняющая
// его в текущем контексте
function User( name ) {
    this.name = name;
}

// Создание нового экземпляра этой функции с указанием имени
var me = new User( "My Name" );

// Мы можем убедиться в том, ее имя было установлено в качестве ее
// собственного свойства
alert( me.name == "My Name" );

// И что это экземпляр объекта User
alert( me.constructor == User );

// Теперь, поскольку User() по-прежнему является функцией, что будет, если
```

```
// мы воспользуемся ею именно в этом качестве?
User( "Test" );

// Поскольку ее 'this' контекст не был установлен, он по умолчанию
// устанавливается на глобальный объект 'window', значит, свойство
// window.name равно переданному имени
alert( window.name == "Test" );
```

Листинг 2.20 показывает работу свойства `constructor`. Это свойство существует в каждом объекте, и всегда будет указывать в обратном направлении, на функцию, создавшую этот объект. Благодаря ему можно довольно эффективно получать дубликат объекта, создавая новый объект того же базового типа, но не с теми же свойствами. Пример этому показан в листинге 2.21.

Листинг 2.21. Пример использования свойства `constructor`

```
// Создание нового, простого объекта User
function User() {}

// Создание нового объекта User
var me = new User();

// Также создание нового объекта User (из конструктора, ссылающегося на
// первый объект)
var you = new me.constructor();

// Мы можем увидеть, что свойства constructor фактически одинаковы
alert( me.constructor == you.constructor );
```

Теперь, зная, как создаются простые объекты, настало время их дополнить компонентами, придающими им реальную пользу: контекстными методами и свойствами.

Публичные методы

Публичные (Public) методы полностью доступны конечному пользователю в пределах контекста объекта. Чтобы успешно пользоваться этими публичными методами, доступными каждому экземпляру конкретного объекта, нужно узнать о свойстве под названием *prototype* (прототип), в котором просто содержится объект, действующий в качестве основной ссылки для всех новых копий его родительского объекта. Фактически любое свойство прототипа будет доступно в любом экземпляре объекта. Этот процесс создания и ссылки предоставляет нам легкую версию наследования, обсуждаемую в главе 3.

Поскольку прототип объекта тоже является объектом, вы можете присоединять к нему новые свойства, как ко всякому другому объекту. Присоединение к прототипу новых свойств сделает их частью каждого объекта созданного в качестве экземпляра прототипа, придавая всем этим свойствам публичность (и всеобщую доступность). Соответствующий пример показан в листинге 2.22.

Листинг 2.22. Пример объекта с методами, присоединенными через объект `prototype`

```
// Создание нового конструктора User
function User( name, age ){
    this.name = name;
    this.age = age;
```

```

}

// Добавление к объекту prototype новой функции
User.prototype.getName = function(){
    return this.name;
};

// Добавление еще одной функции к prototype
// Обратите внимание, что ее контекстом будут и объекты, созданные в
// качестве экземпляра прототипа
User.prototype.getAge = function(){
    return this.age;
};

// Создание экземпляра нового объекта User
var user = new User( "Bob", 44 );

// Мы можем убедиться в том, что те два метода, которые мы присоединили
// вместе с объектом, действуют в пределах надлежащего контекста
alert( user.getName() == "Bob" );
alert( user.getAge() == 44 );

```

При создании новых приложений простые конструкторы и простая обработка прототипа объекта входят в арсенал многих разработчиков кода JavaScript. В остальной части этого раздела я собираюсь объяснить суть ряда других технологических приемов, которыми вы можете воспользоваться для извлечения из объектно-ориентированного кода еще больших преимуществ.

Частные методы

Частные (Private) методы и переменные доступны только другим частным методам, частным переменным, и привилегированным методам (рассматриваемым в следующем разделе). Они представляют собой способ определения кода который будет доступен только внутри самого объекта, но не за его пределами. Эта технология основана на работе Дугласа Крокфорда (Douglas Crockford), на чьем веб-сайте представлен ряд документов, детализирующих работу и порядок использования объектно-ориентированных объектов:

- Список статей по JavaScript: <http://javascript.crockford.com/>
- Статья «Private Members in JavaScript»: <http://javascript.crockford.com/private.html>

В листинге 2.23 мы можем увидеть пример того, как частный метод может быть использован в приложении.

Листинг 2.23. Пример частного метода, пригодного только для использования функцией constructor

```

// Конструктор объекта, представляющего учебный класс
function Classroom( students, teacher ) {
    // частный метод, используемый для отображения всех учащихся класса
    function disp() {
        alert( this.names.join(", ") );
    }

    // Сохранение данных о классе в качестве публичных свойств объекта

```

```

this.students = students;
this.teacher = teacher;

// Вызов частного метода для отображения ошибки
disp();
}

// Создание нового объекта classroom
var class = new Classroom( [ "John", "Bob" ], "Mr. Smith" );

// Невыполнимый код, поскольку disp не является публичным свойством объекта
class.disp();

```

Несмотря на свою простоту, частные методы и переменные играют важную роль в избавлении кода от конфликтных ситуаций, позволяя лучше управлять тем, что видят и с чем работают ваши пользователи. Далее мы собираемся рассмотреть привилегированные методы, представляющие собой сочетание частных и публичных методов, которые можно использовать в ваших объектах.

Привилегированные методы

Термин *привилегированные методы* создал Дуглас Крокфорд (Douglas Crockford) для ссылки на методы, которые могут видеть частные переменные и работать с ними (внутри объекта), будучи доступными пользователям в качестве публичных методов. В листинге 2.24 показан пример использования привилегированных методов.

Листинг 2.24. Пример использования привилегированных методов

```

// Создание нового конструктора объекта User
function User( name, age ) {
    // Попытка определить год рождения пользователя
    var year = (new Date()).getFullYear() - age;

    // Создание нового привилегированного метода, имеющего доступ к
    // переменной year, сохраняя при этом публичную доступность
    this.getYearBorn = function(){
        return year;
    };
}

// создание нового экземпляра объекта user
var user = new User( "Bob", 44 );

// Проверка правильности возвращенного года
alert( user.getYearBorn() == 1962 );

// И уведомление о том, что нам недоступно частное свойство объекта year
alert( user.year == null );

```

В основном, привилегированные методы являются динамически сгенерированными, поскольку они добавляются к объекту во время выполнения программы, а не при первой компиляции кода. Хотя эта технология с

точки зрения вычислительных ресурсов считается более затратной, чем привязка простого метода к прототипу объекта, но помимо этого она является более мощной и гибкой. В листинге 2.25 показан пример того, что можно сделать, используя динамически генерируемые методы.

Листинг 2.25. Пример динамически генерируемых методов, которые создаются при создании нового экземпляра объекта

```
// Создание нового объекта user, принимающего объект свойств (properties)
function User( properties ) {
    // Последовательный перебор свойств объекта, и обеспечение
    // для них нужной области видимости (как ранее рассматривалось)
    for ( var i in properties ) { (function(){
        // Создание для свойства нового получателя
        this[ "get" + i ] = function() {
            return properties[i];
        };

        // создание для свойства нового установщика
        this[ "set" + i ] = function(val) {
            properties[i] = val;
        };
    })(); }
}

// Создание нового экземпляра объекта user и передача в него объекта,
// наполняющего его свойства
var user = new User({
    name: "Bob",
    age: 44
});

// Обратите внимание, что свойства имя (name) не существует, поскольку
// внутри объекта properties оно является частным
alert( user.name == null );

// Тем не менее, мы можем получить доступ к его значению, используя новый
// метод getname(), который был динамически сгенерирован
alert( user.getname() == "Bob" );

// В заключение, мы можем убедиться в том, что у нас есть возможность
// установить и получить возраст (age) используя недавно сгенерированные
// функции
user.setage( 22 );
alert( user.getage() == 22 );
```

Мощность динамически генерируемого кода трудно переоценить. Возможность создавать код на основе существующих переменных — исключительно полезное качество; именно оно делает столь мощными макрокоманды в других языках программирования (таких как Lisp), не выходя за контекст современного языка программирования. Далее мы рассмотрим тип метода, который полезен исключительно своими организационными преимуществами.

Статические методы

Замысел, положенный в основу статических методов практически не отличается от замысла, касающегося любой другой нормальной функции. Тем не менее основное отличие состоит в том, что они существуют как статические свойства объекта. Будучи свойствами, они не доступны в контексте экземпляра этого объекта; доступ к ним открыт только в том же контексте, в котором существует и сам основной объект. Тем, кто знаком с традиционным наследованием, существующем в классах, это напоминает статические методы класса.

Фактически единственным преимуществом от создания кода подобным методом является поддержание чистоты пространства имен объекта, понятия, которое более подробно будет рассмотрено в главе 3. В листинге 2.26 показан пример статического метода, присоединенного к объекту.

Листинг 2.26. Простой пример статического метода

```
// Статический метод, присоединенный к объект User
User.cloneUser = function( user ) {
  // Создание и возвращение нового пользователя (user)
  return new User(
    // это клон другого объекта user
    user.getName(),
    user.getAge()
  );
};
```

Статические методы являются первыми из тех, с которыми мы уже имели дело, чье назначение имеет чисто организационный характер. Это важная ступенька для перехода к теме, обсуждаемой в следующей главе. Фундаментальным аспектом разработки кода JavaScript профессионального качества является его возможность быстро и спокойно взаимодействовать с другими частями кода, оставаясь при этом ясным и доступным. Это важная цель, за достижение которой стоит побороться, и одна из тех целей, которую мы надеемся достичь в следующей главе.

Выводы

Важность усвоения понятий, затронутых в этой главе трудно переоценить. Первая половина главы, давая вам хорошее представление о том, как ведет себя язык JavaScript, и как он может быть использован наилучшим образом, является отправной точкой для полного понимания профессионального подхода к использованию JavaScript. Простое представление о том, как действуют объекты, обрабатываются ссылки и определяется область видимости бесспорно может изменить ваш подход к написанию кода на JavaScript.

По мере освоении навыков грамотного программирования на JavaScript, становится все более очевидной важность создания качественного объектно-ориентированного JavaScript-кода. Во второй половине этой главы мною были охвачены вопросы написания различного объектно-ориентированного кода, которые должны были устроить всех, кто пришел из других языков программирования. Это то самое искусство, на котором основана большая часть современного языка JavaScript, дающее вам существенное преимущество при разработке новых и технически прогрессивных приложений.

Глава 3 Создание кода широкого применения

Разработка кода в сотрудничестве с другими программистами является стандартом для многих корпоративных и командных проектов, и здесь особую важность приобретает хорошая система разработки, способствующая совершению заранее продуманных действий. В связи с тем, что в последние годы JavaScript получил широкое признание, объем программного кода, разработанного профессиональными программистами, существенно возрос. Эти подвижки в восприятии и использовании JavaScript привели к существенному прогрессу в сопутствующей практике разработки.

В данной главе мы собираемся рассмотреть ряд способов улучшения кода, совершенствования его организации и повышения качества, которые определяют его пригодность для использования другими программистами.

Стандартизация объектно-ориентированного кода

Первым и наиболее важным шагом в создании кода широкого применения станет стандартизация способов его написания в рамках всего приложения, что особенно важно для объектно-ориентированного кода. Рассматривая в предыдущей главе поведение объектно-ориентированного кода JavaScript, мы убедились в его особой гибкости, позволяющей придерживаться нескольких различных стилей программирования.

Для начала важно продумать наиболее отвечающую вашим потребностям систему написания объектно-ориентированного кода и осуществления объектного наследования (клонирования свойств объекта в новые объекты). Наверное у всех, кто когда-либо создавал какой-нибудь объектно-ориентированный код на JavaScript, выработался свой собственный подход к этой работе, в котором может быть немало неразберихи. В этом разделе мы собираемся рассмотреть, как в JavaScript работает наследование, после чего рассмотрим, как работают различные альтернативные вспомогательные методы, и как можно будет ими воспользоваться в вашем приложении.

Наследование с использованием прототипов

В JavaScript используется уникальная форма создания объектов и наследования, которая называется *прототипным наследованием*. Суть этого метода (в отличие от классической схемы класс-объект, знакомой большинству программистов) состоит в том, что конструктор объекта может наследовать методы от другого объекта, создавая прототип объекта — *prototype*, на основе которого строятся все остальные новые объекты.

Весь этот процесс облегчается наличием *свойства prototype* (Это свойство есть у любой функции, а раз конструктором может стать любая функция, то это свойство есть и у конструктора). Прототипное наследование сконструировано для одиночного, а не для множественного наследования, тем не менее существуют способы обхода этого ограничения, которые мы рассмотрим в следующем разделе.

Эта форма наследования особенно сложна для понимания в той части, что прототипы не наследуют свойства от других прототипов или других конструкторов; они наследуют их от физических объектов. В листинге 3.1. показан ряд примеров конкретного применения свойства *prototype* для осуществления простого наследования.

Листинг 3.1. Примеры прототипного наследования

```
// Создание конструктора для объекта Person
function Person( name ) {
    this.name = name;
}
```

```

// Добавление нового метода к объекту Person
Person.prototype.getName = function() {
    return this.name;
};

// Создание нового конструктора объекта User
function User( name, password ) {
    // Учтите, что здесь не поддерживается постепенная
    // перегрузка-наследование, например возможность вызова
    // конструктора суперкласса
    this.name = name;
    this.password = password;
};

// Объект User наследует все методы объекта Person
User.prototype = new Person();

// Для объекта User мы добавляет свой собственный метод
User.prototype.getPassword = function() {
    return this.password;
};

```

В предыдущем примере наиболее важной для нас является строка `User.prototype = new Person();`. Давайте разберемся в ее назначении более детально. `User` — это ссылка на функцию конструктора объекта `User`. При помощи выражения `new Person()` создается новый объект `Person`, использующий конструктор `Person`. Результат его работы присваивается в качестве значения свойству `prototype` конструктора `User`. Это означает, что при каждом использовании выражения `new User()`, новый объект `User` будет обладать всеми методами, которые были у объекта `Person`, на тот момент, когда вы использовали выражение `new Person()`.

Помня об особенностях этой технологии, рассмотрим ряд различных надстроек, которые были написаны программистами для упрощения процесса наследования в JavaScript.

Классическое наследование

Классическое наследование представляет собой форму, знакомую большинству разработчиков. У вас есть классы с методами, на основе которых могут быть созданы экземпляры объектов. Новичкам объектно-ориентированного программирования на JavaScript свойственно предпринимать попытки подражания этому стилю программного конструирования, хотя немногим удается понять, как это сделать правильно.

К счастью, один из мастеров программирования на JavaScript, Дуглас Крокфорд (Douglas Crockford), задался целью разработать простой набор методов, которые можно было бы использовать в JavaScript для имитации такого же наследования, как и при использовании классов, о чем он рассказал на своем веб-сайте <http://javascript.crockford.com/inheritance.html>.

В листинге 3.2 показаны три функции, которые он построил для создания полноценной формы классического наследования в JavaScript. Каждая из функций претворяет в жизнь особый аспект наследования: наследование отдельной функции, наследование всех компонентов отдельного родительского объекта, и наследование индивидуальных методов от нескольких родителей.

Листинг 3.2. Три функции, созданные Дугласом Крокфордом для имитации в JavaScript классического стиля наследования

```

// Простой вспомогательный метод, позволяющий привязать новую функцию к
// прототипу объекта
Function.prototype.method = function(name, func) {
    this.prototype[name] = func;
    return this;
};

// Довольно сложная функция, позволяющая весьма изящно наследовать
// функции из других объектов, и сохранять возможность вызова функции
// 'родительского' объекта
Function.method('inherits', function(parent) {
    // Отслеживание, на каком уровне углубления в родительские функции мы
    // находимся
    var depth = 0;

    // Наследование родительских методов
    var proto = this.prototype = new parent();

    // Создание новой 'привилегированной' функции под названием 'uber',
    // которая при вызове выполняет любую функцию, вписанную
    // в наследование
    this.method('uber', function uber(name) {

        var func; // исполняемая функция
        var ret; // возвращение значения функции
        var v = parent.prototype; // родительский прототип

        // Если мы уже находимся внутри другой функции 'uber'
        if (depth) {
            // спуск на необходимую глубину для поиска исходного прототипа
            for(var i=d; i > 0;i+= 1 ) {
                v = v.constructor.prototype;
            }

            // и получение функции из прототипа
            func = v[name];

            // А если это первый вызов 'uber'
        } else {
            // получение выполняемой функции из прототипа
            func = proto[name];

            // Если функция была частью этого прототипа
            if ( func == this[name] ) {
                // переход вместо этого к родительскому прототипу

```

```

        func = v[name];
    }
}

// Отслеживание той глубины, на которой мы находимся в стеке
// наследования
depth += 1;

// Вызов выполняемой функции со всеми аргументами, кроме первого
// (в котором хранится имя исполняемой функции)
ret = func.apply(this, Array.prototype.slice.apply(arguments,
    [1]));

// Сброс глубины стека
depth -= 1;

// Возвращение значения, возвращаемого выполняемой функцией
return ret;
});
return this;
});

// Функция для наследования только двух функций из родительского
// объекта, но не каждой функции, использующей new parent()
Function.method('swiss', function(parent) {
    // Перебор всех наследуемых методов
    for (var i = 1; i < arguments.length; i += 1) {
        // Имя импортируемого метода
        var name = arguments[i];

        // Импорт метода в прототип этого объекта
        this.prototype[name] = parent.prototype[name];
    }

    return this;
});

```

Посмотрим, что именно нам дают эти три функции, и почему мы должны их использовать вместо того, чтобы попытаться создать свою собственную модель прототипного наследования. Замысел их разработки довольно прост:

Function.prototype.method: Служит простым способом присоединения к прототипу конструктора. Это замкнутое выражение работает благодаря тому, что все конструкторы являются функциями, и получают таким образом новый метод `method`.

Function.prototype.inherits: Эта функция может быть использована для обеспечения простого наследования от одного родителя. Код этой функции концентрируется вокруг возможности вызова `this.uber('methodName')` в любом из ваших методов объекта, и предоставлении ему возможности выполнения перезаписываемого им метода родительского объекта. Это один из подходов, не встроенных в модель наследования JavaScript.

Function.prototype.swiss: Это усовершенствованная версия функции `.method()`, которая может быть использована для захвата нескольких методов из одного родительского объекта. Если ее использовать вместе с несколькими родительскими объектами, вы можете получить разновидность функционального, множественного наследования.

Теперь, когда у вас есть представление о том, что нам дают эти функции, листинг 3.3 возвращает нас к примеру `Person-User`, который был рассмотрен в листинге 3.1, но теперь уже с новой, классической разновидностью наследования. Вдобавок к этому вы можете увидеть, какие дополнительные функциональные возможности дает эта библиотека, а также несколько прояснить ситуацию.

Листинг 3.3. Примеры использования JavaScript-функций классической разновидности наследования, разработанных Дугласом Крокфордом

```
// Создание нового конструктора объекта Person
function Person( name ) {
    this.name = name;
}

// Добавление нового метода к объекту Person
Person.method( 'getName', function(){
    return name;
});

// Создание нового конструктора объекта User
function User( name, password ) {
    this.name = name;
    this.password = password;
},

// Наследование всех методов объекта Person
User.inherits( Person );

// Добавление нового метода к объекту User
User.method( 'getPassword', function(){
    return this.password;
});

// перезапись метода, созданного объектом Person,
// но еще один вызов этого метода с использованием функции uber
User.method( 'getName', function(){
    return "Мое имя: " + this.uber('getName');
});
```

Теперь, когда у вас начал проявляться вкус к тем возможностям, которые предоставляются надежной, улучшающей свойства наследования библиотекой JavaScript, нужно рассмотреть некоторые другие, довольно распространенные и популярные методы.

Библиотека Base

Недавним вкладом в области создания объектов и наследования в JavaScript, стала библиотека Base, разработанная Динном Эдвардсом (Dean Edwards). В этой библиотеке предлагается ряд различных способов расширения функциональных возможностей объектов. Дополнительно в ней предлагаются интуитивно понятные средства объектного наследования. Первоначально Дин разрабатывал Base для использования в некоторых своих побочных проектах, включая проект IE7, который служил в качестве полноценного набора обновлений к Internet Explorer. На веб-сайте Дина (<http://dean.edwards.name/weblog/2006/03/base/>) приведены весьма разнообразные примеры, достаточно хорошо раскрывающие возможности библиотеки. В дополнение к ним можно найти ряд примеров в каталоге исходного кода Base: <http://dean.edwards.name/base/>.

Поскольку Base довольно объемная и сложная библиотека, она заслуживает дополнительных разъяснений (которые включены в код, предоставленный в разделе Source Code/Download на веб-сайте Apress — <http://www.apress.com>). Я настоятельно рекомендую в дополнение к чтению этого закомментированного кода просматривать примеры, предоставленные Динном на его веб-сайте, поскольку они могут оказаться исключительно полезными для разъяснения наиболее запутанных моментов.

Но для начала я собираюсь разобраться с несколькими важными аспектами Base, которые могут оказаться очень полезными для ваших разработок. В частности, в листинге 3.4 представлены примеры создания класса, а также наследования от одного родителя, и замещения родительской функции.

Листинг 3.4. Примеры использования библиотеки Base Дина Эдвардса для создания простого класса и организации наследования

```
// Создание нового класса Person
var Person = Base.extend({
    // Конструктор класса Person
    constructor: function( name ) {
        this.name = name;
    },

    // простой метод класса Person
    getName: function() {
        return this.name;
    }
});

// Создание нового класса User, являющегося наследником класса Person
var User = Person.extend({
    // Создание конструктора класса User
    constructor: function( name, password ) {
        // который, в свою очередь, вызывает метод конструктора
        // родительского класса
        this.base( name );
        this.password = password;
    },

    // Создание еще одного простого метода для User
    getPassword: function() {
        return this.password;
    }
});
```

```

    }
  });

```

Посмотрим, как Base удастся достичь трех целей, обозначенных в листинге 3.4, и создать простую форму создания объекта и организации наследования:

Base.extend(...;);: Это выражение используется для создания нового базового объекта конструктора. Функция берет одно свойство — простой объект, содержащий свойства и значения, которые добавлены к объекту и использованы в качестве его прототипных методов.

Person.extend(...;);: Это альтернативная версия синтаксиса *Base.extend()*. Все конструкторы, созданные с помощью метода *.extend()*, получают свой собственный метод *.extend()*, а значит, и возможность непосредственного наследования от них. В листинге 3.4. конструктор *User* создается за счет непосредственного наследования от исходного конструктора *Person*.

this.base();: И, наконец, метод *this.base()* используется для вызова родительской функции, которая была заменена. Заметьте, что это несколько отличается от функции *this.uber()*, используемой в классической библиотеке Крокфорда, поскольку вам не нужно предоставлять имя родительской функции (что может помочь в наведении порядка и разъяснении кода). Среди всех объектно-ориентированных библиотек JavaScript, *Base* обладает наилучшими функциональными возможностями по замене родительских методов.

Лично я считаю, что разработанная Динем библиотека *Base* создает наиболее читаемый, функциональный и понятный объектно-ориентированный код JavaScript. В конечном счете выбор наиболее подходящей библиотеки остается за разработчиком. Далее мы рассмотрим, как объектно-ориентированный код реализован в популярной библиотеке *Prototype*.

Библиотека Prototype

Prototype — это библиотека JavaScript, которая была разработана для работы в связке с популярной веб-средой *Ruby on Rails*. Ее название не стоит путать со свойством конструктора *prototype* — это всего лишь нелепое совпадение.

Если не брать в расчет название, *Prototype* делает JavaScript по внешнему виду и поведению во многом похожим на язык *Ruby*. Чтобы добиться этого эффекта, разработчики *Prototype* воспользовались объектно-ориентированной природой JavaScript, и присоединили к ядру объектов JavaScript ряд функций и свойств. К сожалению, сама библиотека своим создателем вообще не документировалась, но, к счастью, она написана очень понятно, и многие ее пользователи приступили к созданию собственных версий документации. Весь основной код *Prototype* можно совершенно свободно просмотреть на веб-сайте <http://prototype.conio.net/>. А в статье «Painless JavaScript Using Prototype» по адресу <http://www.sitepoint.com/article/painless-javascript-prototype/> можно найти документацию по этой библиотеке.

В этом разделе мы собираемся рассмотреть лишь специфические функции и объекты, которые используются в *Prototype* для создания ее объектно-ориентированной структуры и обеспечения основ наследования. В листинге 3.5 представлен весь код, который *Prototype* использует для достижения этой цели.

Листинг 3.5. Две функции, используемые *Prototype* для воспроизводства объектно-ориентированного кода JavaScript

```

// Создание глобального объекта по имени 'Class',
var Class = {

```

```

// который обладает единственной функцией, создающей новый объект
// конструктора
create: function() {

    // Создание безымянного объекта конструктора
    return function() {
        // This вызывает свой собственный метод инициализации
        this.initialize.apply(this, arguments);
    }
}

// Добавление статического метода к объекту Object, который копирует
// свойства из одного объекта в другой
Object.extend = function(destination, source) {
    // проход через все свойства для их извлечения
    for (property in source) {
        // и добавления к объекту назначения
        destination[property] = source[property];
    }

    // возвращение модифицированного объекта
    return destination;
}

```

Prototype действительно использует всего лишь две определенные функции для создания и управления всей объектно-ориентированной структурой. Изучая код, вы можете обратить внимание на то, что этот подход явно слабее, чем в Base или в классическом методе Крокфорда. В основу этих двух функций положен весьма простой замысел:

Class.create(): Эта функция просто возвращает надстройку в виде безымянной функции, которая может быть использована в качестве конструктора. Этот простейший конструктор всего лишь вызывает и выполняет свойство инициализации объекта. Стало быть, должно, по крайней мере, существовать свойство `initialize`, содержащее функцию вашего объекта; в противном случае код выдаст исключение.

Object.extend(): Эта функция просто копирует все свойства из одного объекта в другой. При использовании имеющегося в конструкторах свойства `prototype`, можно разработать более простую форму наследования (проще, чем исходная форма прототипного наследования, доступная в JavaScript).

Теперь, когда вы узнали, как работает код, положенный в основу Prototype, в листинге 3.6 показываются несколько примеров его использования в самой библиотеке Prototype с целью расширения «родных» объектов JavaScript и придания им нового уровня функциональности.

Листинг 3.6. Примеры использования в Prototype объектно-ориентированных функций для расширения исходных возможностей по обработке строк в JavaScript

```

// Добавление к прототипу String дополнительных методов
Object.extend(String.prototype, {
    // Новая функция удаления из строки тегов HTML

```



```

stripTags: function() {
    return this.replace(/<\/?[>]+>/gi, '');
},

// Превращение строки в массив СИМВОЛОВ
toArray: function() {
    return this.split('');
},

// Превращение текста вида "foo-bar" в 'горбатый' вариант — "fooBar"
camelize: function() {
    // Разбиение строки по дефисам
    var oStringList = this.split('-');

    // Досрочный возврат, если дефисы отсутствуют
    if (oStringList.length == 1)
        return oStringList[0];

    // Дополнительная обработка начала строки
    var camelizedString = this.indexOf('-') == 0
        ? oStringList[0].charAt(0).toUpperCase() +
          oStringList[0].substring(1) : oStringList[0];

    // Превращение первых букв каждой последовательной части в заглавные
    for (var i = 1, len = oStringList.length; i < len; i++) {
        var s = oStringList[i];
        camelizedString += s.charAt(0).toUpperCase() + s.substring(1);
    }

    // и возврат модифицированной строки
    return camelizedString;
}
});

// Пример использования метода stripTags().
// Можете убедиться, что он удаляет из строки весь HTML,
// и оставляет нам строку, состоящую только из одного текста
"<b><i>Hello</i>, world!".stripTags() == "Hello, world!"

// Пример использования метода toArray().
// Мы извлекаем из строки четвертый символ
"abcdefg".toArray()[3] == "d"

// Пример использования метода camelize().
// Он переделывает старую строку в новый формат.
"background-color".camelize() == "backgroundColor"

```

Теперь вернемся к примеру, который уже использовался в этой главе, где фигурировали объекты `Person` и `User`, и объект `User` являлся наследником объекта `Person`. Этот код, использующий объектно-ориентированный стиль `Prototype`, показан в листинге 3.7.

Listing 3-7. Вспомогательные функции `Prototype`, предназначенные для создания классов и осуществления простого наследования

```
// Создание нового объекта Person с использованием пустого конструктора
var Person = Class.create();

// Копирование ряда функций в прототип Person
Object.extend( Person.prototype, {

    // функция, немедленно вызываемая конструктором Person
    initialize: function( name ) {
        this.name = name;
    },

    // Простая функция для объекта Person
    getName: function() {
        return this.name;
    }
});

// Создание нового объекта User с использованием пустого конструктора
var User = Class.create();
// Объект User наследует все функции своего родительского класса
User.prototype = Object.extend( new Person(), {

    // Переписывание старой функции initialize заново
    initialize: function( name, password ) {
        this.name = name;
        this.password = password;
    },

    // добавление новой функции к объекту
    getPassword: function() {
        return this.password;
    }
});
```

Хотя предлагаемые библиотекой `Prototype` объектно-ориентированные технические приемы не являются революционными, они достаточно сильны, чтобы помочь разработчику в создании простого и легко читаемого кода. В конце концов, если вам приходится создавать довольно большой объем объектно-ориентированного кода, то вы, для облегчения задачи, скорее всего, остановите свой выбор на библиотеке `Base`.

Далее мы собираемся рассмотреть приемы подготовки своего объектно-ориентированного кода для широкого использования с кодом других разработчиков и библиотек.

Создание пакета

После завершения работы над вашим великолепным объектно-ориентированным кодом на JavaScript (или во время его написания, если у вас хватает на это мудрости), настает время для доведения его до определенной степени совершенства, позволяющей ему безупречно работать с другими библиотеками JavaScript. К тому же важно понять, что ваш код должен будет использоваться другими разработчиками и пользователями, требования которых могут отличаться от ваших. Возможно, эта задача решается путем написания самого совершенного кода, но ее выполнению может поспособствовать и изучение того, что уже сделано другими.

В этом разделе мы рассмотрим несколько больших библиотек, используемых ежедневно тысячами разработчиков. Каждая из этих библиотек предоставляет уникальные способы управления своей структурой, которые облегчают ее использование и изучение. В дополнение к этому мы рассмотрим несколько способов, которыми можно воспользоваться для очистки своего кода, чтобы у других разработчиков могло сложиться о нем самое благоприятное впечатление.

Организация пространства имен

Довольно важный и в то же время простой технический прием, который можно применить для очистки и упрощения кода, связан с понятием *пространства имен*. На данный момент JavaScript по умолчанию не поддерживает пространства имен (в отличие, к примеру, от Java или Python), поэтому нам приходится обходиться соответствующей, тем не менее, похожей техникой.

В действительности в JavaScript нет ничего, что могло бы употребляться в качестве пространства имен. Тем не менее, воспользовавшись упомянутым выше положением, что в JavaScript все объекты могут иметь свойства, которые в свою очередь могут содержать другие объекты, вы можете создать что-нибудь такое, что выглядит и работает практически так же, как и пространство имен, используемое в других языках программирования. Используя эту технологию, можно создать уникальные структуры, подобные тем, что показаны в листинге 3.8.

Листинг 3.8. Организация пространства имен в JavaScript и его применение

```
// Создание используемого по умолчанию глобального пространства имен
var YAHOO = {};

// Установка некоторых дочерних пространств имен с помощью объектов
YAHOO.util = {};

// Создание заключительного пространства имен, которое содержит свойство с
// функцией
YAHOO.util.Event = {
    addEventListener: function(){ ...; }
};

// Вызов функции внутри конкретно этого пространства имен
YAHOO.util.Event.addEventListener( ...; )
```

Рассмотрим несколько примеров организации пространства имен в ряде различных популярных библиотек, и то, как они вписываются в цельную, расширяемую и дополняемую архитектуру.

Dojo

Dojo — это чрезвычайно популярная среда, удовлетворяющая всем потребностям разработчиков в создании полноценных веб-приложений. Это означает, что в ней содержится масса подчиненных библиотек, которые нуждаются в индивидуальном включении и оценке, поскольку вся библиотека слишком велика, и с ней не так-то легко справиться. Дополнительная информация о Dojo может быть найдена на веб-сайте проекта: <http://dojotoolkit.org/>.

В Dojo имеется целая система создания пакетов, построенная вокруг организации пространства имен в JavaScript. Вы можете импортировать новые пакеты в динамическом режиме, при котором они будут автоматически выполнены и готовы к использованию. В листинге 3.9 показан пример организации пространства имен, используемой в Dojo.

Листинг 3.9. Создание пакетов и организация пространства имен в Dojo

```
<html>
<head>
  <title>Accordion Widget Demo</title>
  <!-- Включение Dojo Framework -->
  <script type="text/javascript" src="dojo.js"></script>
  <!-- Включение различных пакетов Dojo -->
  <script type="text/javascript">
    // Для создания Accordion Container widget импортируются и
    // используются два различных пакета
    dojo.require("dojo.widget.AccordionContainer");
    dojo.require("dojo.widget.ContentPane");
  </script>
</head>
<body>
<div dojoType="AccordionContainer" labelNodeClass="label">
  <div dojoType="ContentPane" open="true" label="Pane 1">
    <h2>Pane 1</h2>
    <p>Nunc consequat nisi vitae quam. Suspendisse sed nunc. Proin...;</p>
  </div>
  <div dojoType="ContentPane" label="Pane 2">
    <h2>Pane 2</h2>
    <p>Nunc consequat nisi vitae quam. Suspendisse sed nunc. Proin...;</p>
  </div>
  <div dojoType="ContentPane" label="Pane 3">
    <h2>Pane 3</h2>
    <p>Nunc consequat nisi vitae quam. Suspendisse sed nunc. Proin...;</p>
  </div>
</div>
</body>
</html>
```

Если вас интересуют вопросы поддержки больших базовых блоков программного кода на JavaScript, то очень мощная пакетная архитектура Dojo безусловно заслуживает внимания. Дополнительно замечу, что очень большой объем библиотеки заставит вас выискивать в ней те функциональные возможности, которые смогут оказаться полезными.

YUI

Есть еще одна библиотека, обслуживающая архитектуру больших пакетов с организованным пространством имен — JavaScript Yahoo UI (<http://developer.yahoo.com/yui/>). Эта библиотека разработана для осуществления и обеспечения решений ряда широко распространенных выразительных средств веб-приложений (таких как перетаскивание элементов). Все эти UI-элементы разбиты на части и распространяются в соответствии с определенной иерархией. Для библиотеки Yahoo UI разработана хорошая документация, которая заслуживает внимания своей полнотой и детализацией.

Для организации функций и свойств в Yahoo UI используется глубокая иерархия пространства имен, во многом похожая на ту, что используется в Dojo. Но в отличие от Dojo, любое «импортирование» внешнего кода осуществляется только вами, а не оператором импортирования. В листинге 3.10 показан пример внешнего вида и работы пространства имен в библиотеке Yahoo UI.

Листинг 3.10. Создание пакетов и организация пространства имен в библиотеке Yahoo UI

```
<html>
<head>
  <title>Yahoo! UI Demo</title>
  <!-- Импорт основной библиотеки Yahoo UI -->
  <script type="text/javascript" src="YAHOO.js"></script>

  <!-- Импорт пакета обработки событий -->
  <script type="text/javascript" src="event.js"></script>

  <!-- Использование импортированной библиотеки Yahoo UI -->
  <script type="text/javascript">
    // Все обработчики событий и утилиты Yahoo содержатся в
    // пространстве имен YAHOO, и дополнительно разбиты на более мелкие
    // пространства имен (подобные 'util')
    YAHOO.util.Event.addListener( 'button', 'click', function() {
      alert( "Спасибо, что щелкнули на кнопке!" );
    });
  </script>
</head>

<body>
  <input type="button" id="button" value="Щелкните!"/>
</body>
</html>
```

Обе библиотеки, и Dojo, и Yahoo UI, очень хорошо справляются с организацией и обслуживанием значительных объемов кода в пределах одного большого пакета. Понимание того, как они это делают, сможет принести вам неоценимую помощь, когда наступит время самим обеспечивать работу архитектуры пакета.

Очистка программного кода

Перед тем как перейти к теме отладки или создания тестовых примеров (что я собираюсь сделать в следующей главе) сначала нужно рассмотреть как создается код, готовый к использованию другими

разработчиками. Если вы хотите, чтобы ваш код продолжил свое существование, и был востребован и модифицирован другими разработчиками, нужно настроиться работу по избавлению кода от таких операторов, которые могли бы быть неправильно истолкованы или использованы. Конечно, можно пройтись по коду и очистить его вручную, но лучше все же воспользоваться инструментом, который поможет заметить ненадежные, потенциально проблемные участки кода. Именно здесь нам может пригодиться JSLint — набор встроенных правил, определяющих участки кода, способные в будущем вызвать проблемы у вас или у других разработчиков. Полноценный анализатор доступен на веб-сайте JSLint по адресу: <http://www.jshint.com/>. Дополнительно все правила и установки JSLint могут быть найдены по адресу: <http://www.jshint.com/lint.html>.

JSLint — это еще один инструмент, разработанный Дугласом Крокфордом (Douglas Crockford) в свойственной ему манере, поэтому если вас что-то не устраивает или не вызывает доверия, то такие положения можно просто проигнорировать. Тем не менее, некоторые из этих правил целиком отвечают здравому смыслу, поэтому я собираюсь разобраться в них поподробнее.

Объявление переменных

Одно из разумных правил, введенных в JSLint гласит о том, что все переменные, используемые в программе, должны быть объявлены перед их использованием. Хотя в JavaScript явные требования по объявлению переменных отсутствуют, но игнорирование этого правила может вызвать путаницу, касающуюся их фактических областей видимости. Например, если нужно присвоить значение необъявленной переменной внутри функции, то как следует рассматривать ее область видимости, внутри функции или глобально? Без просмотра кода тут не обойтись, и этот вопрос неплохо бы прояснить. Пример установленного в JSLint порядка объявления переменных показан в листинге 3.11.

Листинг 3.11. Требования JSLint к объявлению переменных

```
// Неправильное использование переменной
foo = 'bar';

// Правильное использование переменной
var foo;
...;
foo = 'bar';
```

Операторы != и == против операторов !== и ===

Существует одна типичная ошибка, допускаемая разработчиками из-за недостаточного понимания значений false, используемых в JavaScript. В этом языке null, 0, '', false и undefined все идентичны (==) друг другу, поскольку при их вычислении возникает значение false. Это означает, что если вы используете код test == false, то в результате его вычисления будет получено значение true, если переменная test так же имеет значение undefined или равна нулю, что может не совпадать с тем, что вам на самом деле нужно.

И тут на выручку приходят операторы !== и ===. Оба эти оператора рассматривают явное значение переменной (к примеру, null), а не только результат ее вычисления (к примеру, false). JSLint требует, чтобы при работе с false-значениями вместо оператора != или оператора == использовался оператор !== или оператор ===. В листинге 3.12 показан ряд примеров, показывающих, чем отличаются эти операторы друг от друга.

Листинг 3.12. Примеры отличий != и == от !== и ===

```
// Оба результата имеют значение true
null == false
0 == undefined

// Вместо этого в сравнении нужно использовать !== или ===
null !== false
false === false
```

Блоки и скобки

Это правило я воспринимаю с трудом, тем не менее, если вы работаете в общедоступной среде программирования, то есть смысл его придерживаться. Оно гласит, что от использования однострочных блоков следует отказаться. Когда есть условие (к примеру, `if (dog == cat)`), внутри которого только один оператор (`dog = false;`) то скобки, которые обычно требуются для оформления этого условия, можно опустить. То же самое справедливо и для блоков `while()` и `for()`. Хотя это сокращение, предоставляемое JavaScript, можно только приветствовать, игнорирование скобок в программном коде может вызвать ряд непонятных последствий для тех, кто не понимает, какой код находится внутри блока, а какой — за его пределами. Эта ситуация достаточно понятно раскрывается в листинге 3.13.

Листинг 3.13. Неправильно оформленные блоки кода, содержащие один оператор

```
// Это вполне допустимый, нормальный код Javascript
if ( dog == cat )
if ( cat == mouse )
mouse = "cheese";

// JSLint требует, чтобы все это было оформлено следующим образом:
if ( dog == cat ) {
    if ( cat == mouse) {
        mouse = "cheese";
    }
}
```

Точки с запятой

Польза от этого последнего положения станет особенно очевидной при рассмотрении следующего раздела о сжатии кода. Точку с запятой в конце оператора, если он единственный в строке в JavaScript, ставить не обязательно. Если ваш код не сжат, то отсутствие точки с запятой выглядит вполне нормально, но как только символы конца строки будут удалены с целью сокращения размера файла, сразу же возникнут проблемы. Чтобы избежать подобной ситуации, не следует забывать, что в конце всех операторов нужно ставить точку с запятой, как показано в листинге 3.14.

Листинг 3.14. Операторы, требующие наличия точек с запятой

```
// Если вы собираетесь сжимать код Javascript, то
// обязательно ставьте точки с запятой в конце всех операторов
```

```
var foo = 'bar';
var bar = function(){
    alert('hello');
};
bar();
```

В конечном счете это последнее положение подводит нас к понятию сжатия кода JavaScript. Если использование JSLint для создания безупречного кода приносит пользу другим разработчикам и вам самим, то сжатие наиболее полезно для ваших пользователей, поскольку оно позволяет увеличить скорость работы с вашим веб-сайтом.

Сжатие

Наиболее важным аспектом распространения библиотеки JavaScript является сжатие кода, позволяющее сократить время передачи информации по сети. Сжатие должно использоваться в качестве последнего шага, перед самым выпуском вашего кода в эксплуатацию, поскольку зачастую в результате этой операции распознаваемость кода резко ухудшается. Существует три разновидности сжатия кода JavaScript:

- Сжатие, при котором из кода просто удаляются все пустые места и комментарии, после чего в коде остается только самое необходимое.
- Сжатие, при котором удаляются все пустые места и комментарии, а также сокращаются имена всех переменных.
- Сжатие, при котором делается все, ранее перечисленное, а также сокращается до минимума длина всех слов вашего кода, а не только имен переменных.

Я собираюсь рассмотреть две различные библиотеки: JSMIn и Packer. JSMIn подпадает под первую категорию сжатия (удаления всего, что не имеет отношения к коду), а Packer подпадает под третью категорию (полное сжатие всех слов).

JSMIn

Идея, заложенная в JSMIn довольно проста. При обработке блока кода JavaScript из него удаляются все ничего не значащие символы, и остается только чистый функциональный код. В JSMIn это достигается простым удалением всех лишних пустых символов (включая знаки табуляции и коды конца строк) и всех комментариев. Интернет-версия программы сжатия выложена по адресу: <http://www.crockford.com/javascript/jsmin.html>.

Чтобы понять, что происходит с кодом при его обработке JSMIn, мы возьмем блок кода (показанный в листинге 3.15), пропустим его через минимизатор, и посмотрим результат в листинге 3.16.

Листинг 3.15. Код определения типа пользовательского браузера

```
// (c) 2001 Douglas Crockford
// 2001 June 3
// Объект -is- используется для идентификации браузера. Каждая версия
// браузера способна к самоидентификации, но стандартных способов
// осуществления этой операции нет, и некоторые результаты идентификации
// вводят вас в заблуждение. Причина в том, что создатели
// веб-браузеров — неисправимые лжецы. К примеру, браузеры Microsoft
// IE заявляют, что они — Mozilla 4. А Netscape 6 заявляет, что он —
```



```
// version 5.

var is = {
  ie:      navigator.appName == 'Microsoft Internet Explorer',
  java:    navigator.javaEnabled(),
  ns:      navigator.appName == 'Netscape',
  ua:      navigator.userAgent.toLowerCase(),
  version: parseFloat(navigator.appVersion.substr(21)) ||
           parseFloat(navigator.appVersion),
  win:     navigator.platform == 'Win32'
}
is.mac = is.ua.indexOf('mac') >= 0;
if (is.ua.indexOf('opera') >= 0) {
  is.ie = is.ns = false;
  is.opera = true;
}
if (is.ua.indexOf('gecko') >= 0) {
  is.ie = is.ns = false;
  is.gecko = true;
}
```

Листинг 3.16. Сжатая копия кода листинга 3.15

```
// Сжатый код
var is={ie:navigator.appName=='Microsoft Internet Explorer',java:
navigator.javaEnabled(),ns:navigator.appName=='Netscape',ua:
navigator.userAgent.toLowerCase(),version:parseFloat(
navigator.appVersion.substr(21))||parseFloat(navigator.appVersion),win:
navigator.platform=='Win32'} is.mac=is.ua.indexOf('mac')>=0;if(
is.ua.indexOf('opera')>=0){is.ie=is.ns=false;is.opera=true;}
if(is.ua.indexOf('gecko')>=0){is.ie=is.ns=false;is.gecko=true;}
```

Заметьте, что все пробелы и комментарии были удалены, что привело к существенному сокращению общего размера кода.

Вероятно, JSMIn является самой простой утилитой сжатия кода JavaScript. Она отлично подходит в начальной стадии использования сжатия для вводимого в эксплуатацию кода. Но когда настанет необходимость в дальнейшей экономии трафика, вам потребуется перейти на использование Packer, довольно сложной и мощной библиотеки сжатия JavaScript.

Packer

Packer является самой мощной из всех доступных систем сжатия JavaScript. В этой системе, разработанной Дином Эдвардсом (Dean Edwards), используется способ полного сокращения размера кода с его последующим развертыванием и выполнением на лету. Благодаря этой технологии, Packer создает оптимально наименьший возможный размер кода. Его можно рассматривать как самораспаковывающийся ZIP-файл для кода JavaScript. Интернет-версия сценария Packer доступна по адресу: <http://dean.edwards.name/packer/>.

Сценарий Packer слишком большой и сложный, поэтому пытаться создавать что-либо подобное своими силами я бы не советовал. К тому же сгенерированный код содержит пару сотен служебных байтов (чтобы

обеспечить самораспаковку), поэтому для использования со слишком коротким кодом эта система не подходит (для этого лучше подойдет JSMIn). Но для больших файлов она безусловно является идеальным вариантом. Листинг 3.17 показывает выдержку из самораспаковывающегося кода, сгенерированного сценарием Packer.

Листинг 3.17. Часть кода, сжатая с использованием Packer

```
eval(function(p,a,c,k,e,d){e=function(c){return c.toString(36)};if(!''.replace(/^/,String)){while(c--){d[c.toString(a)]=k[c]||c.toString(a)}k=[(function(e){return d[e]})];e=(function(){return'\\w+'});c=1};while(c--){if(k[c]){p=p.replace(new RegExp('\\b'+e(c)+'\\b','g'),k[c])}}return p}('u 1={5:2.f==\'t s
r\',h:2.j(),4:2.f==\'k\',3:2.l.m(),n:7(2.d.o(p))||7(2.d),q:2.g==\'i\'1.
b=1.3.6(\'b\')>=0;a(1.3.6(\'c\')>=0){1.5=1.4=9;1.c=e}a(1.3.6(\'8\')>=0){1.5=
1.4=9;1.8=e}',31,31,'|is|navigator|ua|ns|ie...;.
```

Пользу от сжатия вашего кода, особенно если для этого используется сценарий Packer, трудно переоценить. В зависимости от того, как написан код, зачастую удастся уменьшить его размер более, чем на 50%, что с точки зрения ваших пользователей отразится в лучшую сторону по времени загрузки, а это должно быть одной из главных целей для любого приложения JavaScript.

Распространение

Финальный шаг JavaScript-разработки является необязательным и большей частью зависит от конкретной ситуации. Если вы создаете код для себя или для компании, то, скорее всего, вы его просто распространите среди других разработчиков или выложите его для использования на своем веб-сайте.

Но если вы разработали интересный фрагмент кода и хотите, чтобы весь остальной мир распорядился им по своему усмотрению, то вам понадобится служба наподобие JavaScript Archive Network (JSAN). Эта служба была основана несколькими Perl-разработчиками, которым понравилось использовать функциональные возможности и практические преимущества архива CPAN (Comprehensive Perl Archive Network). Подробнее о JSAN можно узнать на веб-сайте <http://openjsan.org/>.

Требования JSAN заключаются в том, чтобы все передаваемые в архив модули были написаны в безупречно отформатированном объектно-ориентированном стиле, соответствующим архитектуре конкретного модуля. Кроме того, что JSAN является центральным хранилищем кода, в нем имеются средства, пользуясь которыми можно импортировать требования о взаимозависимости вашего кода и внешних модулей JSAN. Они могут значительно упростить написание взаимосвязанных приложений, и избавиться от волнений на счет того, какие модули уже были установлены пользователем. Чтобы понять, как работает типичный модуль JSAN, рассмотрим один из простых модулей, DOM.Insert, доступный по адресу: <http://openjsan.org/doc/r/rk/rkinyon/DOM/Insert/0.02/lib/DOM/Insert.html>.

Этот модуль принимает строку HTML и вставляет ее в конкретное место веб-страницы. Кроме того, что он прекрасно объектно-ориентирован, он также требует для своей работы и загружает два других модуля JSAN, оба из которых показаны в листинге 3.18.

Listing 3-18. Пример модуля JSAN (DOM.Insert)

```
// Мы пытаемся включить некоторые другие модули, используя JSAN
```

```
try {
```

```

// загрузка двух необходимых для работы библиотек JSAN
JSAN.use( 'Class' )
JSAN.use( 'DOM.Utills' )

// если загрузка JSAN не состоялась, будет вызвано исключение
} catch (e) {
    throw "DOM.Insert requires JSAN to be loaded";
}

// Проверка существования пространства имен DOM
if ( typeof DOM == 'undefined' )
    DOM = {};

// Создание нового конструктора DOM.Insert, являющегося наследником объекта
// 'Object'
DOM.Insert = Class.create( 'DOM.Insert', Object, {
    // Конструктор, принимающий два аргумента
    initialize: function(element, content) {
        // Элемент, в который вставляется HTML
        this.element = $(element);

        // Вставляемая строка HTML
        this.content = content;

        // Попытка вставки строки HTML с использованием способа,
        // предусмотренного в Internet Explorer
        if (this.adjacency && this.element.insertAdjacentHTML) {
            this.element.insertAdjacentHTML(this.adjacency, this.content);

            // при других обстоятельствах, попытка использовать способ,
            // предусмотренный в W3C
        } else {
            this.range = this.element.ownerDocument.createRange();
            if (this.initializeRange) this.initializeRange();
            this.fragment =
                this.range.createContextualFragment(this.content);
            this.insertContent();
        }
    }
});

```

Стремление иметь понятно написанный объектно-ориентированный легко встраиваемый код JavaScript, должно стать критерием разработки как для вас самих, так и для любого другого разработчика. Именно это положение мы собираемся положить в основу изучения остальной части языка JavaScript. Поскольку JavaScript продолжает набирать силу, важность такого стиля разработки будет только усиливаться, он будет становиться все более полезным и распространенным.

Выводы

В этой главе мы рассмотрели различные способы построения структур кода, предназначенного для широкого применения. Используя объектно-ориентированные технологии, изученные в предыдущей главе, мы смогли применить их для создания четких структур данных, хорошо вписывающихся в среду коллективной разработки. В дополнение к этому мы рассмотрели наилучшие способы создания поддерживаемого кода, сокращения размера файла JavaScript, и создания пакета кода для его распространения. Знания о том, как создается отлично отформатированный, удобный в сопровождении код, уберегут вас от бесконечных часов разочарований.

Глава 4 Инструменты для отладки и тестирования

При разработке программ на любом языке программирования больше всего времени уходит, наверное, на тестирование и отладку кода. Когда ведется разработка программного кода профессионального уровня, гарантия того, что вы создаете полностью протестированный, поддающийся проверке и свободный от ошибок код, становится особенно актуальной. Есть одна особенность, определяющая существенное отличие JavaScript от других языков программирования: у него нет владельца или поддержки со стороны какой-либо компании или организации (в отличие от C#, PHP, Perl, Python или Java). Эта особенность может вызвать затруднения в получении соответствующей базы для тестирования и отладки вашего кода.

Чтобы сберечь нервы и сократить объем работы, затрачиваемый на отлавливание ошибок в коде JavaScript, может пригодиться любое из существующих мощных средств разработки. Подобные инструменты (правда, различного качества) есть у каждого современного браузера. Их применение превращает разработку на JavaScript в более стройный и многообещающий процесс.

В этой главе будут рассмотрены различные инструменты, пригодные для отладки кода JavaScript, а также вопрос создания многократно используемых, надежных тестовых комплектов, пригодных для проверок не только существующих, но и будущих разработок.

Отладка

Тестирование и отладка всегда идут рука об руку. Устраивая коду всесторонние испытания, вы наверняка столкнетесь с невыловленными ошибками, требующими особого внимания. И тут наступает процесс отладки. Освоив лучшие из доступных инструментальных средств, и научившись искать и устранять ошибки в коде, вы сможете гарантировать его качество и ускорить свою работу

Консоль ошибок

Самый доступный инструмент, имеющийся в том или ином виде во всех современных браузерах — это консоль ошибок. Каждый браузер может предложить консоль определенного качества, со своей степенью доступности интерфейса и ценностью сообщений об ошибках. В конечном итоге вы сами придете к решению, какой из браузеров больше подходит для начала процесса отладки, чья консоль ошибок (или другие отладочные расширения) окажутся наиболее подходящими для разработчиков.

Internet Explorer

Популярность браузера не предполагает, что он, соответственно этому качеству, имеет лучший инструментарий для отладки. К сожалению, консоль ошибок Internet Explorer оставляет желать лучшего. Кроме всего прочего, эта консоль по умолчанию отключена, что еще больше запутывает охоту за ошибками, если только вы не используете Internet Explorer в качестве своего заданного по умолчанию браузера (вообще-то очень сомнительно, чтобы любой, уважающий себя JavaScript-разработчик использовал бы его в этом качестве).

Кроме вышеупомянутого неудобства использования, у консоли ошибок Internet Explorer есть и более существенные проблемы:

- Он выводит сообщение только об одной ошибке; чтобы найти сообщения о других ошибках, нужно перейти на них, воспользовавшись системой меню.
- Сообщения об ошибках изложены непонятно, и их логический смысл едва уловим. Точные описания возникших проблем появляются в них крайне редко.
- Строка, содержащей ошибку, всегда указывается со смещением на единицу, т.е. настоящая строка с ошибкой имеет номер на единицу меньше, чем указанная. Если учесть еще и невнятность текста сообщения об ошибке, то вы должны обладать качеством настоящего охотника за ошибками.

Пример сообщения об ошибке, появляющегося в консоли ошибок Internet Explorer, показан на рис. 4.1.

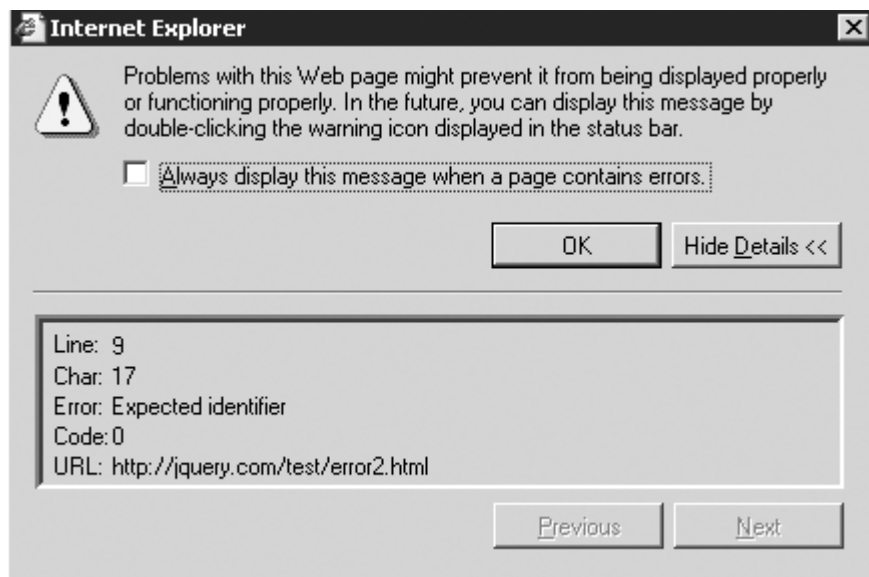


Рис. 4.1. Консоль ошибок JavaScript в Internet Explorer

Как уже упоминалось в начале этого раздела, лучше для начала процесса отладки кода JavaScript воспользоваться другим браузером (а не Internet Explorer). Но если вы все же устранили все ошибки, пользуясь этим браузером, значит, у вас было достаточно времени, чтобы разобраться со всеми странностями Internet Explorer.

Firefox

За последние два года пользовательский интерфейс веб-браузера Firefox был существенно улучшен, значительно упростив для веб-разработчиков создание более качественных веб-сайтов. Консоль ошибок JavaScript претерпела ряд переработок, превратившись в очень полезный инструмент. В консоли ошибок Firefox заслуживают внимания следующие особенности:

- Консоль позволяет вводить произвольные команды JavaScript. Это особенно удобно для определения, какое значение имеет та или иная переменная после загрузки страницы.
- Консоль позволяет проводить сортировку сообщений по их типу, к примеру, ошибки, предупреждения или простые сообщения.
- Самая последняя версия консоли предоставляет наряду с ошибками JavaScript дополнительные предупреждения, касающиеся ошибок или неувязок с таблицами стилей. На неудачно спроектированных веб-сайтах эта особенность может вызвать целый поток ошибок, но в целом она очень полезна для самостоятельного поиска нетипичных ошибок формата.
- Недостаток консоли в том, что она не фильтрует ошибки, чтобы выделить из них те, которые относятся к просматриваемой в данный момент странице, т.е. вы получаете вперемешку ошибки с разных страниц. (Расширение Firebug, которое я буду рассматривать в следующем разделе, поможет решить эту проблему.)

Копия экрана консоли ошибок Firefox показана на рис. 4.2. Обратите внимание на кнопки, которые можно использовать, чтобы переключаться между различными типами сообщений.



Рис. 4.2. Консоль ошибок JavaScript в Firefox

Консоль ошибок Firefox достаточно хороша, но не совершенна. Поэтому для более успешной отладки своих приложений разработчики стремятся воспользоваться различными расширениями Firefox. Некоторые из этих расширений будут рассмотрены далее в этом разделе.

Safari

Браузер Safari один из самых новых на рынке, и один из самых быстроразвивающихся. В процессе этого развития поддержка JavaScript (как в разработке, так и в исполнении кода) была временами довольно неустойчивой. Из-за этого доступ к консоли JavaScript внутри браузера несколько затруднен. Консоль даже не включена в свойства, которые можно просто активировать. Она полностью скрыта в секретном меню отладки, недоступном для обычного пользователя.

Чтобы активировать меню отладки (а вместе с ним и консоль JavaScript) нужно в командной строке запустить команду, показанную в листинге 4.1 (при неработающем Safari).

Листинг 4.1. Команда, предписывающая Safari показать меню отладки

```
defaults write com.apple.Safari IncludeDebugMenu 1
```

При следующем запуске Safari у вас появится новый пункт меню отладки, в который будет включена консоль JavaScript.

Скрытность расположения дает понять, что консоль на данный момент не в лучшей форме. О ней можно сказать лишь следующее:

- Сообщения об ошибках часто носят невнятный характер, и по качеству находятся практически на том же уровне, что и сообщения об ошибках в Internet Explorer.
- Номера ошибочных строк присутствуют, но часто сбрасываются в ноль, заставляя начинать все сначала.

- Фильтрация сообщений об ошибках по страницам отсутствует, но у каждого сообщения есть сценарий, выдавший эту ошибку, ссылка на который дается сразу за ее описанием.

Копия экрана консоли ошибок, запущенной в Safari 2.0, показан на рис. 4.3.

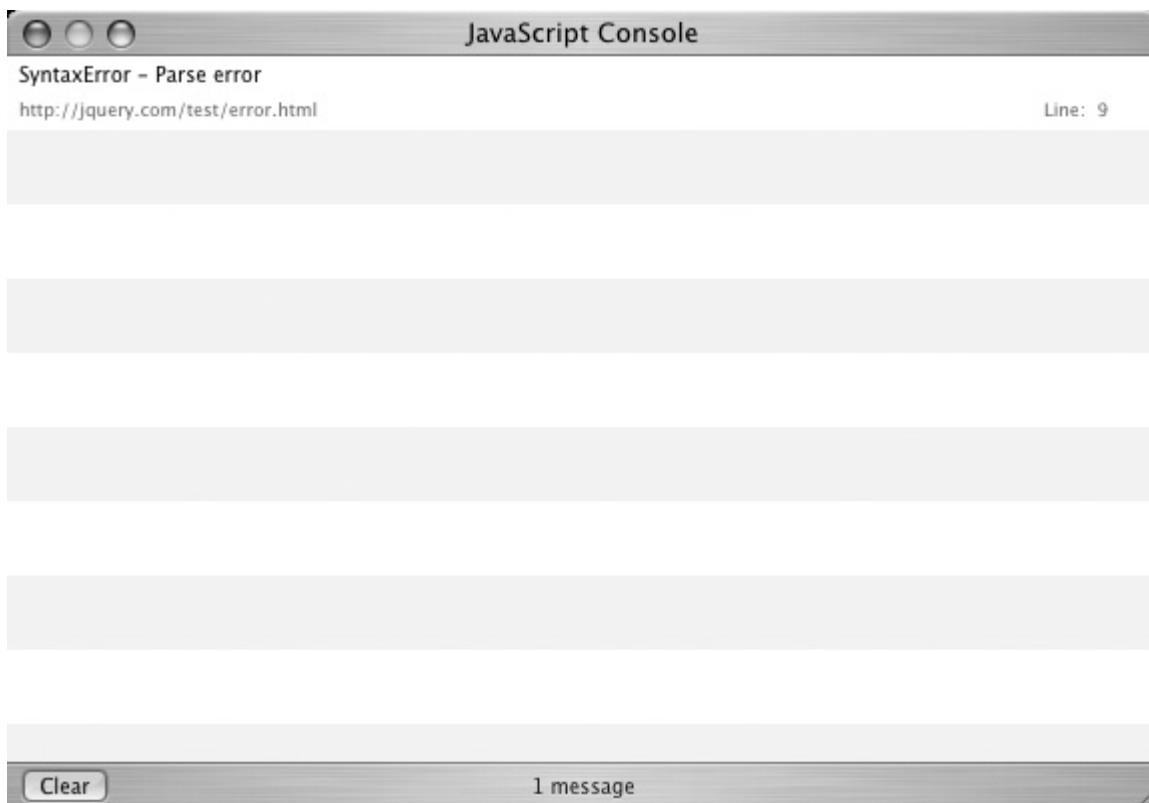


Рис. 4.3. Консоль ошибок JavaScript в Safari

Браузер Safari еще очень далек от того, чтобы стать платформой для веб-разработок. Тем не менее команда разработки WebKit (создающая для Safari движок формирования изображения) неплохо продвинулась в своей работе, улучшив скоростные параметры браузера. Поэтому в ближайшие месяцы и годы стоит ожидать появления для этого браузера новых успешных разработок.

Опера

Последняя консоль ошибок, которую мы собираемся рассмотреть, принадлежит браузеру Опера. К нашему всеобщему удовольствию, разработчики Опера тратят много времени и усилий, чтобы превратить этот браузер в исключительно функциональный и полезный продукт. Вдобавок ко всем свойствам, доступным в консоли ошибок Firefox, в нем имеется следующее:

- Хорошее описание ошибок, позволяющее разобраться в сути проблемы.
- Встроенные фрагменты кода, показывающие, в каком месте допущена ошибка.
- Возможность отфильтровать ошибки по типам (например, JavaScript, CSS и т.д.).

К сожалению, консоль не имеет возможности выполнения команд JavaScript, а жаль, поскольку эта функция могла бы очень пригодиться. Но все-таки в итоге мы получаем превосходную консоль ошибок. На рис. 4.4 показана копия экрана консоли в Opera 9.0.

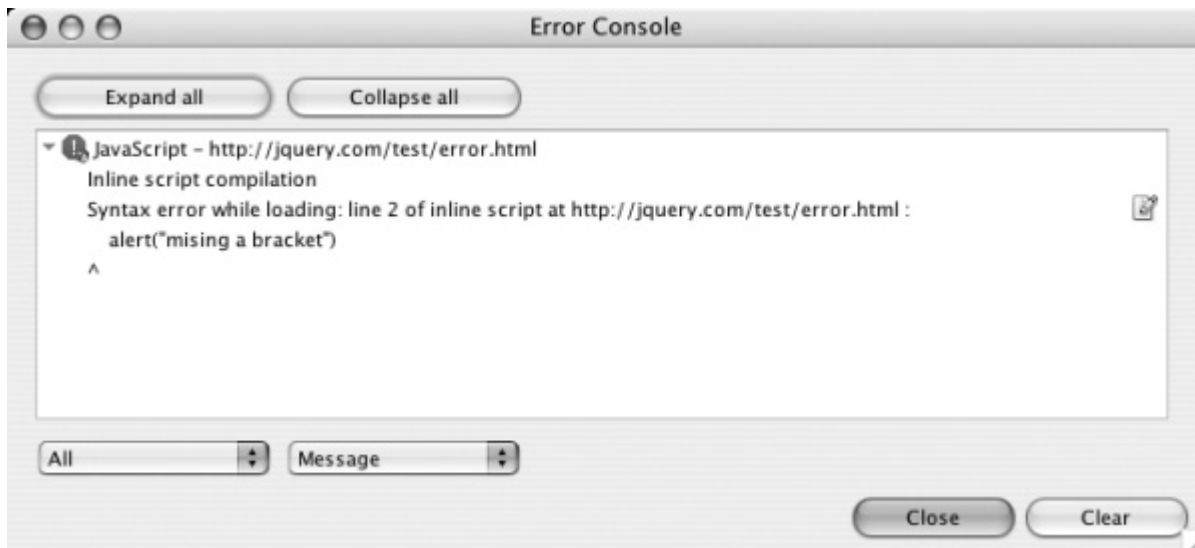


Рис. 4.4. Консоль ошибок JavaScript в Опера

Опера давно относится к веб-разработке со всей серьезностью.

Позиция команды, состоящей из большого количества активных, энергичных разработчиков и составителей технических условий, всегда заключалась в стремлении учесть интересы веб-разработчиков.

Далее я вам покажу ряд связанных с JavaScript расширений браузеров, обладающих довольно мощными возможностями, значительно повышающими производительность разработки.

Инспекторы DOM

Инспектирование DOM — одна из самых полезных, но недостаточно используемых операций, доступных веб-разработчику. Ее можно рассматривать как расширенную версию просмотра исходного кода веб-страницы, позволяющую вам увидеть текущее состояние страницы, после того, как ваш код уже изменил ее содержимое.

Разные DOM-инспекторы на каждом браузере работают по-разному, некоторые предоставляют дополнительные функциональные возможности, позволяющие вам глубже понять, с чем вы работаете. В этом разделе я собираюсь рассмотреть три инспектора, и те особенности, которые отличают их друг от друга.

Инспектор DOM, имеющийся в Firefox

Firefox DOM-инспектор является расширением браузера, который поставляется в предупакованном виде с каждой установкой Firefox (но в установщике он по умолчанию отключен). Это расширение позволяет перемещаться по уже построенному и управляемому HTML-документу. Копия экрана этого расширения показана на рис. 4.5.



Рис. 4.6. DOM-инспектор, встроенный в Safari

Хотя это расширение включено в последние сборки Safari, но сделать его доступным еще более сложно, чем вышеупомянутую консоль JavaScript. Поразительно, как это команда Safari предприняла столько усилий для создания и добавления этих компонентов, а затем скрыла их от разработчиков, желающих их использовать. И так, для активации DOM-инспектора вы должны запустить команду, показанную в листинге 4.2.

Листинг 4.2. Enabling the Safari DOM Inspector

```
defaults write com.apple.Safari WebKitDeveloperExtras -bool true
```

Safari DOM-инспектор имеет довольно большой потенциал для роста и совершенствования, что само по себе хорошо, учитывая талант, проявляемый командой его разработчиков. Но в данный момент для начала работы в качестве базового инструмента лучше использовать Firefox, пока Safari не будет полностью доработан и выпущен в надлежащем виде.

Наглядное представление источника

И, наконец, я хочу представить самый понятный из всех доступных веб-разработчикам DOM-инспектор. Расширение Firefox, названное View Rendered Source[1] (отображение структурной диаграммы), предоставляет дополнительный элемент меню, расположенный рядом с обычным элементом View Source (Просмотр исходного кода страницы), который дает вам выход на полный образ нового HTML-документа, представленный в интуитивно

понятном и доступном виде. Более подробные сведения об этом расширении можно найти на посвященном ему веб-сайте: <http://jennifermadden.com/scripts/ViewRenderedSource.html>.

Вдобавок к предоставлению образа исходного кода, который выглядит очень естественно, этот инструмент дает для каждого уровня документа раскраску, соответствующую его положению в общей иерархии, улучшая ваше представление о том, в каком именно месте кода вы находитесь (см. рис. 4.7).



Рис. 4.7. Расширение View Rendered Source, разработанное для Firefox

Расширение View Rendered Source должно стать стандартом для любого набора инструментов веб-разработчика; его практичность намного превосходит все, что может дать основной пункт меню View Source (Просмотр исходного кода страницы), и позволяет плавно перейти к более сложному расширению Firefox — DOM-инспектору.

Firebug

Firebug — одно из самых важных из появившихся недавно расширений, предназначенных для разработки на JavaScript. Это расширение, созданное Джо Хьюиттом (Joe Hewitt), служит в качестве полного пакета для JavaScript-разработчика. В его состав входит консоль ошибок, отладчик и DOM-инспектор. Более подробная информация об этом расширении может быть найдена на посвященном ему веб-сайте: <http://www.joehewitt.com/software/firebug/>.

Главное преимущество такого обилия интегрированных в единое целое инструментов состоит в том, что вы получаете более конкретное представление о причине возникновения проблемы. К примеру, по щелчку на сообщении об ошибке вы получаете файл JavaScript и строку, в которой произошла ошибка. Здесь вы можете установить контрольные точки останова программы, которые могут использоваться, для пошагового выполнения сценария, и получения лучшего представления, о том, где возникает ошибка. Копия экрана этого расширения показана на рис. 4.8.

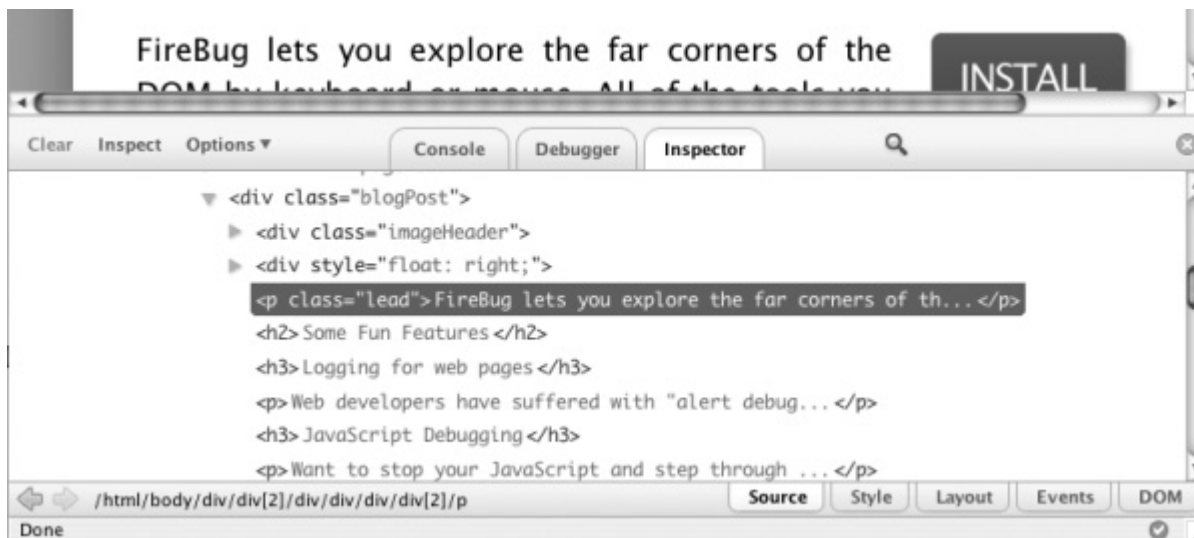


Рис. 4.8. Отладочное расширение Firebug

Пока из современных инструментов еще не появилось ничего лучшего, чем Firebug. Я настоятельно рекомендую выбрать Firefox в сочетании с расширением Firebug в качестве базовой платформы для программирования на JavaScript.

Venkman

Последним элементом, завершающим составление нашего пазла расширений для разработки на JavaScript, будет расширение Venkman. Созданное как часть браузера Mozilla, Venkman является условным названием для проекта отладочного инструмента JavaScript, запущенного разработчиками Mozilla. Более подробная информация об этом проекте и обновленном расширении для Firefox может быть найдена на следующих веб-сайтах:

- Проекта Mozilla Venkman: <http://www.mozilla.org/projects/venkman/>
- Venkman для Firefox: <https://addons.mozilla.org/firefox/216/>
- Руководства по Venkman: <http://www.mozilla.org/projects/venkman/venkman-walkthrough.html>

Важность использования подобного расширения по сравнению с расширением Firebug состоит в том, что благодаря его глубокой интеграции в сам движок JavaScript, появляется возможность предоставить дополнительные средства управления тем, что именно делает ваш код. Копия экрана расширения Venkman для Firefox показана на рис. 4.9.

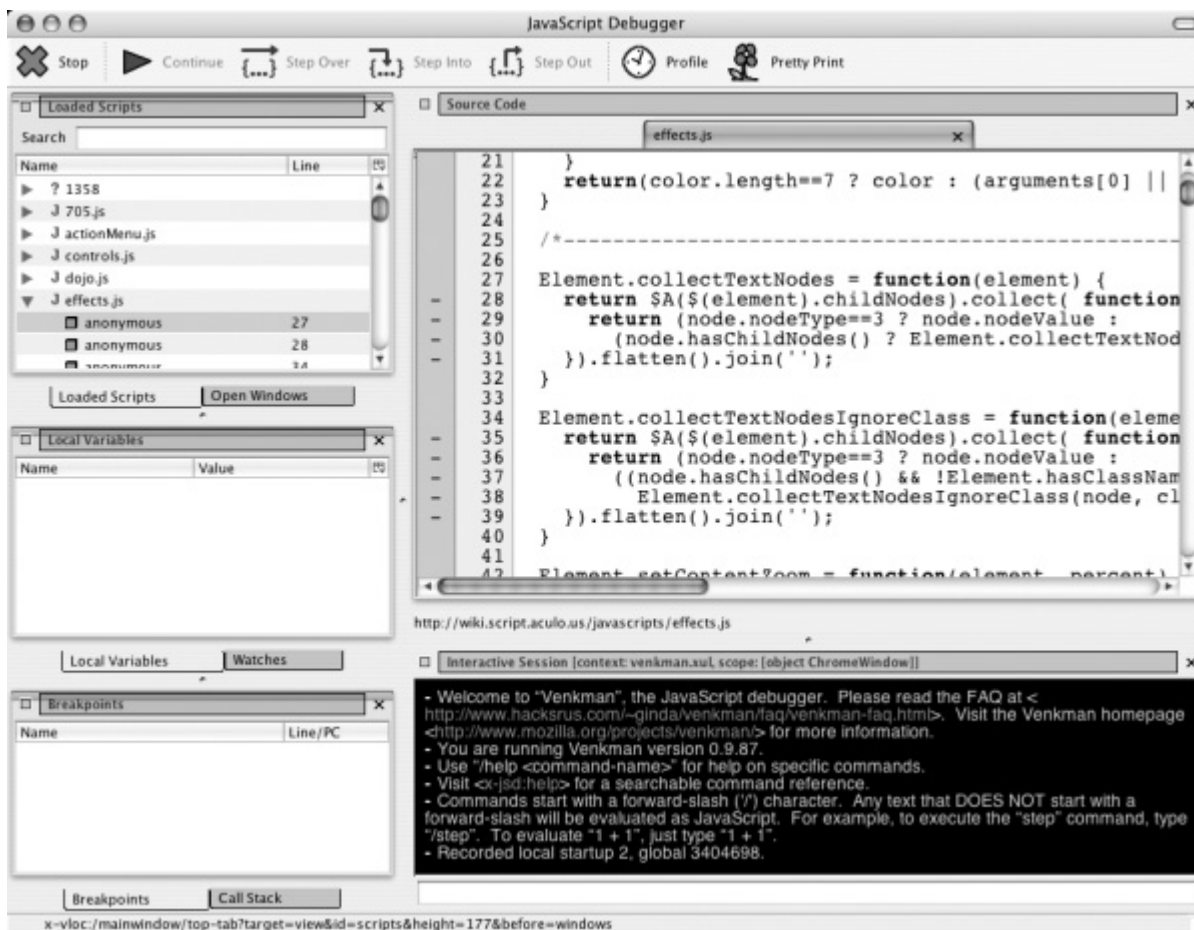


Рис. 4.9. Имеющее давнюю историю расширение Venkman JavaScript debugger, перенесенное на Firefox

Используя все дополнительные средства управления, представленные этим расширением, вы можете точно узнать, какая переменная доступна в конкретной области видимости, а также получить достоверную информацию о состоянии свойств или переменных, и дополнительно к этому получить возможность пошагового выполнения кода и анализа хода его выполнения.

Тестирование

Лично я рассматриваю тестирование кода и создание тестовых примеров как «проверку на будущее». Создавая близкие к реальности тестовые примеры для основного кода или библиотек, вы сможете сэкономить массу времени на отладке, которое обычно тратится на поиск какой-нибудь загадочной ошибки, или, что еще хуже, на невольное внесение в свой код новых ошибок.

Располагая внушительным набором тестовых примеров, что является устоявшейся практикой большинства современных систем программирования, вы сможете оказать помощь не только себе самому, но всем остальным, кто использует вашу программную основу, добавляя к ней новые возможности и устраняя ошибки.

В этом разделе я представлю вам три различные библиотеки, которые могут быть использованы для построения наборов тестовых примеров JavaScript, каждый из которых может быть выполнен кроссбраузерным автоматизированным способом.

JSUnit

Библиотека JSUnit долгое время была чем-то вроде золотого стандарта для блочного тестирования JavaScript. Большая часть ее функциональных возможностей основана на популярном пакете JUnit для Java, что означает ее легкое освоение при условии, что вы знаете, как JUnit работает с Java. На посвященном этой

библиотеке веб-сайте <http://www.jsunit.net/>, имеется масса информации и документации (<http://www.jsunit.net/documentation/>).

Как и большинство других комплексов для блочного тестирования (или по крайней мере те из них, что рассматриваются в этом разделе), этот комплекс содержит три основных компонента:

Прогонщик теста: Эта часть комплекса предоставляет красочное графическое отображение, показывающее насколько далеко продвинулся тест по сравнению с полным прогоном. Она дает возможность загружать тестовые наборы и выполнять их содержимое, регистрируя все предоставляемые ими выходные данные.

Тестовый набор: Это коллекция тестовых примеров (иногда разбитых на несколько веб-страниц).

Тестовые примеры: Это отдельные команды, которые вычисляются в простые выражения типа true/false, тем самым предоставляя вам поддающиеся оценке результаты, позволяющие определить правильность работы вашего кода. По отдельности тестовые примеры могут и не быть полезными, но при их совместном использовании в прогонщике теста, вы можете извлечь выгоду от их взаимодействия.

Все это вместе взятое создает полноценный автоматизированный тестовый комплекс, который может быть использован для запуска и добавления дальнейших тестов. Пример простого тестового комплекса показан в листинге 4.3, а набор тестовых примеров показан в листинге 4.4.

Листинг 4.3. Тестовый комплекс, построенный с использованием JUnit

```
<html>
<head>
  <title>JsUnit Test Suite</title>
  <script src="../../app/jsUnitCore.js"></script>
  <script>
    function suite() {
      var newsuite = new top.jsUnitTestSuite();
      newsuite.addTestPage("jsUnitTests.html");
      return newsuite;
    }
  </script>
</head>
<body></body>
</html>
```

Листинг 4.4. Различные тестовые примеры, которые могут быть использованы в типичной тестовой странице JUnit

```
<html>
<head>
<title>JsUnit Assertion Tests</title>
<script src="../../app/jsUnitCore.js"></script>
<script>
// тестирование выражения на истинность (true)
function testAssertTrue() {
  assertTrue("true should be true", true);
```

```

    assertTrue(true);
}

// Тестирование выражения на ложность (false)
function testAssertFalse() {
    assertFalse("false should be false", false);
    assertFalse(false);
}

// Тестирование двух аргументов на равенство друг другу
function testAssertEquals() {
    assertEquals("1 should equal 1", 1, 1);
    assertEquals(1, 1);
}

// Тестирование двух аргументов на неравенство друг другу
function testAssertNotEquals() {
    assertNotEquals("1 should not equal 2", 1, 2);
    assertNotEquals(1, 2);
}

// Тестирование аргумента на равенство нулевому значению (null)
function testAssertNull() {
    assertNull("null should be null", null);
    assertNull(null);
}

// на неравенство нулевому значению (null)
function testAssertNotNull() {
    assertNotNull("1 should not be null", 1);
    assertNotNull(1);
}

// и еще многое-многое другое ...;
</script>
</head>
<body></body>
</html>

```

По JUnit есть довольно неплохая документация, а поскольку эта библиотека существует уже достаточно долгое время, то вы, скорее всего, сможете найти хорошие примеры ее использования.

J3Unit

Библиотека J3Unit — новичок в мире блочного тестирования JavaScript. Она превосходит JUnit в том, что может быть непосредственно интегрирована с комплексом тестирования на стороне сервера, к примеру, с JUnit или Jetty. Эта библиотека может быть очень полезна для Java-разработчиков, потому что они могут быстро прогнать все свои тестовые примеры для кода как на клиентской, так и на серверной стороне. Но, поскольку не все работают с Java, J3Unit также предоставляет статический режим, который может быть исполнен на вашем

браузере, так же как и любые другие библиотеки блочного тестирования. Более подробные сведения о J3Unit можно найти на веб-сайте, посвященном этой библиотеке: <http://j3unit.sourceforge.net/>.

Поскольку связка тестовых примеров на стороне клиента с кодом на стороне сервера является довольно редким режимом работы этой библиотеки, давайте посмотрим, как в J3Unit работают статические блочные тесты на стороне клиента. К нашему всеобщему удовольствию, они ведут себя практически также, как и другие тестовые комплекты, позволяя легко переключиться на работу с этой библиотекой, что становится понятным из просмотра кода листинга 4.5.

Листинг 4.5. Простой тест, выполняемый с помощью J3Unit

```
<html>
<head>
<title>Sample Test</title>
<script src="js/unittest.js" type="text/javascript"></script>
<script src="js/suiterunner.js" type="text/javascript"></script>
</head>
<body>
<p id="title">Sample Test</p>
<script type="text/javascript">
new Test.Unit.Runner({
  // Тестирование невидимого и видимого режимов отображения элемента
  testToggle: function() {with(this) {
    var title = document.getElementById("title");
    title.style.display = 'none';
    assertNotVisible(title, "title should be invisible");
    element.style.display = 'block';
    assertVisible(title, "title should be visible");
  }},

  // Тестирование добавления одного элемента к другому
  testAppend: function() {with(this) {
    var title = document.getElementById("title");
    var p = document.createElement("p");
    title.appendChild( p );
    assertNotNull( title.lastChild );
    assertEquals( title.lastChild, p );
  }}
});
</script>
</body>
</html>
```

Хотя J3Unit относительно новая библиотека, она подает множество надежд для развития среды блочного тестирования. Если вас заинтересовал ее объектно-ориентированный стиль, я советую к ней присмотреться.

Test.Simple

Последним примером блочного тестирования JavaScript служит еще один новичок. Библиотека Test.Simple была представлена одновременно с созданием JSAN в качестве способа определения общих стандартов тестирования всех представленных там модулей JavaScript. Поскольку Test.Simple получила широкое распространение, у нее имеется обширная документация и множество примеров использования, что в совокупности является весьма важным аспектом использования среды тестирования. Более подробные сведения о Test.Simple (и сопутствующей ей библиотеке Test.More) могут быть найдены в следующих источниках:

- Test.Simple: <http://openjsan.org/doc/t/th/theory/Test/Simple/>
- Документация по Test.Simple: <http://openjsan.org/doc/t/th/theory/Test/Simple/0.21/lib/Test/Simple.html>
- Документация по Test.More: <http://openjsan.org/doc/t/th/theory/Test/Simple/0.21/lib/Test/More.html>

Библиотека Test.Simple предоставляет обширный арсенал методов тестирования наряду с полноценным прогонщиком тестов, обеспечивающим их автоматизированное выполнение. Пример типового теста Test.Simple показан в листинге 4.6.

Листинг 4.6. Использование Test.Simple и Test.More для выполнения тестов

```
// Загрузка модуля Test More (для самотестирования!)
new JSAN('../lib').use('Test.More');

// Планирование проведения шести тестов (чтобы определить все недочеты)
plan({tests: 6});

// Тестирование трех простых примеров
ok( 2 == 2, 'two is two is two is two' );
is( "foo", "foo", 'foo is foo' );
isnt( "foo", "bar", 'foo isnt bar');

// Тестирование с использованием регулярных выражений
like("fooble", /^foo/, 'foo is like fooble');
like("FooBle", /foo/i, 'foo is like FooBle');
like("/usr/local/", '^\/usr\/local', 'regexes with slashes in like' );
```

Мне лично нравится легкость использования библиотек Test.Simple и Test.More, поскольку они не доставляют больших хлопот и помогают поддерживать простоту вашего кода. В конечном счете вам решать, какой из тестовых комплексов лучше подойдет, поскольку подбор такого комплекса для кода — очень важный вопрос, который нельзя обойти стороной.

Вывод

Возможно, весь представленный в этой главе материал не стал особым откровением для искушенных опытом программистов, но объединение всех рассмотренных понятий применительно к JavaScript, в конечном итоге будет способствовать простоте и удобству использования этого языка, и его росту в качестве профессионального языка программирования. Я настоятельно рекомендую, чтобы вы ввели в практику своих разработок процесс отладки и тестирования. Я уверен, что только он позволит вам создавать более качественный и свободный от ошибок код JavaScript.

[1] Теперь это расширение называется View Source Chart (*прим. переводчика*)

Глава 5 Объектная модель документа

Из всех достижений, произошедших за последнее десятилетие в веб-разработке, написание сценариев с использованием объектной модели документа — DOM (Document Object Model) стало, пожалуй, наиболее важной технологией, которой может воспользоваться веб-разработчик для улучшения впечатлений пользователей от своей работы.

Использование DOM в написании сценариев для добавления к странице ненавязчивого кода JavaScript (что означает его невмешательство в работу браузеров, не поддерживающих JavaScript, или с отключенной пользователем поддержкой) дает возможность предоставления весь спектр современных усовершенствований, готовых порадовать ваших пользователей, не досажая тем из них, кто ими не пользуется. Дополнительный эффект, возникший благодаря использованию DOM-сценариев проявился в том, что весь ваш код получил четкое разделение на объекты, которыми стало легче управлять.

К нашему всеобщему удовлетворению, все современные браузеры поддерживают DOM, а также дополнительно поддерживают встроенное DOM-представление текущего HTML-документа. Ко всем объектам имеется свободный доступ из кода JavaScript, что во многом способствует работе современных веб-разработчиков. Умение пользоваться этой технологией и знание самых эффективных приемов ее использования может дать основной толчок к созданию вашего следующего веб-приложения.

В этой главе мы рассмотрим ряд тем, связанных с DOM. Предполагая, что некоторые читатели не знакомы с этой моделью, я начну с ее основ, и проведу вас по всем самым важным понятиям. А для тех, кто уже знаком с моделью DOM, я обещаю показать несколько очень эффективных технологических приемов, которым, как я уверен, вы несомненно обрадуетесь, и начнете их использовать на собственных веб-страницах.

Введение в объектную модель документа

Модель DOM является стандартным способом представления XML-документов (введенным организацией W3C). Она, конечно, не является самой быстросействующей, простой и легкой в использовании, но зато она самая распространенная, и ее реализация существует во многих языках программирования, используемых для веб-разработки (среди которых Java, Perl, PHP, Ruby, Python и JavaScript). DOM создавалась с целью предоставить разработчикам интуитивно понятный способ перехода по иерархии XML-документа. Даже если вы не вполне знакомы с XML, для вас будет очень радостной вестью, что все HTML-документы (которые с точки зрения браузеров являются документами XML) имеют готовое к использованию DOM-представление.

Переходы по DOM

Способом представления в DOM структуры XML, является дерево, по которому можно осуществлять переходы. Вся используемая терминология позаимствована у генеалогического дерева (родители, дети, родные сестры и т.д.). В отличие от обычного семейного дерева, все XML-документы начинаются с одного корневого узла (который называется *элементом document*), в котором содержатся указатели на его детей. Каждый дочерний узел имеет в свою очередь обратный указатель на своего родителя, на своих братьев по уровню и на свои собственные дочерние элементы.

Для ссылки на различные объекты дерева XML, в DOM используется специальная терминология. Каждый объект дерева DOM является узлом. Каждый узел может иметь различный тип, например, элемент, текст или документ. Чтобы продолжить нашу учебу, нам нужно знать, как выглядит документ, отвечающий модели DOM, и как по нему осуществлять переходы после того, как он будет выстроен. Давайте исследуем работу подобного DOM-построения, рассмотрев простой фрагмент HTML-кода:

```
<p><strong>Hello</strong> how are you doing?</p>
```

Каждая часть этого фрагмента разбивается на DOM-узлы, имеющие указатели от каждого узла на его прямых родственников (родителей, детей, сестер). Если вам понадобилось бы составить полную схему существующих родственных отношений, то она бы выглядела, как показано на рис. 5.1. Каждая часть фрагмента (прямоугольники со скругленными углами представляют элементы, а правильные прямоугольники — текстовые узлы) отображена вместе со всеми доступными ссылками.

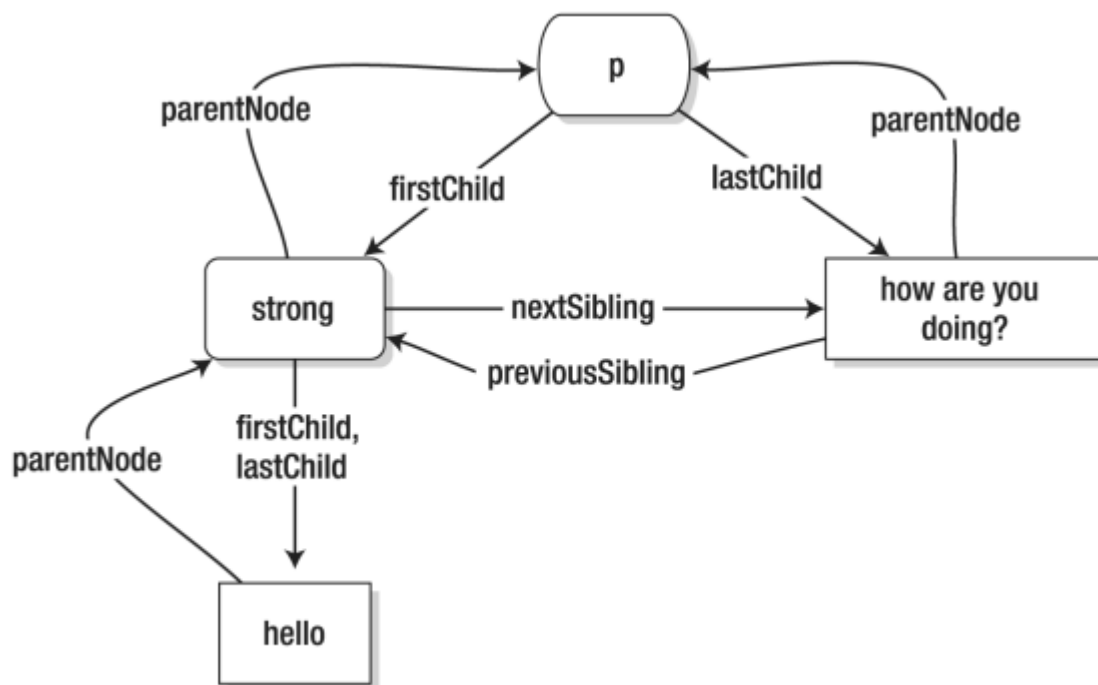


Рис. 5.1. Родственные связи между узлами

Каждый отдельно взятый DOM-узел содержит семейство указателей, которые могут использоваться для ссылок на родственные ему узлы. Воспользуемся этими указателями для освоения перемещений по DOM. На рис. 5.2. показаны все доступные указатели. Каждое из этих свойств, доступное на каждом узле DOM, является указателем на другой DOM-элемент (или содержит значение null, если элемент не существует).

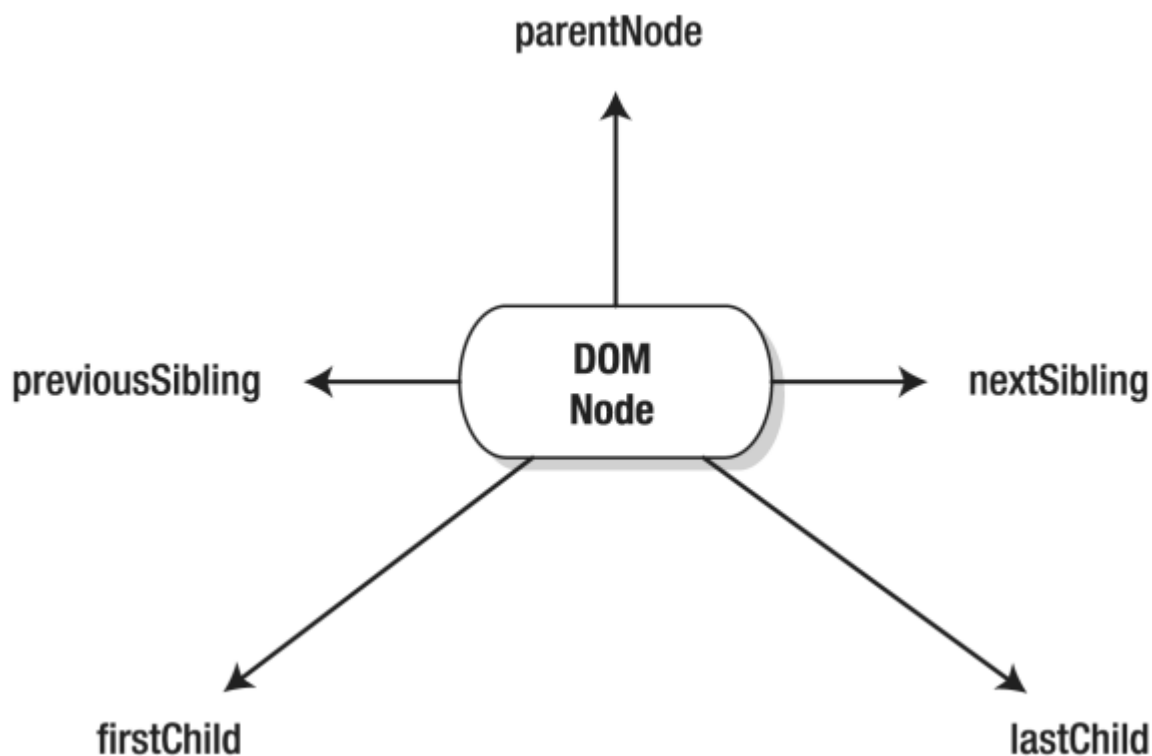


Рис. 5.2. Перемещение по дереву DOM с использованием указателей

Используя только различные указатели, можно перемещаться к любому элементу или текстовому блоку на странице. Лучше всего понять, как это будет работать на практике, рассмотрев обычную HTML-страницу, показанную в листинге 5.1.

Листинг 5.1. Простая веб-страница HTML, являющаяся двойником простого XML-документа

```

<html>
<head>
  <title>Введение в DOM</title>
</head>
<body>
  <h1>Введение в DOM</h1>
  <p class="test">Есть ряд причин, по которым DOM – явление удивительное,
    и вот некоторые из них:</p>
  <ul>
    <li id="everywhere">Ее можно найти повсюду.</li>
    <li class="test">Ею легко пользоваться.</li>
  <li class="test">Она помогает найти все, что нужно, причем
    сделать это довольно быстро.</li>
  </ul>
</body>
</html>
  
```

В примере документа корневым элементом является элемент `<html>`. Получить в JavaScript доступ к корневому элементу довольно просто:

```
document.documentElement
```

Корневой узел имеет все указатели, используемые для перемещений, точно также, как и любой другой DOM-узел. Используя эти указатели вы имеете возможность приступить к просмотру всего документа, перемещаясь на любой нужный элемент. К примеру, чтобы добраться до элемента `<h1>`, можно воспользоваться следующим кодом:

```
// Этот код не работает!
document.documentElement.firstChild.nextSibling.firstChild
```

Но мы тут же натываемся на свое первое препятствие: указатели DOM могут указывать как на текстовые узлы, так и на элементы. А предыдущий оператор на самом деле вместо того, чтобы указывать на элемент `<h1>`, указывает на элемент `<title>`. Почему так происходит? А потому что есть одна из самых неприятных и наиболее спорных сторон XML: пустого пространства. Если присмотреться, то можно заметить, что между элементами `<html>` и `<head>` на самом деле находится символ конца строки, который рассматривается как пустое пространство, а значит, что фактически первым следует текстовый узел, а не `<head>`-элемент. Из этого обстоятельства следует извлечь три урока:

- Создание красивой и понятной HTML-разметки фактически может привести к путанице при попытке просмотреть DOM с использованием только одних указателей.
- Использование для перемещения по документу только одних DOM-указателей может стать слишком громоздким и непрактичным.
- Зачастую непосредственный доступ к текстовым узлам не нужен, он требуется только к тем элементам, которые их окружают.

В связи с этим возникает вопрос: А нет ли более практичного способа для поиска элементов в документе? Конечно же есть! Вооружившись парой полезных функций, вы с легкостью сможете улучшить существующие методы и значительно упростить перемещения по DOM.

Как справиться в DOM с пустым пространством

Вернемся к примеру HTML-документа. Ранее мы уже пытались переместиться к элементу `<h1>` и столкнулись с трудностями, обусловленными наличием внешних текстовых узлов. Ладно бы это касалось только одного-единственного элемента, но что получится, если вам захочется добраться до элемента, который следует за `<h1>`? Вы опять наткнетесь на досадную ошибку пустого пространства, вынуждающую применить `.nextSibling.nextSibling`, чтобы перескочить символы конца строки между элементами `<h1>` и `<p>`. Но не все еще потеряно. Существует методика, показанная в листинге 5.2, которая действует в качестве обхода этой ошибки пустого пространства. Эта специфическая методика удаляет из DOM-документа все текстовые узлы, содержащие только пустые пространства, облегчая тем самым проход по дереву модели. Ее применение не окажет заметного влияния на отображение вашего HTML, но значительно упростит для вас самостоятельные перемещения по документу. Следует заметить, что результаты работы этой функции не являются необратимыми, поэтому при каждой загрузке HTML-документа ее придется запускать заново.

Листинг 5.2. Обход ошибки пустого пространства в XML-документах

```
function cleanWhitespace( element ) {
    // Если element не предоставлен, работать со всем HTML-документом
    element = element || document;
    // В качестве отправной точки использовать первый дочерний узел
    var cur = element.firstChild;

    // Действовать, пока не закончатся дочерние узлы
    while ( cur != null ) {
```

```

// Если узел текстовый, и не содержит ничего, кроме пустого
// пространства
if ( cur.nodeType == 3 && ! /\S/.test( cur.nodeValue ) ) {
    // Удалить текстовый узел
    element.removeChild( cur );

    // А если это элемент
} else if ( cur.nodeType == 1 ) {
    // осуществить рекурсивный вызов вниз по документу
    cleanWhitespace( cur );
}

cur = cur.nextSibling; // перемещение через дочерние узлы
}
}

```

Скажем, вы хотите воспользоваться этой функцией в примере документа для перемещения к элементу который следует за первым элементом `<h1>`. Код, выполняющий эту задачу, может иметь следующий вид:

```

cleanWhitespace();

// Обнаружение элемента H1
document.documentElement
    .firstChild      // Обнаружение элемента Head
    .nextSibling     // Обнаружение элемента <body>
    .firstChild      // Получение элемента H1
    .nextSibling     // Получение смежного абзаца

```

У этого метода есть свои преимущества и недостатки. Самое большое преимущество состоит в том, что получаете какой-то здравый смысл в попытках перемещения по DOM-документу. Но эта методика работает очень медленно, поскольку вам нужно пройти по каждому у отдельно взятому DOM-элементу и текстовому узлу, выискивая текстовые узлы, содержащие только пустые пространства. Если в документе большой объем содержимого, эта методика может существенно замедлить загрузку вашего веб-сайта. К тому же при каждой вставке в документ нового кода HTML, необходимо провести пересканирование этой части DOM, чтобы воспрепятствовать добавлению к документу новых текстовых узлов, состоящих из пустых пространств.

В этой функции применен один важный подход — использование типов узлов. Тип узла может быть определен путем проверки свойства `nodeType` на наличие определенного значения. Есть несколько возможных значений, но три из них встречаются чаще других:

- *Элемент* (`nodeType = 1`): Этот тип соответствует большинству элементов XML-файла. Например, элементы ``, `<a>`, `<p>` и `<body>` имеют значение свойства `nodeType` равное 1.
- *Текст* (`nodeType = 3`): Этот тип соответствует всем текстовым участкам внутри документа. При перемещении по DOM-структуре с помощью методов `previousSibling` и `nextSibling` вы будете часто сталкиваться с участками текста внутри, между элементами.
- *Документ* (`nodeType = 9`): Этот тип соответствует корневому элементу документа. К примеру, в HTML-документе — это элемент `<html>`.

В дополнение к этому для ссылок на различные типы узлов DOM вы можете использовать константы (но только при работе с браузерами, не относящимися к семейству IE). К примеру, чтобы не запоминать значение чисел 1, 3 или 9, вы можете просто воспользоваться константами `document.ELEMENT_NODE`, `document.TEXT_NODE` или `document.DOCUMENT_NODE`. Поскольку постоянная очистка DOM от пустых пространств может стать слишком обременительным занятием, нам нужно исследовать другие способы перемещения по DOM-структуре.

Простое перемещение по DOM-структуре

Используя принцип простого DOM-перемещения (наличие указателей для всех направлений перемещения) вы можете разрабатывать функции, которые больше вам подойдут при перемещении по HTML DOM-документу. Этот принцип основан на том факте, что большинству веб-разработчиков требуются лишь перемещения по DOM-элементам, и очень редко по их собратьям — текстовым узлам. Вам помогут несколько полезных функций, которые можно применить вместо стандартных `previousSibling`, `nextSibling`, `firstChild`, `lastChild` и `parentNode`. В листинге 5.3 показана функция, которая возвращает элемент, который предшествует текущему элементу, или `null`, если предыдущий элемент не найден, работающая наподобие свойства элемента `previousSibling`.

Листинг 5.3. Функция, предназначенная для обнаружения предыдущего сестринского элемента по отношению к переданному ей элементу

```
function prev( elem ) {
    do {
        elem = elem.previousSibling;
    } while ( elem && elem.nodeType != 1 );
    return elem;
}
```

В листинге 5.4 показана функция, возвращающая элемент, следующий за текущим элементом, или `null`, если следующий элемент не найден, работающая наподобие свойства элемента `nextSibling`.

Листинг 5.4. Функция, предназначенная для обнаружения следующего сестринского элемента по отношению к переданному ей элементу

```
function next( elem ) {
    do {
        elem = elem.nextSibling;
    } while ( elem && elem.nodeType != 1 );
    return elem;
}
```

В листинге 5.5 показана функция, возвращающая первый дочерний элемент, работающая наподобие свойства элемента `firstChild`.

Листинг 5.5. Функция для обнаружения первого дочернего элемента по отношению к переданному ей элементу

```
function first( elem ) {
    elem = elem.firstChild;
    return elem && elem.nodeType != 1 ?
        next ( elem ) : elem;
}
```

В листинге 5.6 показана функция, возвращающая последний дочерний элемент, работающая наподобие свойства элемента `lastChild`.

Листинг 5.6. Функция для обнаружения последнего дочернего элемента по отношению к переданному ей элементу

```
function last( elem ) {
    elem = elem.lastChild;
    return elem && elem.nodeType != 1 ?
        prev ( elem ) : elem;
}
```

В листинге 5.7 показана функция, возвращающая родительский элемент, работающая наподобие свойства элемента `parentNode`. Дополнительно вы можете указать ей количество родительских элементов, на которое сразу нужно подняться — например, `parent(elem,2)` будет эквивалентно выражению `parent(parent(elem))`.

Листинг 5.7. Функция для обнаружения родителя указанного элемента

```
function parent( elem, num ) {
    num = num || 1;
    for ( var i = 0; i < num; i++ )
        if ( elem != null ) elem = elem.parentNode;
    return elem;
}
```

Используя эти новые функции, вы можете быстро просматривать DOM-документ, не испытывая волнений насчет текста, который находится между элементами. Например, чтобы обнаружить элемент, следующий за элементом `<h1>`, подобно тому, что мы делали раньше, теперь нужно сделать следующее:

```
// Обнаружение элемента, следующего за элементом <h1>
next( first( document.body ) )
```

Изучая этот код, нужно обратить внимание на две особенности. Первая касается новой ссылки: `document.body`. Все современные браузеры предоставляют ссылку на элемент `<body>` внутри свойства `body` HTML DOM-документа. Этим обстоятельством можно воспользоваться для того, чтобы сделать код компактнее и понятнее. Вторая, как вы могли заметить, заключается в том, что способ написания функции противоречит норме интуитивного восприятия. Обычно, представляя себе перемещение, вы можете сказать: «Начиная с элемента `<body>`, получить первый элемент, а затем получить следующий элемент», но способ, которым это отображено физически, кажется развернутым в обратном направлении. Чтобы обойти эту несурязицу, я теперь рассмотрю несколько способов, чтобы сделать ваш код, предназначенный для перемещений, более понятным.

Привязка к каждому HTML-элементу

В Firefox и Opera имеются мощные и доступные прототипы объектов, названные `HTMLElement`, которые позволяют привязывать функции и данные к каждому отдельному HTML DOM-элементу. Функции, описанные в предыдущем разделе несколько сложны и непонятны, и могут быть заменены другими, улучшенными функциями. Одним из лучших способов для этого является привязка ваших функций непосредственно к прототипу `HTMLElement`, что приведет к их непосредственной привязке к каждому отдельному HTML DOM-элементу. Чтобы это заработало, вам нужно внести в функции, созданные в предыдущем разделе, следующие три изменения:

1. Нужно добавить в начале функции одну строку, чтобы ссылка на элемент была сделана, как на данный объект — `this`, а не получена из перечня аргументов.

2. Нужно убрать аргумент элемента, надобность в котором уже миновала.
3. Нужно привязать функцию к прототипу `HTMLElement`, чтобы у вас появилась возможность использовать ее с каждым HTML-элементом, имеющимся в DOM.

К примеру, функция `next` будет выглядеть, как показано в листинге 5.8.

Листинг 5.8. Динамическая привязка новой функции DOM-перемещения ко всем HTML DOM-элементам

```
HTMLElement.prototype.next = function() {
  var elem = this;
  do {
    elem = elem.nextSibling;
  } while ( elem && elem.nodeType != 1 );
  return elem;
};
```

Теперь вы можете воспользоваться функцией `next` (и всеми другими функциями, после их предварительной переделки) следующим образом:

```
// Простой пример — получение первого <p>-элемента
document.body.first().next()
```

Благодаря этому код становится намного проще и понятнее. Теперь, получив возможность создавать код, который следует естественному ходу размышлений, вы можете сделать свой JavaScript в целом намного понятнее. Если такой стиль написания вас заинтересовал, я настоятельно рекомендую взять на заметку библиотеку `jQuery JavaScript` (<http://jquery.com>), в которой делается большая ставка на использование этой технологии.

ПРИМЕЧАНИЕ

Поскольку `HTMLElement` существует только в трех современных браузерах (`Firefox`, `Safari` и `Opera`) для его работы в `Internet Explorer` нужно предварительно предпринять специальные меры. Речь идет о общедоступной и очень полезной библиотеке, созданной Джейсоном Карлом Дэвисом (`Jason Karl Davis`) (<http://browserland.org>), которая предоставляет доступ к `HTMLElement` (а также ряд других свойств) в двух, не поддерживающих этот прототип объектов браузерах. Дополнительные сведения об этой библиотеке можно найти здесь: <http://www.browserland.org/scripts/htmllement/>.

Стандартные методы DOM

Все современные реализации DOM содержат несколько методов, улучшающих жизнь разработчиков. Их совместное использование с некоторыми собственными функциями делает перемещение по DOM намного приятнее. Для начала рассмотрим два довольно мощных метода, включенных в JavaScript DOM:

`getElementById("everywhere")`: Этот метод, который может быть выполнен исключительно в отношении объекта документа, обнаруживает все элементы, ID которых равен `everywhere`. Это очень мощная функция, предоставляющая самый быстрый способ немедленного доступа к элементу.

`getElementsByTagName("li")`: Этот метод, который может быть выполнен в отношении любого элемента, обнаруживает все нисходящие элементы, у которых в качестве имени тега фигурирует `li`, и возвращает их в `NodeList` (который практически идентичен массиву).

ВНИМАНИЕ

Вы должны понимать, что `getElementById` работает с HTML-документами: он просматривает все элементы и обнаруживает один из них, имеющий атрибут под названием `id`, имеющий определенное значение. Но если вы загружаете удаленный XML-документ и используете `getElementById` (или используете реализацию DOM в любом другом языке, кроме JavaScript), то в нем по умолчанию не используется атрибут `id`. Это сделано намеренно; XML-документ должен определить явным образом, что из себя представляет атрибут `id`, используя обычно XML-определение или схему.

ВНИМАНИЕ

Метод `getElementsByTagName` возвращает `NodeList`. Эта структура по своему виду и поведению во многом похожа на обычный массив JavaScript, с одной важной оговоркой: в нем отсутствуют обычные для массива JavaScript методы `.push()`, `.pop()`, `.shift()` и т.д. Об этом нужно помнить при работе с `getElementsByTagName`, чтобы избежать неприятностей.

Эти методы доступны во всех современных браузерах и могут быть очень полезны для обнаружения определенных элементов. Возвращаясь к предыдущему примеру, в котором мы пытались обнаружить элемент `<h1>`, теперь мы можем сделать следующее:

```
document.getElementsByTagName("h1")[0]
```

Этот код будет гарантированно работать, и всегда возвращать первый `<h1>`-элемент, встречающийся в документе. Еще раз вернемся к документу и представим, что нам нужно получить все ``-элементы и добавить к ним обрамление:

```
var li = document.getElementsByTagName("li");
for ( var j = 0; j < li.length; j++ ) {
    li[j].style.border = "1px solid #000";
}
```

В заключение, представим, что нам нужно сделать полужирным текст в первом ``-элементе, который, как оказалось, имеет связанный с ним и известный нам ID:

```
document.getElementById("everywhere").style.fontWeight = 'bold';
```

Здесь вы, возможно, обратили внимание, что процесс получения отдельного элемента с определенным ID требует большого количества сопроводительного текста, что, впрочем, справедливо и для извлечения элемента по имени тега. Чтобы обойти это обстоятельство и упростить процесс извлечения, можно создать упаковочную функцию:

```
function id(name) {
    return document.getElementById(name);
}
```

В листинге 5.9. показана простая функция, предназначенная для обнаружения элементов внутри HTML DOM-документа по имени тега. Эта функция принимает от одного до двух аргументов. Если предоставляется один аргумент, и это имя тега, то поиск будет вестись во всем HTML-документе. Иначе вы должны предоставить в первом необязательном аргументе DOM-элемент, который будет использован в качестве контекста.

Листинг 5.9. Функция для обнаружения элементов по имени тега внутри HTML DOM-документа

```
function tag(name, elem) {
    // Если контекстный элемент не предоставлен, вести поиск по всему
    // документу
```

```
return (elem || document).getElementsByTagName(name);
}
```

Давайте еще раз вернемся к проблеме обнаружения элемента, который следует за первым элементом `<h1>`. К нашему удовлетворению, код для осуществления этой операции может стать еще короче:

```
// Обнаружение элемента, находящегося сразу же за первым элементом <h1>
next( tag("h1")[0] );
```

Эти функции предоставляют вам мощь, необходимую для быстрого извлечения нужных для работы элементов внутри DOM-документа. Перед тем, как изучить способы применения этой мощи для модификации DOM, нужно накоротке взглянуть на проблемы загрузки DOM при первоначальном запуске вашего сценария.

Ожидание загрузки HTML DOM

При работе с HTML DOM-документами существует одно затруднение, связанное с тем, что код JavaScript может быть исполнен еще до того, как DOM полностью загрузится, что является потенциальной угрозой возникновения в коде ряда проблем. Последовательность операций в браузере выглядит следующим образом:

- Синтаксический анализ HTML.
- Загрузка внешних сценариев и таблиц стиля.
- Выполнение сценариев по мере их разбора в документе.
- Полное построение HTML DOM.
- Загрузка изображений и внешнего контента.
- Завершение загрузки страницы.

Сценарии, находящиеся в заголовке, и загружаемые из внешнего файла выполняются до фактического построения HTML DOM. Как отмечено ранее, это является существенной проблемой, поскольку все сценарии в этих двух местах не будут иметь доступа к DOM. Но, к счастью существует ряд способов обхода этой проблемы.

Ожидание загрузки страницы

Безусловно, самой распространенной методикой является простое ожидание загрузки всей страницы перед выполнением любых DOM-операций. Эта методика может быть использована за счет простой привязки функции, которая должна быть выполнена после загрузки страницы, к событию `load` объекта `window`. События будут подробно рассмотрены в главе 6. В листинге 5.10 показан пример выполнения связанного с DOM кода, после того, как завершится загрузка страницы.

Листинг 5.10. Функция `addEventListener`, предназначенная для привязки обратного вызова к свойству `window.onload`

```
// Ожидание загрузки страницы
// (Используется addEvent, рассмотренная в следующей главе)
addEventListener(window, "load", function() {
    // Выполнение HTML DOM-операций
    next( id("everywhere") ).style.background = 'blue';
});
```

Может быть эта операция и проще всех, но она всегда будет и медленнее всех. Из очередности загрузочных операций можно заметить, что завершение загрузки страницы — это самый последний выполняемый шаг. Это значит, что если у вас на странице существенное количество изображений, видеофрагментов и т.д., пользователи должны какое-то время ждать, пока, наконец, не будет выполнен код JavaScript.

Ожидание загрузки основной части DOM

Вторая методика носит слишком изощренный характер, и ее применение совершенно не рекомендуется. Если помните, в предыдущем разделе я сказал, что встроенные сценарии после того, как DOM будет выстроена. Это верно лишь наполовину. Фактически сценарии выполняются по мере их обнаружения, когда DOM еще находится в процессе построения. Это значит, если сценарий встроено в страницу где-то посередине, то он будет иметь непосредственный доступ к первой половине DOM. Но если сценарий будет встроено в качестве последнего элемента на странице, то вы получите доступ ко всем предыдущим элементам, имеющимся в DOM, давая вам хитрый способ имитировать DOM-загрузку. Реализация этого метода обычно выглядит так, как показано в листинге 5.11.

Листинг 5.11. Определение загрузки DOM путем вставки тега `<script>` (содержащего вызов функции) в конце HTML DOM

```
<html>
<head>
  <title>Тестирование загрузки DOM</title>
  <script type="text/javascript">
    function init() {
      alert( "DOM загружен!" );
      tag("h1")[0].style.border = "4px solid black";
    }
  </script>
</head>
<body>
  <h1>Тестирование загрузки DOM</h1>
  <!-- Основная часть HTML находится здесь -->
  <script type="text/javascript">init();</script>
</body>
</html>
```

В этом примере сценарий встроено в DOM последним элементом; это будет последним элементом, который будет подвергнут анализу и выполнению. Единственное, что здесь будет выполнено — это функция `init`, которая должна содержать любой связанный с DOM код, который нужно выполнить. Самая большая проблема, связанная с этим решением заключается во вносимом им беспорядке: в HTML вносится дополнение только ради того, чтобы определить, загружен ли DOM. Такая методика обычно считается внесением беспорядка, поскольку вы добавляете на веб-страницу дополнительный, ненужный код только для того, чтобы проверить состояние ее загрузки.

Вычисление окончания загрузки DOM

И последняя методика, которая может быть использована для отслеживания загрузки DOM, является, наверное, самой сложной (с точки зрения реализации), но также и наиболее эффективной. Вы получаете простоту привязки к событию загрузки окна в сочетании со скоростью технологии встроенного сценария.

Эта методика работает за счет контроля готовности необходимых вам свойств HTML DOM-документа с максимально возможной физической скоростью, без блокирования браузера. Есть несколько вещей, которые нужно проконтролировать, чтобы понять, что с HTML-документом уже можно работать:

1. *document*: Нужно посмотреть, существует ли уже DOM-документ. Если проверка проводится слишком рано, то скорее всего, будет получен неопределенный результат.

2. `document.getElementsByTagName` и `document.getElementById`: Нужно проверить, есть ли у документа часто используемые функции `getElementsByTagName` и `getElementById`; эти функции будут существовать по их готовности к применению.
3. `document.body`: Чтобы полностью удостовериться в готовности, нужно проверить, был ли полностью загружен элемент `<body>`. Теоретически это должна отловить предыдущая проверка, но я обнаружил примеры, в которой эта проверка не приводила к достаточно надежным результатам.

Используя эти проверки вы получаете возможность получения достаточно четкой картины, показывающей, когда модель DOM будет готова к использованию (Это *вполне подходящий* вариант, на который может быть затрачено всего несколько миллисекунд). Этот метод практически безупречен. Используя только ранее перечисленные проверки, сценарий будет относительно хорошо работать на всех современных браузерах. Но недавно, в связи с последними усовершенствованиями системы кэширования, осуществленными в Firefox, событие загрузки окна может состояться еще до того, как ваш сценарий будет в состоянии определить, готов ли DOM к работе. Чтобы воспользоваться этим преимуществом, я также привязываю проверку к событию загрузки окна, в надежде получить некоторое дополнительное ускорение.

В заключение функция `domReady` соберет ссылки на все функции, которые нужно запускать, когда DOM готова к работе. Как только можно будет считать, что DOM готова, перебираются все ссылки, и функции выполняются одна за другой. В листинге 5.12 показана функция, которая может быть использована для слежения за тем, когда DOM будет полностью загружена.

Листинг 5.12. Функция, предназначенная для отслеживания готовности DOM

```
function domReady( f ) {
    // Если DOM уже загружен, немедленно выполнить функцию
    if ( domReady.done ) return f();

    // Если мы уже дополнили функцию
    if ( domReady.timer ) {
        // внести ее в список исполняемых
        domReady.ready.push( f );
    } else {
        // Привязывание события завершения загрузки страницы,
        // на тот случай если ее загрузка закончится первой.
        // Здесь используется addEvent.
        addEvent( window, "load", isDOMReady );

        // Инициализация массива исполняемых функций
        domReady.ready = [ f ];

        // Проверка DOM на готовность, проводимая как можно быстрее
        domReady.timer = setInterval( isDOMReady, 13 );
    }
}

// Проверка на готовность DOM к перемещению по ее структуре
function isDOMReady() {
    // Если мы уже определили готовность страницы — проигнорировать
    // дальнейшее выполнение
```

```

if ( domReady.done ) return false;

// Проверка доступности некоторых функций и элементов
if ( document && document.getElementsByTagName &&
    document.getElementById && document.body ) {

    // Если они готовы, можно прекратить проверку
    clearInterval( domReady.timer );
    domReady.timer = null;

    // Выполнение всех ожидавших функций
    for ( var i = 0; i < domReady.ready.length; i++ )
        domReady.ready[i]();

    // Сохранение того, что только что было сделано
    domReady.ready = null;
    domReady.done = true;
}
}

```

Теперь мы посмотрим, как это могло бы выглядеть в HTML-документе. Функция `domReady` должна быть использована, также, как используется функция `addEventListener` (рассматриваемая в главе 6), связывая запуск вашей конкретной функции с готовностью документа к перемещению по его элементам и обращению с ними. Для этого примера я поместил функцию `domReady` во внешний файл JavaScript, названный `domready.js`. В листинге 5.13 показано, как можно воспользоваться новой функцией `domReady` для отслеживания загрузки DOM.

Листинг 5.13. Использование функции `domReady` для определения готовности DOM к перемещениям по ее структуре и внесению изменений

```

<html>
<head>
  <title>Тестирование загрузки DOM</title>
  <script type="text/javascript" src="domready.js"></script>
  <script type="text/javascript">
    function tag(name, elem) {
      // Если контекстный элемент не предоставлен, вести поиск по всему
      // документу
      return (elem || document).getElementsByTagName(name);
    }

    domReady(function() {
      alert( "DOM загружен!" );
      tag("h1")[0].style.border = "4px solid black";
    });
  </script>
</head>
<body>
  <h1>Тестирование загрузки DOM</h1>
  <!-- Основная часть HTML находится здесь -->

```



```
</body>
</html>
```

Теперь, когда вы узнали о нескольких способах перемещения по типичному DOM XML-документу, и как обойти сложности с загрузкой HTML DOM-документа, должен быть поставлен вопрос: нет ли каких-нибудь более подходящих способов обнаружения элементов в HTML-документе? К нашему удовольствию, на этот вопрос можно ответить твердым «да».

Обнаружение элементов в документе HTML

Желаемый порядок обнаружения элементов в HTML-документе часто сильно отличается от порядка, связанного с XML-документом. Это звучит несколько нелепо, учитывая, что современный HTML фактически является подмножеством XML; однако HTML-документ содержит ряд существенных отличий, которыми можно воспользоваться.

Для JavaScript/HTML-разработчика есть два наиболее известных преимущества: использование классов и знание CSS-селекторов. С расчетом на их применение можно создать ряд мощных функций, упрощающих и проясняющих перемещение по DOM.

Обнаружение элементов по имени класса

Обнаружение элементов по имени их классов — довольно распространенная технология, популяризированная Симоном Уиллисоном (Simon Willison) (<http://simon.incutio.com>) в 2003 году и впервые показанная Эндрю Хэйвордом (Andrew Hayward) (<http://www.mooncalf.me.uk>). Эта технология довольно проста и понятна: поиск ведется по всем элементам (или подмножеству элементов) в целях обнаружения тех из них, которые обладают указанным классом. Возможная реализация показана в листинге 5.14.

Листинг 5.14. Функция, осуществляющая поиск всех элементов, имеющих определенное имя класса

```
function hasClass(name,type) {
    var r = [];
    // Обнаружение имени класса (работает и при наличии
    // нескольких имен класса)
    var re = new RegExp("(^|\\s)" + name + "(\\s|$)");

    // Ограничение поиска элементами определенного типа
    // или поиск по всем элементам
    var e = document.getElementsByTagName(type || "*");
    for ( var j = 0; j < e.length; j++ )
        // Если элемент имеет нужный класс, добавление его в
        // возвращаемый массив
        if ( re.test(e[j]) ) r.push( e[j] );

    // Возвращение списка соответствующих элементов
    return r;
}
```

Теперь эту функцию можно использовать для быстрого обнаружения любого элемента или любого элемента определенного типа (например, или <r>), имеющего указанное имя класса. При указании названия

тегов, по которым ведется поиск, работа всегда будет вестись быстрее, чем при поиске по всем элементам (*), поскольку элементов, которые нужно просмотреть в поиске нужных, будет меньше. Возьмем, к примеру, наш HTML-документ. Если нужно обнаружить все элементы, имеющие класс `test`, можно воспользоваться следующим кодом:

```
hasClass("test")
```

Если нужно обнаружить только ``-элементы, имеющие класс `test`, можно воспользоваться следующим кодом:

```
hasClass("test", "li")
```

И наконец, если нужно обнаружить первый ``, имеющий класс `test`, можно воспользоваться следующим кодом:

```
hasClass("test", "li")[0]
```

Эта функция и сама по себе обладает достаточной мощностью. Но если ее использовать в сочетании с методами `getElementById` и `getElementsByTagName`, то получится мощный набор инструментов, который можно использовать для проведения с DOM и более сложной работы.

Обнаружение элементов по селектору CSS

Будучи веб-разработчиком, вы уже знаете способ выбора HTML-элементов: CSS-селекторы. CSS-селектор — это выражение, используемое для применения CSS-стилей к набору элементов. С каждой редакцией стандарта CSS (1, 2 и 3) к спецификации селекторов добавляются все новые и новые особенности, позволяющие разработчикам упростить обнаружение нужных элементов. К сожалению, разработчики браузеров совершенно не спешат с предоставлением полной реализации селекторов CSS 2 и 3, а значит, вы можете и не знать о некоторых интересных возможностях, которые ими уже предоставлены. Если вы интересуетесь всеми новинками CSS, я рекомендую изучить W3C страницы, относящиеся к этой теме:

- Селекторы CSS 1: <http://www.w3.org/TR/REC-CSS1#basic-concepts/>
- Селекторы CSS 2: <http://www.w3.org/TR/REC-CSS2/selector.html>
- Селекторы CSS 3: <http://www.w3.org/TR/2005/WD-css3-selectors-20051215/>

В общем, свойства, доступные в каждой спецификации, схожи в том, что каждая последующая редакция содержит также все свойства предыдущих. Но с каждой редакцией прибавляется и количество новых свойств. К примеру, CSS 2 содержит атрибуты и дочерние селекторы, а CSS 3 предоставляет дополнительную языковую поддержку, выбор по типу атрибута и отрицание. Например, все эти выражения представляют из себя вполне допустимые CSS-селекторы:

`#main <div> p`: Это выражение обнаруживает элемент с ID равным `main`, всех его потомков, являющихся `<div>`-элементами, а затем всех потомков, являющихся `<p>`-элементами. Оно является вполне допустимым селектором CSS 1.

`div.items > p`: Это выражение обнаруживает все `<div>`-элементы, имеющие класс `items`, а затем обнаруживает все дочерние `<p>`-элементы. Оно является вполне допустимым селектором CSS 2.

`div:not(.items)`: Это выражение обнаруживает все `<div>`-элементы, у которых нет класса `items`. Оно является вполне допустимым селектором CSS 3.

Возможно, у вас вызвало удивление, почему я рассматриваю CSS-селекторы, если фактически их нельзя использовать для обнаружения элементов (только для применения CSS-стилей). Но и в этой области некоторые

предприимчивые разработчики сумели преодолеть стереотипы и создать реализации CSS-селекторов, способные к полноценной обработке в стиле всех редакций, от CSS 1 до CSS 3. Использование этих библиотек позволит легко и быстро выбрать любой элемент, и выполнить над ним какие-нибудь операции.

cssQuery

Первая общедоступная библиотека с полной поддержкой CSS 1–3 получившая название `cssQuery`, была создана Дином Эдвардсом (Dean Edwards) (`dean.edwards.name`). Заложенный в нее замысел был довольно прост: вы предоставляете ей CSS-селектор, и `cssQuery` обнаруживает все соответствующие элементы. Кроме того, `cssQuery` разбита на несколько подбиблиотек, по одной для каждого уровня CSS-селекторов, и у вас есть возможность при желании отключить поддержку CSS 3, если в ней нет необходимости. Эта выдающаяся и всеобъемлющая библиотека работает на всех современных браузерах (Дин является сторонником кроссбраузерной поддержки). Для использования этой библиотеки вам нужно предоставить ей селектор, и дополнительно, контекстный элемент, в котором ведется поиск. Рассмотрим ряд примеров:

```
// Обнаружение всех <p>, дочерних по отношению к <div>-элементам
cssQuery("div > p");

// Обнаружение всех <div>, <p> и <form>-элементов
cssQuery("div,p,form");

// Обнаружение всех <p> и <div>-элементов, а затем обнаружение всех
// <a>-элементов, которые находятся внутри них
var p = cssQuery("p,div");
cssQuery("a",p);
```

В результате выполнения функции `cssQuery` возвращается массив соответствующих элементов. Теперь вы можете проводить над ними какие-нибудь операции, как и в случае выполнения метода `getElementsByTagName`. Например, чтобы добавить обрамление вокруг всех ссылок на Google, можно сделать следующее:

```
// Добавление обрамления вокруг всех ссылок на Google
var g = cssQuery("a[href^='google.com']");
for ( var i = 0; i < g.length; i++ ) {
    g[i].style.border = "1px dashed red";
}
```

Дополнительные сведения о `cssQuery` можно найти на веб-сайте Дина Эдвардса (Dean Edwards), там же можно загрузить и весь исходный код: <http://dean.edwards.name/my/cssQuery/>.

СОВЕТ

Дин Эдвардс — настоящий волшебник JavaScript; созданный им код просто поразителен. Я настоятельно рекомендую внимательно изучить его библиотеку `cssQuery`, или по крайней мере посмотреть, как на JavaScript пишется отлично расширяемый код.

jQuery

Этот новичок в мире библиотек JavaScript, предоставляет некоторые совершенно новые способы написания кода. Сначала я приписывал ее к «простой» CSS-селекторной библиотеке, во многом похожей на `cssQuery`, пока Дин Эдвардс не выпустил свою выдающуюся библиотеку `cssQuery`, заставив этот код развиваться в несколько в ином направлении. Библиотека обеспечивает полную поддержку CSS 1–3 наряду с некоторыми основными функциональными возможностями XPath. Но на первое место выходят дополнительные возможности

по содействию перемещению по DOM и работе с ее объектами. Как и `cssQuery`, `jQuery` обладает полной поддержкой всех современных браузеров. Рассмотрим ряд примеров выборки элементов с использованием обычного для `jQuery` сочетания CSS и XPath:

```
// Обнаружение всех <div>-элементов, у которых имеется класс 'links', и
// <p>-элементов внутри них
$("div.links [p]")

// Обнаружение всех потомков всех <p> и <div>-элементов
$("p,div").find("*")

// Обнаружение каждой второй ссылки, указывающей на Google
$("a[@href^='google.com']:even")
```

Для использования результатов, полученных от `jQuery`, есть два варианта. Во-первых, можно воспользоваться выражением `$("выражение").get()`, чтобы получить массив соответствующих элементов — точно такой же результат дает применение `cssQuery`. Во-вторых, можно воспользоваться специальными, встроенными в `jQuery` функциями для работы с CSS и с DOM. Итак, если вернуться к примеру с `cssQuery` по добавлению оформления ко все ссылкам на Google, то можно сделать следующее:

```
// Добавление оформления вокруг всех ссылок на Google
$("a[@href^=google.com]").css("border","1px dashed red");
```

Вы можете найти документацию, множество демонстраций и примеров, а также получить возможность выборочной загрузки на веб-сайте проекта `jQuery`: <http://jquery.com/>.

ПРИМЕЧАНИЕ

Нужно отметить, что ни `cssQuery`, ни `jQuery` фактически не требуют для перемещений использования именно HTML-документа; они могут быть использованы с любым XML-документом. Для изучения разновидности перемещений, присущих только XML, прочитайте следующий раздел, посвященный XPath.

XPath

Выражения XPath — невероятно мощный способ перемещений по XML-документам. С учетом многолетней истории существования, предполагается, что везде, где есть реализация DOM, где-то сразу за ней находится и XPath. Выражения XPath, даже при всем их многословии, намного мощнее всего, что может быть написано с использованием CSS-селектора. В таблице 5.1 показано строчное сравнение различных CSS-селекторов и XPath-выражений.

Таблица 5.1. Сравнение селекторов CSS 3 и выражений XPath

Цель	CSS 3	XPath
Все элементы	*	//*
Все <p>-элементы	p	//p
Все дочерние элементы	p > *	//p/*
Элемент по его ID	#foo	//*[@id='foo']
Элемент по его классу	.foo	//*[contains(@class,'foo')]
Элемент с атрибутом	*[title]	//*[@title]
Первый дочерний элемент всех <p>-	p > *:first-	//p/*[0]

элементов	child	
Все <p>-элементы , имеющие дочерние А	Not possible	//p[a]
Следующий элемент	p +	* //p/following-sibling::*[0]

Если вас заинтересовали предыдущие выражения, я рекомендую посмотреть две XPath-спецификации (хотя обычно в современных браузерах полностью поддерживается только XPath 1.0), чтобы получить представление о том, как работают эти выражения:

- *XPath 1.0*: <http://www.w3.org/TR/xpath/>
- *XPath 2.0*: <http://www.w3.org/TR/xpath20/>

Если вы хотите по-настоящему углубиться в эту тему, я рекомендую приобрести книгу издательства O'Reilly «XML in a Nutshell», написанную Эллиоттом Харольдом (Elliote Harold) и Скоттом Минзом (Scott Means) (2004 г.), или книгу издательства Apress «Beginning XSLT 2.0: From Novice to Professional», написанную Джени Теннисон (Jeni Tennison) (2005 г.). В дополнение к этому есть несколько замечательных руководств, которые помогут освоить использование XPath:

- *W3Schools XPath Tutorial*: <http://w3schools.com/xpath/>
- *ZVON XPath Tutorial*: <http://zvon.org/xxl/XPathTutorial/General/examples.html>

В настоящее время XPath не пользуется полной поддержкой в браузерах; для IE и Mozilla имеются полные (хотя и отличающиеся) реализации XPath, а версии для Safari и Opera находятся в стадии разработки. Чтобы обойти это обстоятельство существуют две XPath-реализации, написанные полностью на JavaScript. Вообще-то они довольно медлительны (по сравнению с реализациями на основе браузеров), но будут слаженно работать на всех современных браузерах:

- *XML for Script*: <http://xmljs.sf.net/>
- *Google AJAXSLT*: <http://goog-ajaxslt.sf.net/>

В дополнение к этому, проект под названием Sarissa (<http://sarissa.sf.net/>) нацелен на создание общей надстройки над каждой браузерной реализацией. Это может дать возможность однократного создания кода, связанного с доступом к XML, сохраняя при этом скоростные преимущества от использования поддерживаемого браузером XML-парсинга. Самая большая проблема, связанная с применением этой технологии — сохраняющаяся до сих пор слабая поддержка XPath в браузерах Opera и Safari, устраненная в предыдущих реализациях XPath.

Вообще-то технология использования встроенного в браузер XPath рассматривается как экспериментальная по сравнению с имеющими широкую поддержку решениями на основе JavaScript. Тем не менее, объем использования и популярность XPath только возрастают, и эту технологию определенно нужно рассматривать в качестве претендента на трон CSS-селектора.

Теперь, когда вы располагаете знаниями и инструментарием, необходимым для обнаружения любого DOM-элемента, или даже набора DOM-элементов, настало время рассмотреть, куда всю эту мощь можно применить. Возможно все, от работы с атрибутами и до добавления и удаления DOM-элементов.

Получение содержимого элемента

Содержимое всех DOM-элементов может быть одним из трех вариантов: текста, дополнительных элементов или сочетания текста и элементов. Вообще-то, наиболее часто встречаются первый и второй варианты. В этом разделе мы собираемся рассмотреть распространенные способы извлечения содержимого элементов.

Получение текста, находящегося внутри элемента

Возможно, для тех, кто плохо знаком с DOM, задача получения текста, находящегося внутри элемента является довольно запутанной. Тем не менее, это такая же задача, связанная с работой с HTML DOM- и XML DOM-документами, поэтому если вы узнаете о том, как это, расставит все по своим местам. В примере DOM-структуры, показанной на рис. 5.3, имеется корневой `<p>`-элемент, который содержит элемент `` и блок текста. Элемент `` сам по себе тоже содержит блок текста.

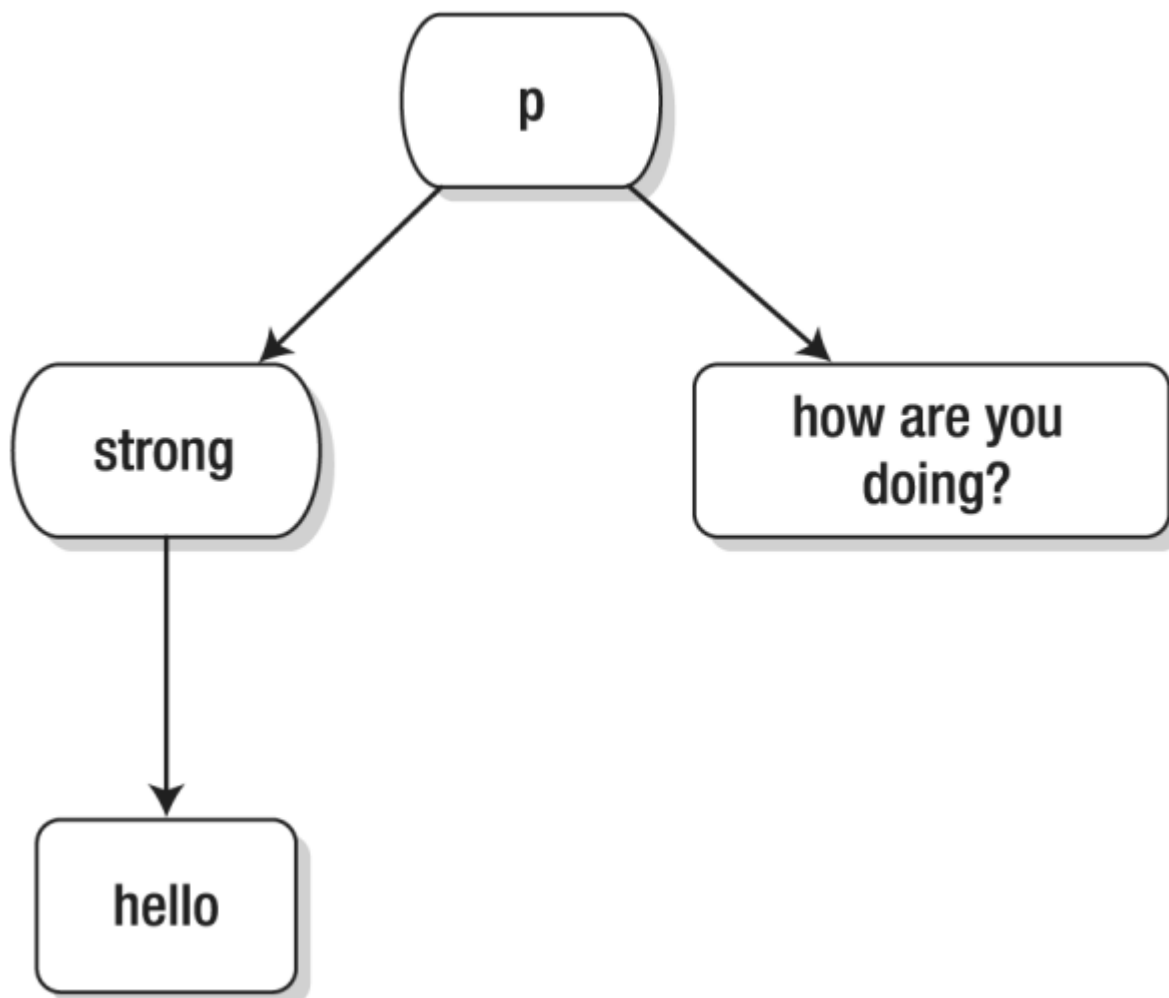


Рис. 5.3. Пример DOM-структуры, содержащей как элементы, так и текст

Посмотрим, как можно получить текст из каждого из этих элементов. Легче начать с элемента ``, поскольку в нем, кроме текстового узла ничего не содержится.

Следует заметить, что существует свойство под названием `innerText`, которое захватывает текст внутри элемента на тех браузерах, которые не работают на движке Mozilla. Для нашей задачи это очень удобно. К сожалению, поскольку это свойство не работает на существенной части парка браузеров, и не работает в XML DOM-документах, нужно рассмотреть жизнеспособные альтернативы.

Весь фокус получения текстового содержимого элемента состоит в том, что нужно помнить, что, как ни странно, текст не содержится непосредственно внутри элемента, он содержится в дочернем текстовом узле. То есть предполагается, что переменная `strongElem` содержит ссылку на элемент ``. В листинге 5.15 показано, как извлечь текст из элемента, используя DOM.

Листинг 5.15. Получение текстового содержимого элемента ``

```
// На браузерах, не связанных с движком Mozilla:
strongElem.innerHTML

// На всех платформах:
strongElem.firstChild.nodeValue
```

Теперь, когда мы узнали, как получить текстовое содержимое из отдельного элемента, нужно посмотреть, как получить объединенное текстовое содержимое элемента `<p>`. В процессе решения этой задачи вы можете также разработать универсальную функцию для получения текстового содержимого любого элемента, независимо от того, что в нем фактически содержится, что и показано в листинге 5.16. Вызов вида `text(элемент)` вернет строку, в которой содержится объединенное текстовое содержимое элемента и всех содержащихся в нем дочерних элементов.

Листинг 5.16. Универсальная функция для извлечения текстового содержимого из элемента

```
function text(e) {
    var t = "";

    // Если элемент был передан, получение его дочерних элементов
    // иначе, предположение о том, что передан массив
    e = e.childNodes || e;

    // Просмотр всех дочерних узлов
    for ( var j = 0; j < e.length; j++ ) {
        // Если это не элемент, присоединить его текстовое значение
        // Иначе, провести рекурсивный перебор всех дочерних составляющих
        // элемента
        t += e[j].nodeType != 1 ?
            e[j].nodeValue : text(e[j].childNodes);
    }
    // Возвращение соответствующего текста
    return t;
}
```

Имея функцию, которая может быть использована для получения текстового содержимого любого элемента, вы можете извлечь текстовое содержимое элемента `<p>`, использованного в предыдущем примере. Код для выполнения этой задачи выглядит следующим образом:

```
// Получение текстового содержимого элемента <p>
text( pElem );
```

Особенно приятно узнать, что эта функция гарантированно работает как с HTML, так и с XML DOM-документами, то есть теперь у вас есть совместимый способ извлечения текстового содержимого любого элемента.

Получение HTML, находящегося внутри элемента

В отличие от получения текста, находящегося внутри элемента, получения находящегося внутри него HTML — одна из самых простых из выполняемых DOM-задач. К счастью, благодаря свойству, разработанному

командой Internet Explorer, все современные браузеры теперь включают дополнительное свойство для каждого HTML DOM-элемента: `innerHTML`. Благодаря этому свойству вы можете получить весь HTML и текст, находящиеся внутри элемента. К тому же, свойство `innerHTML` работает очень быстро — зачастую намного быстрее, чем проведение рекурсивного поиска для обнаружения всего текстового содержимого элемента. Однако не все так радужно. Как именно реализовать свойство `innerHTML`, зависит от браузера, и, поскольку, единых стандартов для этого не существует, браузер может вернуть любой, соответствующий его понятиям контекст. Рассмотрим, к примеру, некоторые из тех, довольно странных ошибок, которые должны быть вами учтены при использовании свойства `innerHTML`:

- Браузеры, основанные на движке Mozilla, не возвращают при использовании свойства `innerHTML` элементов `<style>`.
- Internet Explorer возвращает свои элементы с использованием только заглавных букв, что при поиске какой-то последовательности может сорвать ваши планы.
- Свойство `innerHTML` постоянно доступно только как свойство элементов HTML DOM-документов; попытка использовать его в XML DOM-документах приводит к извлечению нулевых значений.

Использование свойства `innerHTML` не вызывает затруднений; обращение к свойству дает вам строку, в которой находится HTML-содержимое элемента. Если элемент не содержит каких-либо подэлементов, а содержит только текстовое наполнение, возвращаемая строка будет содержать только текст. Чтобы посмотреть, как это работает, мы изучим два элемента, показанные на рис. 5.3:

```
// Получение внутреннего HTML – innerHTML – элемента <strong>
// должно вернуть "Hello"
strongElem.innerHTML

// Получение внутреннего HTML – innerHTML – элемента <p>
// должно вернуть "<strong>Hello</strong> how are you doing?"
pElem.innerHTML
```

Если вы уверены в том, что элемент не содержит ничего, кроме текста, этот метод может служить как самый простой заменитель трудностям получения текста, содержащегося в элементе. С другой стороны, наличие возможности извлечения HTML-контента из элементов означает, что теперь вы можете создавать необычные динамические приложения, в которых применяется правка по месту. Более подробно разговор на эту тему будет вестись в главе 10.

Работа с атрибутами элементов

За извлечением содержимого элементов следующей наиболее часто совершаемой операцией является получение и установка значения атрибутов элементов. Как правило, список имеющихся у элемента атрибутов предварительно загружается с информацией, собранной из XML-представления самого элемента и сохраненной для последующего доступа в ассоциативном массиве, как в следующем примере фрагмента HTML, находящегося внутри веб-страницы:

```
<form name="myForm" action="/test.cgi" method="POST">
  ...
</form>
```

После загрузки в DOM и в переменную `formElem`, HTML элемент `form` будет иметь ассоциативный массив, из которого можно получить атрибуты в виде пар имя-значение. Результат этой работы выглядит следующим образом:


```
formElem.attributes = {
    name: "myForm",
    action: "/test.cgi",
    method: "POST"
};
```

Определение существования атрибутов элемента с использованием массива атрибутов должно быть абсолютно тривиальной задачей, но есть одна проблема: по какой-то причине Safari это не поддерживает. Но и это еще не все, потенциально полезная функция `hasAttribute` не поддерживается в Internet Explorer. Так как же все-таки определить наличие атрибутов? Один из возможных способов показан в листинге 5.17, и заключается в использовании функции `getAttribute` (речь о которой пойдет в следующем разделе) с проверкой, не вернула ли она нулевое значение.

Листинг 5.17. Определение наличия у элемента конкретного атрибута

```
function hasAttribute( elem, name ) {
    return elem.getAttribute(name) != null;
}
```

Теперь, имея в арсенале эту функцию, и зная, как используются атрибуты, вы готовы приступить к извлечению и установке значений атрибутов.

Получение и установка значений атрибута

Для извлечения принадлежащих атрибутам данных из элемента существуют два различных метода, применение которых зависит от типа используемого вами DOM-документа. Если вы хотите обезопасить себя от неожиданностей и всегда использовать универсальные, совместимые с XML DOM методы, то это — `getAttribute` и `setAttribute`. Они могут быть использованы следующим образом:

```
// Получение атрибута
id("everywhere").getAttribute("id")

// Установка значения атрибута
tag("input")[0].setAttribute("value", "Your Name");
```

В дополнение к этой стандартной паре `getAttribute-setAttribute`, HTML DOM-документы обладают специальным набором свойств, которые работают с атрибутами в качестве быстрых извлекателей-установщиков. Они доступны во всех современных реализациях DOM (но гарантированно работают лишь в HTML DOM-документах), поэтому их использование может дать вам большие преимущества в написании компактного кода. В следующем коде показывается, как можно использовать свойства DOM доступа к DOM-атрибутам и установки их значений:

```
// Быстрое получение атрибута
tag("input")[0].value

// Быстрая установка значения атрибута
tag("div")[0].id = "main";
```

При работе с атрибутами существует несколько странных случаев, о которых вы должны знать. Один из них чаще всего возникает при доступе к атрибуту имени класса. Чтобы работать с именами классов одинаково на

всех браузерах, нужно использовать для обращения к ним свойство `className`, используя выражение `elem.className`, и не использовать более соответствующее по названию выражение `getAttribute("class")`. Такая же проблема возникает и с атрибутом `for`, который был переименован в `htmlFor`. К тому же существуют проблемы и двумя CSS-атрибутами: `cssFloat` и `cssText`. Это своеобразное соглашение об именах возникло из-за того, что такие слова, как `class`, `for`, `float` и `text` являются в JavaScript зарезервированными словами.

Чтобы обойти все эти странные случаи и упростить весь процесс работы с получением и установкой нужных атрибутов, нужно воспользоваться функцией, которая возьмет для вас все эти заботы на себя. В листинге 5.18 показывается функция для получения и установки значений атрибутов элементов. Вызов этой функции с двумя параметрами, к примеру, `attr(element, id)`, приведет к возврату значения указанного атрибута. Если вызвать функцию с тремя параметрами, к примеру, `attr(element, class, test)`, то будет установлено значение атрибута и возвращено его новое значение.

Листинг 5.18. Получение и установка значений атрибутов элементов

```
function attr(elem, name, value) {
    // Гарантирование допустимости предоставленного имени
    if ( !name || name.constructor !== String ) return '';

    // Определение, не относится ли это имя к тем самым «роковым»
    // именам
    name = { 'for': 'htmlFor', 'class': 'className' }[name] || name;

    // Если пользователь устанавливает значение, то также
    if ( typeof value !== 'undefined' ) {
        // сначала установить быстрый способ
        elem[name] = value;

        // По возможности воспользоваться setAttribute
        if ( elem.setAttribute )
            elem.setAttribute(name, value);
    }

    // Вернуть значение атрибута
    return elem[name] || elem.getAttribute(name) || '';
}
```

Наличие стандартного способа для доступа к атрибутам и их изменения, независимо от их реализации, является довольно мощным инструментом. В листинге 5.19 показано несколько примеров использования функции `attr` в некоторых часто встречающихся ситуациях для упрощения работы с атрибутами.

Листинг 5.19. Использование функции `attr` для установки и извлечения значений атрибутов из DOM-элементов

```
// Установка атрибута class для первого <h1>-элемента
attr( tag("h1")[0], "class", "header" );

// Установка значения для каждого элемента <input>
var input = tag("input");
```

```

for ( var i = 0; i < input.length; i++ ) {
    attr( input[i], "value", "" );
}

// Добавление обрамления к элементу <input>, у которого атрибут name
// имеет значение 'invalid'
var input = tag("input");
for ( var i = 0; i < input.length; i++ ) {
    if ( attr( input[i], "name" ) == 'invalid' ) {
        input[i].style.border = "2px solid red";
    }
}

```

До сих пор я рассматривал обычно используемые в DOM атрибуты (например, ID, class, name и т.д.), значение которых можно получать и устанавливать. Но технология установки и получения нетрадиционных атрибутов была бы тоже очень полезна. К примеру, вы можете добавить новый атрибут (который будет виден только при доступе к DOM-версии элемента) а затем, чуть позже, снова его извлечь, и все это без модификации физических свойств документа. Предположим, к примеру, что вам нужен список элементов с их определениями, в котором по щелчку на элементе ему давалось бы развернутое определение. Код HTML для этой конструкции имел бы вид, показанный в листинге 5.20.

Листинг 5.20. HTML-документ со списком определений, скрытых от просмотра

```

<html>
<head>
    <title>Раскрываемый список определений</title>
    <style>dd { display: none; }</style>
</head>
<body>
    <h1>Раскрываемый список определений</h1>
    <dl>
        <dt>Коты</dt>
        <dd>Пушистые, дружелюбные создания.</dd>
        <dt>Собака</dt>
        <dd>Любят заигрывать и крутиться вокруг человека.</dd>
        <dt>Мыши</dt>
        <dd>Коты любят их есть.</dd>
    </dl>
</body>
</html>

```

Подробности, касающиеся событий, будут рассмотрены в главе 6, а пока я попытаюсь сохранить простоту нашего кода обработки событий. Далее следует небольшой сценарий, позволяющий щелкать на определяемых терминах и показывать (или скрывать) их определения. Этот сценарий должен быть включен в заголовок вашей страницы или в этот заголовок должен быть включен внешний файл с этим сценарием. В листинге 5.21 показан код, необходимый для построения раскрываемого списка определений.

Листинг 5.21. Сценарий, позволяющий осуществлять динамическое включение и выключение определений

```
// Ожидание готовности DOM к работе
domReady(function(){

    // Поиск всех определяемых терминов
    var dt = tag("dt");
    for ( var i = 0; i < dt.length; i++ ) {

        // Отслеживание щелчка пользователя на термине
        addEvent( dt[i], "click", function() {

            // Проверка, раскрыто определение или нет
            var open = attr( this, "open" );

            // Переключение отображения определения
            next( this ).style.display = open ? 'none' : 'block';

            // Запоминание того, что определение раскрыто
            attr( this, "open", open ? '' : 'yes' );

        });
    }
});
```

Теперь, когда вы знаете, как перемещаться по DOM, и как исследовать и изменять атрибуты, нужно изучить, как создавать новые DOM-элементы, вставлять их куда угодно, и удалять те элементы, которые уже не нужны.

Модификация DOM

Зная, как модифицировать DOM, вы можете делать все, от создания налету собственных XML-документов для построения динамических форм, которые приспособляются к пользовательскому вводу; вы обретаёте практически безграничные возможности. Модификация DOM постигается в три этапа: сначала нужно изучить, как создавать новый элемент, потом нужно изучить, как его вставлять в DOM, а затем нужно изучить как его оттуда извлечь.

Создание узлов с использованием DOM

Основной метод, стоящий за модификацией DOM, состоит в использовании функции `createElement`, которая даёт возможность создания нового элемента на лету. Но этот новый элемент при создании не вставляется немедленно в DOM (что часто ставит в тупик тех, кто только начинает работать с DOM). Сначала я сосредоточу внимание на создании DOM-элемента.

Метод `createElement` принимает один параметр, название тега элемента, и возвращает виртуальное DOM-представление этого элемента — без каких-либо включённых в него атрибутов или стилевых установок. Если вы разрабатываете приложение, использующее сгенерированные с помощью XSLT XHTML-страницы (или XHTML-страницы, снабжённые точным типом контента), то вам нужно помнить, что вы фактически используете XML-документ, и что ваши элементы нуждаются в наличии связанного с ними соответствующего пространства имен XML. Чтобы беспрепятственно обойти это обстоятельство, у вас должна быть простая функция, которая сможет спокойно протестировать, имеет ли используемый вами HTML DOM-документ возможность создания новых элементов с пространством имен (свойство, присущее XHTML DOM-документам). Если дело обстоит именно таким образом, вы можете создать новый DOM-элемент с соответствующим пространством имен XHTML, как это показано в листинге 5.22.

Листинг 5.22. Универсальная функция для создания нового DOM-элемента

```
function create( elem ) {
    return document.createElementNS ?
        document.createElementNS( 'http://www.w3.org/1999/xhtml', elem ) :
        document.createElement( elem );
}
```

Используя предыдущую функцию, вы можете, к примеру, создать простой <div>-элемент и присоединить к нему дополнительную информацию:

```
var div = create("div");
div.className = "items";
div.id = "all";
```

В дополнение нужно заметить, что существует DOM-метод, предназначенный для создания новых текстовых узлов, который называется `createTextNode`. Он принимает единственный аргумент, текст, который должен появиться внутри узла, и возвращает созданный текстовый узел.

Теперь, используя только что созданные DOM-элементы и текстовые узлы, вы можете вставить их в DOM-документы, именно туда, где они нужны.

Вставка в DOM

Вставка в DOM — дело весьма непростое, и временами вставит в тупик даже опытных пользователей DOM. Для его осуществления в арсенале разработчиков есть две функции.

Первая из них, `insertBefore`, позволяет вставлять элемент перед следующим дочерним элементом. При ее использовании это выглядит примерно следующим образом:

```
родительскийУзелПредыдущегоУзла.insertBefore( вставляемыйУзел,
                                                предыдущийУзел );
```

Мнемоническое правило, которое я использую, чтобы запомнить порядок следования аргументов выражается фразой: «Вставляем первый элемент перед вторым». Совсем скоро я покажу вам более простой способ, как это запомнить.

Теперь, имея в своем распоряжении функцию для вставки узлов (речь идет как об элементах, так и о текстовых узлах) перед другими узлами, вы можете задаться вопросом: «А как же вставить узел в качестве последнего дочернего узла?». Для этого существует другая функция под названием `appendChild`, позволяющая выполнить эту задачу. Функция `appendChild` вызывается для элемента, добавляемого к определенному узлу в самый конец списка его дочерних узлов. Использование этой функции выглядит следующим образом:

```
родительскийЭлемент.appendChild( вставляемыйУзел );
```

Чтобы избавиться от необходимости вспоминать конкретный порядок аргументов в функциях `insertBefore` и `appendChild`, можно воспользоваться двумя вспомогательными функциями, которые были созданы мной для решения этой проблемы. Использование новых функций показано в листингах 5.23 и 5.24, в них порядок вызова аргументов дается относительно того узла (элемента), в который производится вставка, после чего указывается вставляемый элемент (узел). Дополнительно функция `before` позволяет предоставить необязательный родительский элемент, что потенциально приводит к сокращению кода. И наконец, обе эти функции позволяют передавать им срок для вставки (дополнения), которая будет автоматически превращена для вас в текстовый

узел. Рекомендуется, чтобы родительский элемент передавался в виде ссылки (на тот случай, если он имеет значение null).

Листинг 5.23. Функция для вставки элемента перед другим элементом

```
function before( parent, before, elem ) {
    // Выяснение, предоставлен ли родительский (parent) узел
    if ( elem == null ) {
        elem = before;
        before = parent;
        parent = before.parentNode;
    }
    parent.insertBefore( checkElem( elem ), before );
}
```

Листинг 5.24. Функция добавления элемента в качестве дочернего к другому элементу

```
function append( parent, elem ) {
    parent.appendChild( checkElem( elem ) );
}
```

Вспомогательная функция, показанная в листинге 5.25 облегчает вам вставку как элемента, так и текста (который автоматически превращается в нормальный текстовый узел).

Листинг 5.25. Вспомогательная функция для функций before и append

```
function checkElem( elem ) {
    // Если предоставлена только строка, превращение ее в текстовый узел
    return elem && elem.constructor == String ?
        document.createTextNode( elem ) : elem;
}
```

Теперь, используя функции before и append, и создавая новые DOM-элементы, вы можете пополнять DOM информацией, доступной для пользовательского просмотра (см. листинг 5.26).

Листинг 5.26. Использование функций append и before

```
// Создание нового <li>-элемента
var li = create("li");
attr( li, "class", "new" );

// Создание текстового содержимого и добавление его к <li>
append( li, "Спасибо за то, что вы посетили наш сайт!" );

// Добавление <li> в верхнюю строку первого упорядоченного списка
before( first( tag("ol")[0] ), li );

// Запуск этого оператора превратит пустой элемент <ol>
```

```
<ol></ol>
```

```
// в следующий:
```

```
<ol>
  <li class='new'>Спасибо за то, что вы посетили наш сайт!</li>
</ol>
```

Как только вы вставите эту информацию в DOM (используя либо `insertBefore`, либо `appendChild`) она будет тот час же выведена на экран на обозрение пользователя. Поэтому вы можете воспользоваться этим обстоятельством для предоставления мгновенного отклика на его действия. Это особенно полезно для интерактивных приложений, в которых требуется пользовательский ввод.

Теперь, посмотрев как создавать и вставлять узлы с применением исключительно DOM-методов, будет особенно полезно посмотреть на альтернативные методы вставки в DOM какого-нибудь содержимого.

Вставка в DOM кода HTML

Технология непосредственной вставки HTML в документ имеет даже большую популярность, чем создание и вставка в DOM обычных DOM-элементов. Проще всего для этого воспользоваться упоминавшимся ранее методом `innerHTML`. Вдобавок к тому, что он является способом извлечения HTML, находящегося внутри элемента, он также предоставляет способ установки HTML внутри элемента. В качестве примера простоты его использования давайте представим, что у нас есть пустой элемент ``, к которому нужно добавить несколько элементов ``. Код, выполняющий это действие, может выглядеть следующим образом:

```
// Добавление нескольких LI к OL-элементу
tag("ol")[0].innerHTML = "<li>Коты.</li><li>Собаки.</li><li>Мыши.</li>";
```

Правда же, это намного проще, чем упорно создавать несколько DOM-элементов и связанные с ними текстовые узлы? Вас может также обрадовать известие (соответствующее информации на <http://www.quirksmode.org>) что этот способ еще и быстрее работает, чем использование методов DOM. Но, не все так безоблачно — еще существует ряд коварных проблем, возникающих при использовании метода вставки `innerHTML`:

- Как уже ранее упоминалось, метод `innerHTML` отсутствует XML DOM-документах, значит, в этих документах вам придется и дальше пользоваться традиционными DOM-методами создания узлов и элементов.
- XHTML-документы, созданные с использованием XSLT, который находится на стороне клиента, не имеют метода `innerHTML`, поскольку они тоже относятся к XML-документам.
- Метод `innerHTML` полностью удаляет любые узлы, которые уже существуют внутри элемента, значит, способ последовательного добавления или вставки впереди существующего содержимого, существующий в DOM-методах, в нем отсутствует.

Последний пункт особенно огорчает, поскольку вставка впереди или добавление к концу дочернего списка другого элемента — весьма полезное свойство. И все-таки, применив магию DOM, вы можете приспособить наши методы `append` и `before` для работы с обычными HTML-строками в дополнение к работе с обычными DOM-элементами. Перемены осуществляются в два этапа. Сначала создается новая функция `checkElem`, показанная в листинге 5.27, которая способна обрабатывать HTML-строки, DOM-элементы и массивы DOM-элементов.

Листинг 5.27. Преобразование массива, представляющего смесь из DOM-узлов и HTML-строк в настоящий массив DOM-узлов

```
function checkElem(a) {
  var r = [];
```

```

// Превращение аргумента в массив, если он еще им не является
if ( a.constructor !== Array ) a = [ a ];

for ( var i = 0; i < a.length; i++ ) {
    // Если это строка
    if ( a[i].constructor === String ) {
        // Создание временного элемента для помещения в него HTML
        var div = document.createElement("div");

        // Вставка HTML, для превращения его в DOM-структуру
        div.innerHTML = a[i];

        // Обратное извлечение DOM-структуры из временного DIV-элемента
        for ( var j = 0; j < div.childNodes.length; j++ )
            r[r.length] = div.childNodes[j];
    } else if ( a[i].length ) { // Если это массив
        // Предположение, что это массив DOM-узлов
        for ( var j = 0; j < a[i].length; j++ )
            r[r.length] = a[i][j];
    } else { // Иначе, предположение, что это DOM-узел
        r[r.length] = a[i];
    }
}
return r;
}

```

Затем, как показано в листинге 5.28, нужно приспособить две функции вставки к работе с этой модификацией `checkElem`.

Листинг 5.28. Усовершенствованные функции для вставки и добавления в DOM

```

function before( parent, before, elem ) {
    // Выяснение, предоставлен ли родительский (parent) узел
    if ( elem === null ) {
        elem = before;
        before = parent;
        parent = before.parentNode;
    }

    // Получение нового массива элементов
    var elems = checkElem( elem );

    // Обратный перебор элементов массива,
    // поскольку мы добавляем элементы к началу
    for ( var i = elems.length - 1; i >= 0; i-- ) {
        parent.insertBefore( elems[i], before );
    }
}

```



```
function append( parent, elem ) {
    // Получение массива элементов
    var elems = checkElem( elem );

    // Добавление всех элементов к родительскому элементу
    for ( var i = 0; i <= elems.length; i++ ) {
        parent.appendChild( elems[i] );
    }
}
```

Теперь, с использованием этих новых функций, добавление `` в упорядоченный список становится невероятно простой задачей:

```
append( tag("ol")[0], "<li>Мышеловка.</li>" );

// Выполнение этой простой строки может добавить дополнительный HTML в этот <ol>-список
<ol>
    <li>Коты.</li>
    <li>Собаки.</li>
    <li>Мыши.</li>
</ol>
```

// превращая его в следующий список:

```
<ol>
    <li>Коты.</li>
    <li>Собаки.</li>
    <li>Мыши.</li>
    <li>Мышеловки.</li>
</ol>
```

// А выполнение простого оператора, использующего функцию `before`

```
before( last( tag("ol")[0] ), "<li>Зебры.</li>" );
```

// Превратит `` в:

```
<ol>
    <li>Коты.</li>
    <li>Собаки.</li>
    <li>Зебры.</li>
    <li>Мыши.</li>
</ol>
```

Благодаря применению этой технологии ваш код становится намного компактнее и пригоднее к разработке. А если вам захочется развернуться в обратную сторону, и удалить узлы из DOM? Как всегда, для этого найдется другой метод.

Удаление узлов из DOM

Удаление узлов из DOM случается едва ли не чаще, чем их создание и вставка. К примеру, при создании динамической формы, запрашивающей ввод неограниченного количества элементов, становится важным

разрешить пользователям удалять ту часть страницы, с которой они больше не желают работать. Возможность удаления узла заложена в одну функцию: `removeChild`. Она используется точно так же, как и функция `appendChild`, но имеет обратный эффект. Ее использование выглядит примерно так:

```
РодительскийУзел.removeChild( УдаляемыйУзел );
```

С расчетом на нее, вы можете создать две отдельные функции для быстрого удаления узлов. Одна из них показана в листинге 5.29.

Листинг 5.29. Функция для удаления узла из DOM

```
// Удаление из DOM отдельного узла
function remove( elem ) {
    if ( elem ) elem.parentNode.removeChild( elem );
}
```

В листинге 5.30 показана функция для удаления из элемента всех дочерних узлов, в которой используется только одна ссылка на DOM-элемент.

Листинг 5.30. Функция для удаление из элемента всех дочерних узлов

```
// Удаление из DOM всех дочерних узлов элемента
function empty( elem ) {
    while ( elem.firstChild )
        remove( elem.firstChild );
}
```

Представим, к примеру, что вам нужно удалить тот элемент ``, который вы добавили в предыдущем разделе, предполагая, что вы дали пользователю возможность как следует рассмотреть этот ``, и он может быть удален без всяких последствий. Следующий пример показывает, что для получения желаемого результата вы можете воспользоваться кодом JavaScript:

```
// Удаление последнего <li> из <ol>
remove( last( tag("ol")[0] ) )

// Приведенный выше код превратит этот фрагмент:

<ol>
    <li>Учите Javascript.</li>
    <li>???.</li>
    <li>Получите прибыль!</li>
</ol>

// в этот:

<ol>
    <li>Учите Javascript.</li>
    <li>???.</li>
</ol>

// Если бы мы запустили вместо функции remove() функцию empty()
empty( last( tag("ol")[0] ) )
```

```
// То остался бы просто пустой список <ol>:  
<ol></ol>
```

Изучив возможность удаления узла из DOM, вы завершили освоение урока о том, как работает объектная модель документа, и как из этого можно извлечь для себя наибольшую выгоду.

Вывод

В этой главе я рассмотрел многие вопросы, относящиеся к объектной модели документа. К сожалению, некоторые темы, к примеру, ожидание загрузки DOM, оказались сложнее других, и их рассмотрение будет продолжено в обозримом будущем. Тем не менее, использование всего здесь изученного, позволит вам создавать практически любые динамические компоненты веб-приложений.

Если вы желаете изучить некоторые примеры DOM-сценариев в действии, просмотрите приложение A, в которое именно для этого включено множество дополнительного кода. Дополнительно, еще больше примеров создания DOM-сценариев можно найти в Интернете, на веб-сайте книги по адресу: <http://jspro.org>, или в разделе исходного кода и загрузки — Source Code/Download на веб-сайте издательства Apress: <http://www.apress.com>. Далее я собираюсь привлечь ваше внимание к следующим компонентам ненавязчивого создания DOM-сценариев: к событиям.

Глава 6 События

Наиболее важной стороной создания ненавязчивых DOM-сценариев является использование динамически связанных *событий*. Конечная цель написания полезного кода JavaScript состоит в получении веб-страницы, работающей для пользователей, независимо от того, какой браузер они используют, или на какой платформе работают. Для достижения этой цели вы устанавливаете заданный набор свойств, которыми хотите воспользоваться, и исключаете все браузеры, которые их не поддерживают. Для браузеров, не поддерживающих эти свойства, вы предлагаете работоспособные, но менее интерактивные версии веб-сайта. Преимущества от такой организации взаимодействия JavaScript и HTML включают более совершенный код, более понятные веб-страницы, и лучшее взаимодействие с пользователем. Всего этого можно достичь за счет использования событий DOM для улучшения взаимодействия, происходящего в веб-приложениях.

С годами понятие событий в JavaScript получало развитие, приближаясь к тому надежному, но полупригодному состоянию, в котором оно сейчас и находится. К счастью, благодаря существованию общих черт, вы можете разрабатывать замечательные инструменты, помогающие создавать мощные, безупречно написанные веб-приложения.

В начале этой главы я собираюсь представить вам работу событий в JavaScript и сравнить ее с моделями событий, существующими в других языках программирования. Затем я собираюсь рассмотреть информацию, предоставляемую моделью событий, и лучшие способы ее контролирования. После того как будет рассмотрена привязка событий к DOM-элементам и различные виды доступных событий, в заключение я покажу, как интегрировать некоторые эффективные и ненавязчивые технологии написания сценариев в любую веб-страницу.

Введение в события JavaScript

Если вы заглянете в основу любого кода JavaScript, то увидите, что события являются именно тем связующим элементом, который все и скрепляет. В хорошо спроектированном JavaScript-приложении, вы стремитесь иметь источник данных и его визуальное представление (в HTML DOM). Чтобы синхронизировать эти две стороны приложения, вам необходимо отслеживать все действия пользователя и, следовательно, его попытки обновить информацию на вашем веб-сайте. Сочетание использования событий DOM и JavaScript является основным союзом, определяющим облик современных веб-приложений.

Асинхронные события против потоков

В JavaScript используется весьма уникальная система событий. Она работает абсолютно асинхронно, вообще не используя потоков. Это означает, что весь код вашего приложения будет зависеть от каких-либо действий, например, от щелчка пользователя или загрузки страницы — что будет приводить к выполнению определенного кода.

Основное отличие программ, спроектированных для работы с потоками от программ, спроектированных для работы с асинхронными событиями состоит в том, как происходит ожидание происходящего. В программах, рассчитанных на потоки, вы постоянно ведете проверку, пока не будут выполнены заданные вами условия. Тогда как в асинхронных программах вы просто регистрируете в качестве обработчика события функцию обратного вызова, а затем, когда наступит событие, обработчик даст вам знать, выполняя вашу функцию обратного вызова.

Теперь посмотрим, как пишется программа JavaScript, если используются потоки, и как она пишется, если используются асинхронные обратные вызовы.

Потоки JavaScript

На сегодняшний день в JavaScript потоки не существуют. Самое близкое, что может быть использовано в этом качестве — функция обратного вызова `setTimeout()`, но даже при этом модель будет далека от идеала. Если

JavaScript был бы традиционным, работающим с потоками языком программирования, то что-либо подобное коду, приведенному в листинге 6.1 могло бы сработать. В данной имитации кода ведется ожидание, пока страница не будет полностью загружена. Если бы JavaScript был потоковым языком программирования, то вам бы пришлось делать что-либо подобное. Хорошо, что до этого дело не доходит.

Листинг 6.1. Ложный код JavaScript для имитации потока

```
// ПРИМЕЧАНИЕ: Этот код не работает!
// Ожидание, пока страница не загрузится, путем осуществления постоянных
// проверок
while ( ! window.loaded() ) { }

// Страница загружена, можно приступать к работе
document.getElementById("body").style.border = "1px solid #000";
```

Если заметили, в этом коде используется цикл, в котором осуществляется постоянная проверка, какое значение возвращает `window.loaded()`, истинное или нет. Невзирая на факт отсутствия у объекта `windows` метода `loaded()`, давайте посмотрим, почему это не работает в JavaScript. Это происходит из-за того, что все циклы в JavaScript являются блокирующими (то есть пока они не закончат работу, ничего другого произойти не может). Если бы JavaScript был в состоянии обрабатывать потоки, то вы увидели бы что-либо, подобное изображенному на рис. 6.1. На этом рисунке имеющийся в вашем коде цикл `while` постоянно проверяет, не загрузилось ли окно. Но в JavaScript это не работает из-за того, что все циклы в нем — блокирующие (поэтому никакие другие операции не могут быть выполнены, пока цикл работает).

`while (! window.loaded()) { }`

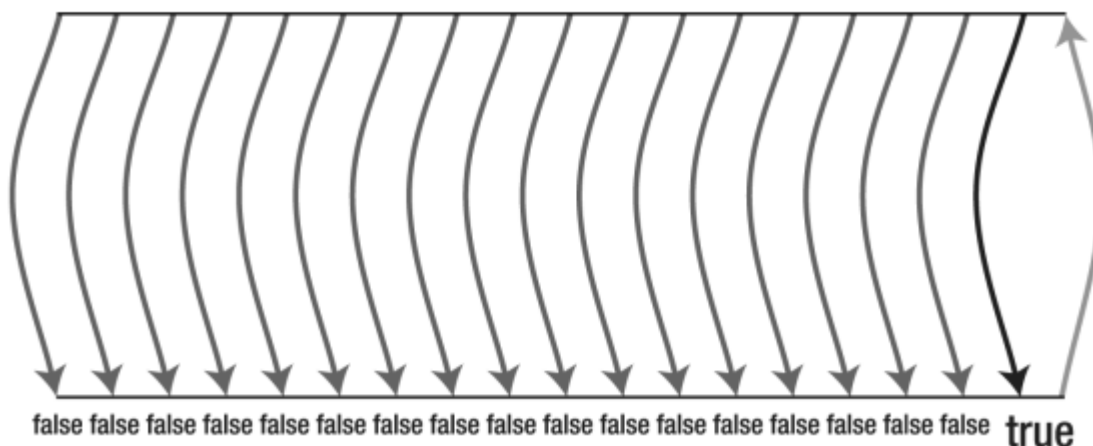


Рис. 6.1. Вот что бы получилось, если бы JavaScript умел обрабатывать потоки

На самом деле, поскольку наш цикл `while` продолжает работать и блокировать нормальный ход выполнения приложения, то он никогда не получит истинное значение. В результате пользовательский браузер остановится и зависнет, что может привести к аварийному завершению работы. Из этого можно извлечь следующий урок: если кто-нибудь будет утверждать, что использует цикл `while` для ожидания (в JavaScript) окончания работающего процесса, то он либо лжет, либо сильно заблуждается.

Асинхронные обратные вызовы

Программной альтернативой использованию потоков для постоянной проверки обновлений является использование асинхронных обратных вызовов, что, собственно, и делается в JavaScript. Используя простую терминологию, вы сообщаете DOM-элементу, что при каждом возникновении события определенного вида, вы

хотите, чтобы для его обработки была вызвана функция. Это означает, что вы можете предоставить ссылку на код, который желательно в нужный момент выполнить, а браузер берет на себя всю заботу о деталях. В листинге 6.2. показан пример кода, в котором используется обработчик событий и обратный вызов. В нем вы увидите действующий код, необходимый для привязки функции к обработчику события (`window.onload`) в JavaScript. Обработчик `window.onload()` будет вызван при каждом событии окончания загрузки страницы. Он также подойдет и для других общих событий, к примеру, для щелчка, перемещения мыши и щелчка на кнопке типа `submit`.

Листинг 6.2. Асинхронный обратный вызов в JavaScript

```
// Регистрация функции для вызова при каждой загрузке страницы
window.onload = loaded;

// Функция, вызываемая при каждой загрузке страницы.
function loaded() {
    // Страница уже загружена, можно приступить к работе
    document.getElementById("body").style.border = "1px solid #000";
}
```

Если сравнить код в листинге 6.2 с кодом, показанным в листинге 6.1, будут видны явные различия. Немедленно выполняется только тот код, который привязан к обработчику события (функции `loaded`), к его перехватчику (свойству `onload`). Как только страница будет полностью загружена, браузер вызывает функцию, связанную с `window.onload`, и выполняет ее. Примерный ход работы кода JavaScript показан на рис. 6.2. На рисунке изображено представление об использовании функции обратного вызова для ожидания в JavaScript загрузки страницы. Поскольку ожидание здесь в принципе невозможно, вы регистрируете обратный вызов (`loaded`), который состоится при полной загрузке страницы, за обработчиком (`window.onload`).

Есть одно, не слишком очевидное обстоятельство, касающееся нашего простого перехватчика и обработчика, которое связано тем, что порядок событий может изменяться, и обработка может отличаться в зависимости от вида события и расположения элемента в структуре DOM. В следующем разделе мы рассмотрим две различные фазы событий, и причины, которые вызывают эту разницу.

window.onload = loaded;



Рис. 6.2. Представление об использовании обратных вызовов для ожидания загрузки страницы

Фазы события

В JavaScript события происходят в двух фазах, которые называются захватом (capturing) и всплытием, подобно пузырькам (bubbling). Это означает следующее: если событие происходит в отношении элемента (к примеру, пользователь щелкает на ссылке, вызывая тем самым событие щелчка), элементы, позволяющие его обработать, и порядок, в котором происходит обработка, варьируются. Порядок выполнения можно увидеть в примере на рис. 6.3. На этом рисунке показано, какие обработчики события задействуются и в каком порядке при каждом щелчке пользователя на первом <a>-элементе страницы.

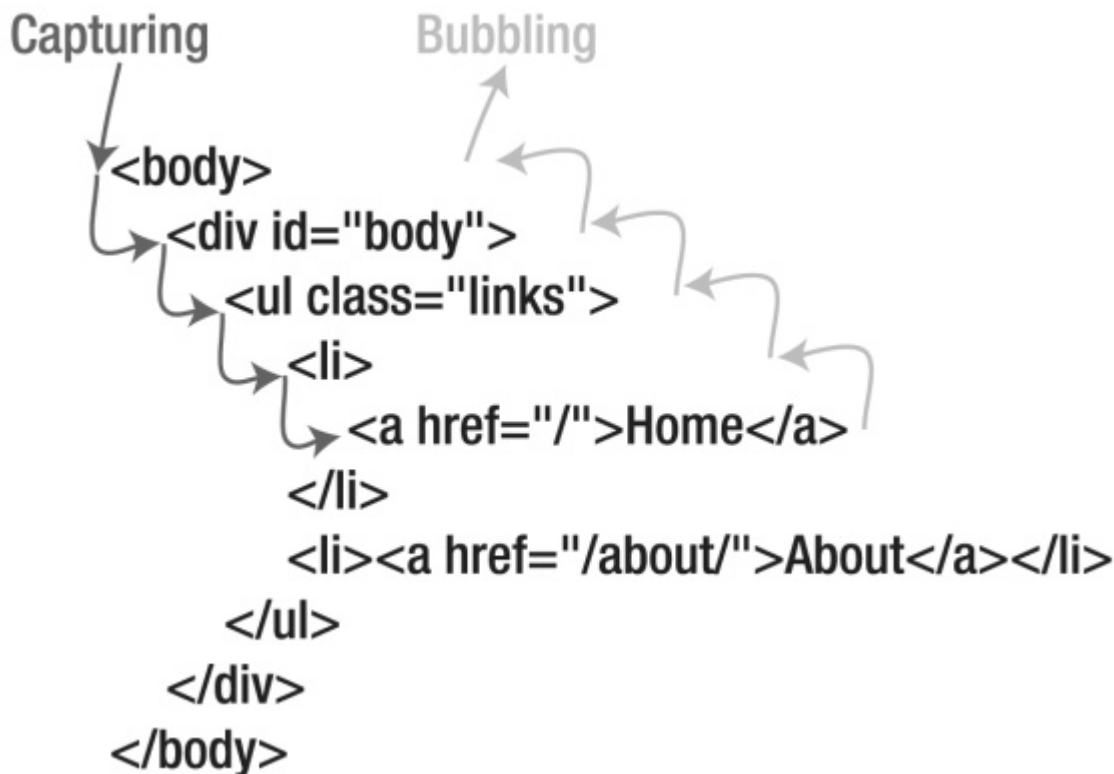


Рис. 6.3. Две фазы обработки события

Если посмотреть на простой пример щелчка на ссылке (на рис. 6.3), можно увидеть порядок выполнения обработки события. Под тем предлогом, что пользователь щелкнул на элементе <a>, сначала срабатывает обработчик щелчка, принадлежащий элементу document, затем обработчик, принадлежащий элементу <body>, затем обработчик, принадлежащий элементу <div>, и так далее, вплоть до элемента <a>; это называется фазой захвата. Как только это все завершится, процесс идет вспять, вверх по дереву, и срабатывают по порядку обработчики событий , , <div>, <body> и document.

Существуют вполне конкретные причины, по которым обработка события построена именно таким образом, и почему все это превосходно работает. Рассмотрим простой пример. Предположим, вам нужно изменять цвет фона каждого -элемента при каждом прохождении над ним указателя мыши, и возвращать исходный цвет, когда указатель мыши уходит за пределы элемента — обычные запросы для многих систем меню. Именно эту задачу и выполняет код, показанный в листинге 6.3.

Листинг 6.3. Сценарий обозначаемых перемещений с эффектами зависания

```
// Обнаружение всех элементов <li> для подключения к ним обработчиков
// событий
var li = document.getElementsByTagName("li");
for ( var i = 0; i < li.length; i++ ) {

    // Подключение к элементу <li> обработчика события mouseover,
```

```

// который изменяет цвет фона <li> на синий.
li[i].onmouseover = function() {
    this.style.backgroundColor = 'blue';
};

// Подключение к элементу <li> обработчика события,
// который изменяет цвет фона <li> на исходный белый
li[i].onmouseout = function() {
    this.style.backgroundColor = 'white';
};
}

```

Код ведет себя точно в соответствии с вашим представлением: при проходе указателя вашей мыши над элементом ``, его фоновый цвет изменяется; при выходе указателя за пределы элемента, цвет возвращается к исходному. Но при этом вы не понимаете, что при каждом прохождении указателя над ``, вы на самом деле периодически переключаетесь между двумя различными элементами. Поскольку элемент `` также содержит элемент `<a>`, вы перемещаете указатель мыши и над ним, а не только над ``. Взглянем на точный ход возникновения событий:

- ` mouseover`: Указатель мыши находится над элементом ``.
- ` mouseout`: Вы перемещаете его с `` на `<a>`, который находится внутри.
- `<a> mouseover`: Теперь указатель мыши находится над элементом `<a>`.
- ` mouseover`: Событие `mouseover`, принадлежащее `<a>`, «всплывает» вверх к событию `mouseover`, принадлежащему элементу ``.

Если вы по направлению возникновения событий заметили, что фаза захвата полностью проигнорирована, не переживайте, я о ней не забыл. Способ привязки перехватчиков событий заключается в использовании старых «традиционных» средств привязки событий за счет установки свойства элемента `onevent`, которое поддерживает только всплытие, а не захват события. Этот способ привязки событий наряду с другими рассмотрен в следующем разделе.

Вдобавок к странному порядку возникновения событий вы могли заметить два неожиданных действия: возникновение `mouseout` при выходе из `` и всплытие `mouseover` от `<a>` к ``. Рассмотрим это более подробно.

Первое событие `mouseout` возникает из-за того, что с точки зрения браузера вы покинули пределы родительского элемента `` и переместились на другой элемент. Это происходит из-за того, что элемент, находящийся поверх других элементов (как элемент `<a>` по отношению к родительскому элементу ``) моментально получает фокус указателя мыши.

Принадлежащее элементу `<a>` событие `mouseover` всплывает к родительскому элементу ``, бросая тем самым спасательный круг нашему фрагменту кода. Поскольку вы не привязали к элементу `<a>` никакого перехватчика, событие просто перемещается вверх по дереву DOM в поиске другого элемента, имеющего перехватчик. И первым попавшимся элементом в этом процессе всплытия оказывается ``, который перехватывает входящие события `mouseover` (что, собственно, и требовалось).

Но тут возникает другой вопрос, что произойдет, если у элемента <a> будет собственный обработчик события `mouseover`? Существует ли какой-нибудь способ, который сможет остановить всплытие события? Эта важная и полезная тема будет рассмотрена в следующем разделе.

Общие свойства событий

Одной из сильных сторон событий JavaScript является наличие относительно совместимых свойств, которые дают в процессе разработки больше возможностей и средств управления. Самым простым и давним понятием является объект события, предоставляющий собой набор метаданных и контекстно-зависимых функций, позволяющих вам работать, к примеру, с событиями мыши и клавиатуры. Кроме этого существуют функции, которые можно использовать для изменения нормального хода захвата-всплытия события. Изучение этих свойств в полном объеме может значительно упростить вашу жизнь.

Объект события

Стандартным свойством обработчиков событий является способ обращений к объекту события, в котором содержится контекстно-зависимая информация о текущем событии. Для определенных событий этот объект служит очень ценным ресурсом. К примеру, при обработке нажатий клавиш можно получить доступ к свойству объекта `keyCode` и получить код нажатой клавиши. Более подробная информация, относящаяся к специфике объекта события, изложена в приложении Б.

У объекта события есть одна сложность: его реализация в Internet Explorer отличается от спецификации, предложенной W3C. В Internet Explorer имеется один глобальный объект события (который может быть гарантированно найден в глобальной переменной свойства `window.event`), тогда как в других браузерах имеется переданный им отдельный параметр, в котором содержится объект события. Пример гарантированного использования объекта события показан в листинге 6.4. Это пример изменения типичного поведения элемента `<textarea>`. Обычно пользователи могут находясь в `textarea` нажать клавишу `Enter`, и вызвать появление дополнительного символа конца строки. А вместо этого нужно простое расширение текстового окна? Именно эту задачу и выполняет следующая функция.

Листинг 6.4. Подмена выполняемой функции путем использования событий DOM

```
// Обнаружение на странице первого элемента <textarea>, и привязка к нему
// перехватчика нажатия клавиатуры
document.getElementsByTagName("textarea")[0].onkeypress = function(e) {
    // Если объект события отсутствует, использование глобальной
    // переменной (только для IE)
    e = e || window.event;

    // Если нажата клавиша Enter, вернуть false (то есть ничего не делать)
    return e.keyCode != 13;
};
```

В объекте события содержится множество свойств и методов, их имена и поведение варьируются от браузера к браузеру. Сейчас я не хочу вдаваться в подробности, но настоятельно рекомендую прочесть приложение Б, в котором приведен большой список всех свойств объекта события, рассказано, как их использовать и приведены примеры использования.

Ключевое слово `this`

Ключевое слово `this` (как уже было рассмотрено в главе 2) служит способом обращения к текущему объекту внутри функции. При использовании ключевого слова `this` современные браузеры дают всем

обработчикам событий некоторый контекст. Но только часть этого контекста (и только в некоторых методах) работает должным образом и относится к текущему элементу; этот вопрос мы рассмотрим поглубже буквально через минуту. К примеру, в листинге 6.5, я могу воспользоваться этим обстоятельством только для того, чтобы создать одну универсальную функцию обработки щелчков, но не смогу использовать это ключевое слово, чтобы определить, какой элемент в данный момент обрабатывается. В листинге показан пример использования только одной функции для обработки события щелчка, но так как ключевое слово `this` в ней используется для ссылки на элемент, все будет работать должным образом.

Листинг 6.5. Изменение цвета фона и переднего плана всех элементов `` по щелчку

```
// Обнаружение всех элементов <li> и привязка к каждому из них
// обработчика щелчка
var li = document.getElementsByTagName("li");
for ( var i = 0; i < li.length; i++ ) {
    li[i].onclick = handleClick;
}

// Обработчик щелчка при вызове изменяет цвет фона
// и цвет переднего плана определенного элемента
function handleClick() {
    this.style.backgroundColor = "blue";
    this.style.color = "white";
}
```

Фактически ключевое слово `this` предоставляет лишь некоторое удобство, однако, я полагаю, что вы согласитесь с тем, что при правильном использовании оно может существенно упростить код JavaScript. Весь код, который связан в этой книге с событиями, я старался написать, используя это ключевое слово.

Прекращение всплытия событий

Теперь, когда вы уже знаете как работают захват и всплытие событий, давайте исследуем вопросы управления этими процессами. В одном из предыдущих примеров был поднят важный вопрос: если вам нужно будет, чтобы событие произошло только для заданного элемента и не касалось его родительских элементов, то способов остановки процесса у нас пока нет. Остановка всплытия события ситуацию, изображенную на рис. 6.4, в которой показан результат захвата события первым элементом `<a>` и прекращения последующего всплытия.

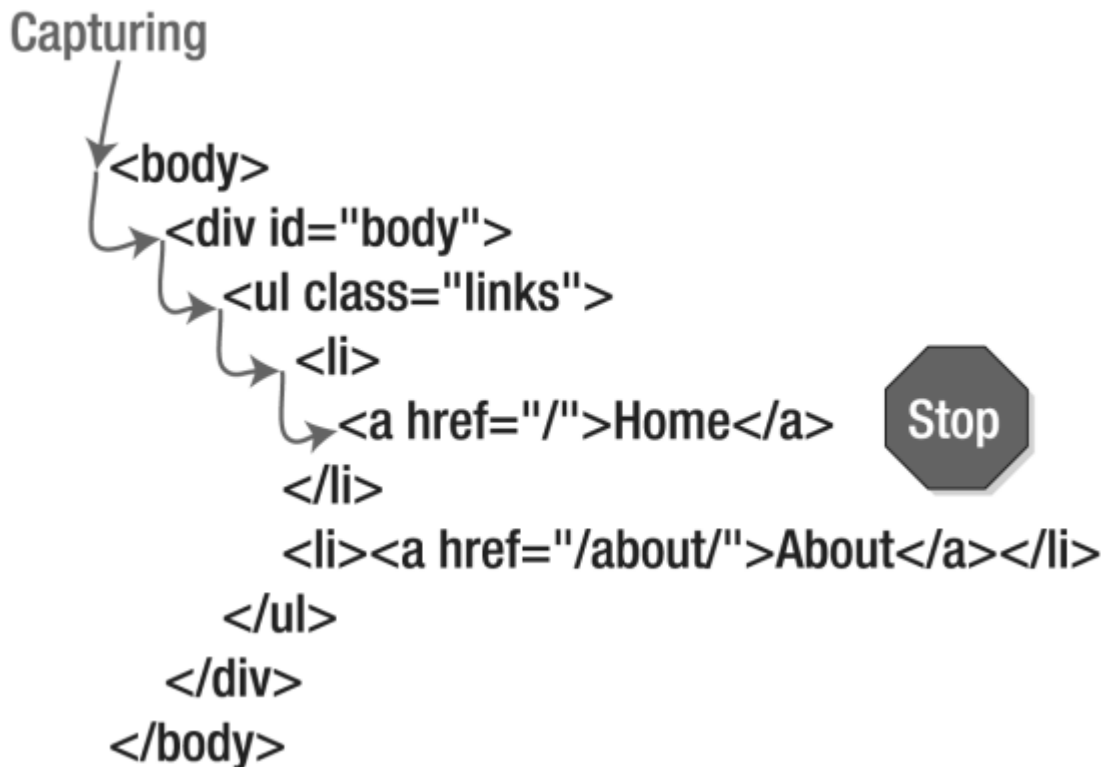


Рис. 6.4. Результат захвата события первым элементом <a>

Остановка всплытия (или захвата) события может стать исключительно полезной в сложных приложениях. К сожалению, Internet Explorer предлагает иной способ остановки всплытия события, чем все остальные браузеры. Универсальная функция прекращения всплытия события показана в листинге 6.6. Она принимает единственный аргумент: объект события, переданный в обработчик события. Функция обрабатывает два разных способа прекращения всплытия события: стандартный W3C-способ, и нестандартный способ, принадлежащий Internet Explorer.

Листинг 6.6. Универсальная функция для остановки всплытия события

```

function stopBubble(e) {
  // Если предоставлен объект события, значит это не IE-браузер
  if ( e && e.stopPropagation )
    // и он поддерживает W3C-метод stopPropagation()
    e.stopPropagation();
  else
    // В противном случае нужно воспользоваться способом
    // прекращения всплытия события, существующим в Internet Explorer
    window.event.cancelBubble = true;
}

```

Теперь вы, наверное, хотите спросить, когда я хочу остановить всплытие события? По правде говоря, в большинстве случаев вам не придется об этом волноваться. Потребности в этом возникнут, когда вы начнете разрабатывать динамические приложения (особенно те, которые работают с клавиатурой или мышью). В листинге 6.7 показан небольшой фрагмент, который добавляет красное обрамление вокруг текущего элемента, на котором вы проносите указатель мыши. Это достигается добавлением обработчиков событий `mouseover` и `mouseout` к каждому DOM-элементу. Если вы не остановите всплытие события, то при каждом проходе указателя мыши над элементом красное обрамление будет получать как сам элемент, так и все его родительские элементы, что не соответствует вашему желанию.

Листинг 6.7. Использование stopBubble() для создания интерактивного набора элементов

```
// Обнаружение и проход по всем элементам, имеющимся в DOM
var all = document.getElementsByTagName("*");
for ( var i = 0; i < all.length; i++ ) {

    // Отслеживание прохода указателя мыши над элементом
    // и добавление к элементу красного обрамления
    all[i].onmouseover = function(e) {
        this.style.border = "1px solid red";
        stopBubble( e );
    };

    // Отслеживание выхода указателя мыши за пределы элемента
    // и удаление ранее добавленного обрамления
    all[i].onmouseout = function(e) {
        this.style.border = "0px";
        stopBubble( e );
    };
}
}
```

Теперь, имея возможность остановить всплытие события, вы получаете полное управление над тем, какой именно элемент занимается отслеживанием и обработкой события. Это основной инструмент исследования разработки динамических веб-приложений. Теперь остается только отключить исходные действия браузера, что позволит вам подменять все, что он делает и реализовать вместо этого новые функциональные возможности.

Подмена исходных действий браузера

Для большинства имеющихся событий у браузера есть некоторые исходные действия, которые всегда им совершаются. К примеру, щелчок на элементе <a> перенесет вас на связанную с ним веб-страницу; это исходное действие браузера. На рис. 6.5 показано, что оно всегда будет совершаться после обеих фаз захвата и всплытия. В этом примере показаны результаты пользовательского щелчка на имеющимся на веб-странице элементе <a>. Событие, как уже ранее говорилось, начинается с путешествия по DOM в обеих фазах: захвата и всплытия. Но только как элемент будет пройден, браузер попытается выполнить свое исходное действие для этого события и элемента. В данном случае, посетить страницу, обозначенную как /.

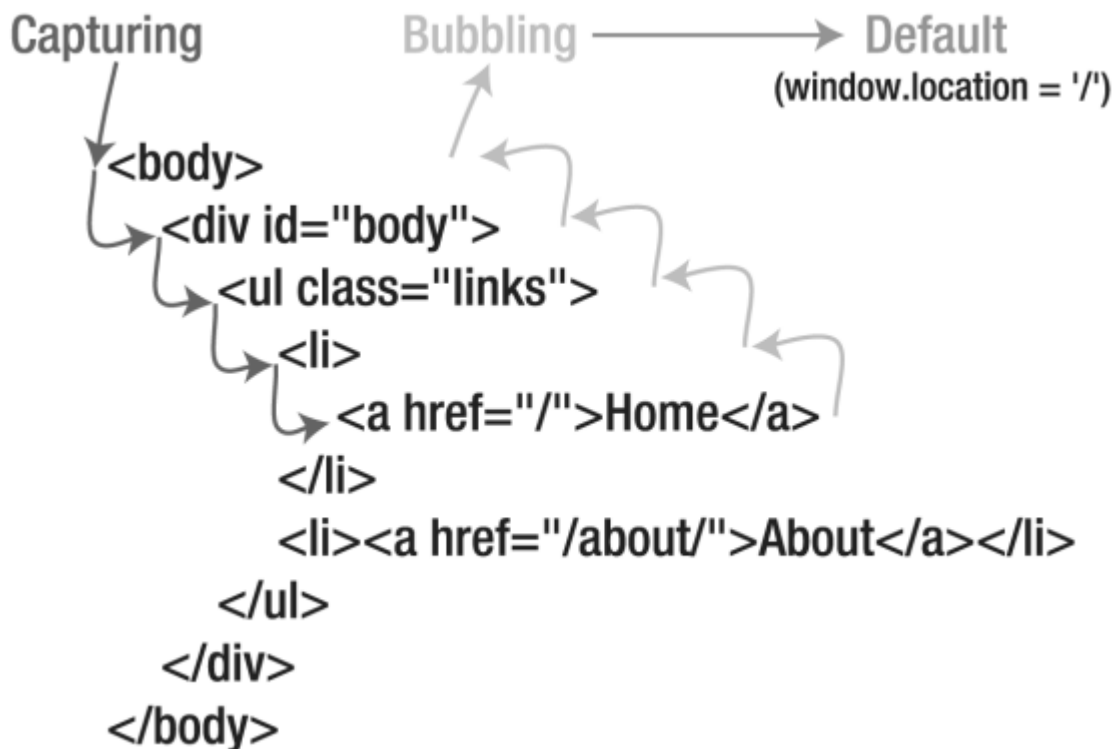


Рис. 6.5. Полный жизненный цикл события

Исходные действия могут быть сведены в понятие всего, что делается браузером без вашего на то конкретного указания. Рассмотрим примеры разного рода совершаемых исходных действий, и событий, по поводу которых это происходит:

- Щелчок на элементе `<a>` перенаправит вас на URL, предоставленный в его атрибуте `href`.
- Нажатие на клавиатуре сочетания `Ctrl+S`, приведет к попытке браузера сохранить физическое представление веб-сайта.
- Отправка HTML `<form>` приведет к передаче данных запроса на определенный URL и перенаправлению браузера по указанному адресу.
- Перемещение указателя мыши над элементом изображения — ``, имеющим атрибут `alt` или `title` (в зависимости от браузера) приведет к появления подсказки, предоставляющего описание изображения.

Все ранее перечисленные действия будут выполняться браузером даже если вы остановите всплытие события, или у вас вообще не будет привязано к элементу никакого обработчика события. Это может привести в ваших сценариях к значительным проблемам. Что делать, если вам нужно задать передаваемой форме иное поведение? Или если нужно, чтобы элемент `<a>` вел себя не так, как предписано его предназначением? Ведь воспрепятствовать всплытию события для того, чтобы предотвратить исходные действия будет не достаточно, вам понадобится некий особенный код, который будет непосредственно управлять этим процессом. Как и в случае с отменой всплытия, есть два способа остановки исходных действий: особый IE-способ, и способ, предусмотренный W3C. Оба эти способа показаны в листинге 6.8. Приведенная в нем функция принимает один аргумент: объект события, передаваемый в обработчик события. Эта функция может быть использована в самом конце обработчика события, таким вот образом: `return stopDefault(e);` — поскольку ваш обработчик тоже должен вернуть `false` (который теперь вернет для вас функция `stopDefault`).

Листинг 6.8. Универсальная функция для предотвращения исходных действий браузера

```
function stopDefault( e ) {
    // Предотвращение исходных действий браузера (W3C)
    if ( e && e.preventDefault )
```

```

    e.preventDefault();
    // Ссылка на остановку действия браузера в IE
else
    window.event.returnValue = false;

return false;
}

```

Теперь, используя функцию `stopDefault`, вы можете остановить любое исходное действие, предоставляемое браузером. Как показано в листинге 6.9, это позволит вам задать в сценарии вполне определенную реакцию на действия пользователя. Код делает так, чтобы все имеющиеся на странице ссылки загружались в отдельном элементе `<iframe>`, а не в открываемой целиком новой странице. Это позволит вам удержать пользователя на странице, и дать ему возможность более интерактивного взаимодействия.

ПРИМЕЧАНИЕ

Предотвращение исходных действий во всех необходимых случаях работает на 95%. Все усложняется при перемещении от браузера к браузеру, из-за того, что предотвращение исходного действия зависит от самого браузера (они не всегда все делают правильно), особенно при работе с предотвращением действий, вызванных нажатием клавиш в текстовых полях, и с предотвращением действий внутри элементов `<iframe>`; во всех остальных случаях все должно проходить более-менее гладко.

Листинг 6.9. Использование `stopDefault()` для подмены функциональных действий браузера

```

// Предположим, на странице уже есть элемент IFrame,
// и его ID имеет значение 'iframe'
var iframe = document.getElementById("iframe");

// Обнаружение на странице всех элементов <a>
var a = document.getElementsByTagName("a");
for ( var i = 0; i < a.length; i++ ) {

    // Привязка к <a> обработчика щелчка
a[i].onclick = function(e) {
    // Установка места нахождения IFrame
    iframe.src = this.href;
    // Предотвращение любых посещений из браузера веб-сайтов, на которые
    // указывают элементы <a> (что является исходным действием)
    return stopDefault( e );
};
}

```

Подмена исходных действий — неразрывное сочетание DOM и событий, которые объединяются в форме ненавязчивого создания DOM-сценариев. Чуть позже, в разделе «Создание ненавязчивых DOM-сценариев», я поговорю об этом с функциональной точки зрения. Тем не менее, не все так гладко; основные разногласия возникают, когда приходит время заняться привязкой ваших обработчиков событий к DOM-элементам. Фактически существуют три различных способа привязки событий, одни хуже, другие лучше, и все они будут рассмотрены в следующем разделе.

Привязка перехватчиков событий

Привязка обработчиков событий к элементам была в JavaScript постоянно развивающимся предметом поиска.

Все началось с браузеров, вынуждающих пользователей делать код обработчика событий встроенным в их HTML-документы. К счастью, эта методика со временем в значительной степени утратила популярность (что нам на руку, поскольку она идет вразрез с принципами абстракции данных при создании ненавязчивых DOM-сценариев).

Когда Netscape и Internet Explorer активно конкурировали, в каждом из них модели регистрации событий развивались по отдельности, но очень похожими путями. В конечном итоге модель Netscape была изменена, чтобы стать стандартом W3C, а модель Internet Explorer осталась той же самой.

На сегодняшний день осталось три безотказных способа регистрации событий. Хотя традиционный метод является ответвлением старого, встраиваемого способа привязки обработчиков событий, но он надежен и сложен в работе. Другие методы относятся к IE и W3C способам регистрации событий. В завершение я продемонстрирую надежный набор способов, которые могут применяться разработчиками без оглядки на используемый браузер.

Традиционная привязка

До сих пор в этой главе я использовал именно традиционный способ привязки событий. Это самый простой и наиболее совместимый способ привязки обработчиков событий. Чтобы им воспользоваться, вы прикрепляете функцию в качестве свойства DOM-элемента, за которым нужно наблюдать. В листинге 6.10 показан ряд примеров прикрепления событий с использованием традиционного метода.

Листинг 6.10. Прикрепление событий с использованием традиционного метода их привязки

```
// Обнаружение первого элемента <form> и прикрепление к нему обработчика
// события 'submit'
document.getElementsByTagName("form")[0].onsubmit = function(e) {
    // Остановка всех попыток передачи данных формы
    return stopDefault( e );
};

// Прикрепление обработчика события нажатия клавиши к элементу <body>
// текущего документа
document.body.onkeypress = myKeyPressHandler;

// Прикрепление к странице обработчика события загрузки
window.onload = function(){ ...; };
```

Эта технология имеет ряд преимуществ и недостатков, о которых вы должны знать при ее использовании.

Преимущества традиционной привязки

Традиционный метод имеет следующие преимущества:

- Самое большое преимущество использования традиционного метода заключается в его невероятной простоте и непротиворечивости, которые в значительной степени гарантируют его однообразную работу независимо от используемого браузера.
- При обработке события ключевое слово `this` ссылается на текущий элемент, что может оказаться очень полезным обстоятельством (как показано в листинге 6.5).

Недостатки традиционного способа привязки

У традиционного метода имеются следующие недостатки:

- Традиционный метод работает только со всплытием событий, но не с захватом и всплытием.
- Одновременно к элементу может быть привязан только один обработчик событий. При работе с популярным свойством `window.onload` существует потенциальная возможность получения довольно странных результатов (отдельные фрагменты кода, используемые для реализации такого же способа привязки событий, полностью переписывают предыдущую обработку). Пример проявления этой проблемы показан в листинге 6.11, где обработчик событий переписывает старый обработчик.
- Параметр объект события доступен только в браузерах, не имеющих отношения к Internet Explorer.

Листинг 6.11. Обработчики события, переписывающие друг друга

```
// Привязка исходного обработчика события
window.onload = myFirstHandler;

// Где-нибудь в другой библиотеке, которую вы включили,
// первый обработчик переписывается, и по окончании загрузки страницы
// вызывается только второй обработчик — 'mySecondHandler'
window.onload = mySecondHandler;
```

Зная о возможности безоговорочной подмены обработчика другими обработчиками событий, вы, скорее всего, выберете традиционные средства привязки только для простых ситуаций, в которых вы сможете быть уверены в коде, работающем вместе с вашим кодом. Тем не менее, обойти эту запутанную ситуацию можно только одним способом — воспользоваться предлагаемыми браузерами современными методами привязки событий.

DOM-привязка: W3C

Единственный, по-настоящему стандартный способ привязки обработчиков событий к DOM-элементам разработан W3C. Памятуя об этом разработчики каждого современного браузера, за исключением Internet Explorer, включают в него поддержку этого способа привязки события.

Код для привязки новой функции обработки довольно прост. Он существует в виде функции для каждого DOM-элемента (которая носит название `addEventListener`), принимающей три параметра: название события (например, `click`), функцию, которая будет обрабатывать событие, и логический флаг для разрешения или отмены захвата события. В листинге 6.12 приводится пример использования функции `addEventListener`.

Листинг 6.12. Пример кодового фрагмента, в котором используется способ привязки обработчиков событий, принятый W3C

```
// Обнаружение первого элемента <form> и привязка к нему обработчика события
// 'submit'
document.getElementsByTagName("form")[0].
  addEventListener('submit',function(e){
    // Остановка всех попыток отправки данных формы
    return stopDefault( e );
  }, false);

// Привязка обработчика события нажатия на клавишу к элементу <body>
// текущего документа
```



```
document.body.addEventListener('keypress', myKeyPressHandler, false);
```

```
// Привязка к странице обработчика события ее загрузки
window.addEventListener('load', function(){ ...; }, false);
```

Преимущества W3C-привязки

Способ привязки, предложенный W3C, имеет следующие преимущества:

- Этот метод поддерживает обе фазы обработки события: и захват и всплытие. Фаза события переключается за счет установки последнего параметра функции `addEventListener` в `false` (для всплытия) или `true` (для захвата).
- Внутри функции обработки события ключевое слово `this` ссылается на текущий элемент.
- Объект события всегда доступен в первом параметре функции обработки.
- К элементу можно привязать сколько угодно событий, без переписывания ранее привязанных обработчиков.

Недостаток W3C-привязки

У способа привязки, предложенного W3C, имеются следующий недостаток:

- Он не работает в Internet Explorer; вместо него в этом браузере нужно применять функцию `attachEvent`.

Если бы Internet Explorer использовал способ привязки обработчиков событий, предложенный W3C, эта глава могла бы быть значительно короче, поскольку не пришлось бы рассматривать альтернативные способы привязки событий. Но на данный момент метод привязки событий, предложенный W3C, является наиболее полноценным и простым в использовании.

DOM-привязка: IE

Способ, используемый для привязки событий в Internet Explorer во многом кажется похожим на способ, предложенный W3C. Но если вникнуть в подробности, выясняется, что в некоторых деталях есть весьма существенные различия. В листинге 6.13 показано несколько примеров привязки обработчиков событий в Internet Explorer.

Листинг 6.13. Примеры привязки обработчиков событий к элементам способом, существующем в Internet Explorer

```
// Обнаружение первого элемента <form> и привязка к нему обработчика события
// 'submit'
document.getElementsByTagName("form")[0].attachEvent('onsubmit',function(){
    // Остановка всех попыток отправки данных формы
    return stopDefault();
},);
```

```
// Привязка обработчика события нажатия на клавишу к элементу <body>
// текущего документа
document.body.attachEvent('onkeypress', myKeyPressHandler);
```

```
// Привязка к странице обработчика события ее загрузки
window.attachEvent('onload', function(){ ...; });
```

Преимущество IE-привязки

Преимущество способа привязки событий, существующего в Internet Explorer, заключается в следующем:

- К элементу можно привязать сколько угодно событий, без переписывания ранее привязанных обработчиков.

Недостатки IE-привязки

Недостатки способа привязки событий, существующего в Internet Explorer, заключаются в следующем:

- Internet Explorer при захвате события поддерживает только фазу всплытия.
- Внутри функции перехватчика ключевое слово `this` указывает на объект `window`, а не на текущий элемент (огромный недостаток IE).
- Объект события доступен только в параметре `window.event`.
- Название события должно иметь префикс «on» — например, `onclick` вместо простого `click`.
- Он работает только в Internet Explorer. Для браузеров, не имеющих отношения к IE, нужно использовать W3C-функцию `addEventListener`.

Поскольку свойства событий отвечают стандарту не в полной мере, существующая в Internet Explorer реализация привязки событий считается весьма ущербной. Из-за множества недостатков до сих пор приходится использовать различные обходные маневры, приводящие эту систему к приемлемому поведению. Но не все еще потеряно: стандартная функция добавления событий к DOM все же существует, и она сможет в значительной степени скрасить ситуацию.

addEventListener и removeEvent

В соревновании, затаенном Петером-Паулем Кохом (Peter-Paul Koch) (в <http://quirksmode.org>) в конце 2005 года, он попросил всех, кто занимается программированием на JavaScript, разработать новую пару функций — `addEventListener` и `removeEvent`, которые смогли бы предоставить пользователям надежный способ добавления и удаления событий в отношении элемента DOM. Я вышел из этого соревнования победителем, создав довольно компактный и достаточно хорошо работающий код. Но позже, один из членов жюри, Дин Эдвардс (Dean Edwards), выпустил другую версию функции, которые значительно превзошли результаты моего творчества. В его реализации использовались традиционные средства привязки обработчиков событий, полностью игнорировавшие современные методы. Благодаря этому его реализация могла работать на большом количестве браузеров, обеспечивая к тому же все необходимые тонкости, связанные с событиями (нормальную работу ключевого слова `this` и стандартный объект события). В листинге 6.14 показан пример кодового фрагмента, в котором используются всевозможные аспекты обработки событий, использующие преимущества новой функции `addEventListener`, где имеет место предотвращение исходной реакции браузера на события, включение нормального объекта события, и включение нормального ключевого слова `this`.

Листинг 6.14. Пример фрагмента кода, в котором используется функция `addEventListener`

```
// Ожидание завершения загрузки страницы
addEventListener( window, "load", function(){

// Отслеживание любого пользовательского нажатия клавиши
addEventListener( document.body, "keypress", function(e){
    // Если пользователь нажал сочетание клавиш Пробел + Ctrl
    if ( e.keyCode == 32 && e.ctrlKey ) {
```

```

// Отображение нашей специальной формы
this.getElementsByTagName("form")[0].style.display = 'block';

// Гарантирование отсутствия странного поведения
e.preventDefault();

}
});
});

```

Функция `addEventListener` предоставляет невероятно простой и в то же время мощный способ работы с DOM-событиями. Если взглянуть на все его преимущества и недостатки, станет совершенно понятно, что эта функция может служить в качестве совместимого и надежного средства работы с событиями. Ее полный исходный код, работающий со всеми браузерами, не требующий большого объема памяти, обрабатывающий ключевое слово `this` и определяющий стандарты функций объекта события, приведен в листинге 6.15.

Листинг 6.15. Библиотека `addEventListener/removeEvent`, созданная Дином Эдвардсом

```

// addEvent/removeEvent written by Dean Edwards, 2005
// with input from Tino Zijdel
// http://dean.edwards.name/weblog/2005/10/add-event/

function addEvent(element, type, handler) {
    // присвоение каждому обработчику события уникального ID
    if (!handler.$$guid) handler.$$guid = addEvent.guid++;

    // создание хэш-таблицы видов событий для элемента
    if (!element.events) element.events = {};

    // создание хэш-таблицы обработчиков событий для каждой пары
    // элемент-событие
    var handlers = element.events[type];
    if (!handlers) {
        handlers = element.events[type] = {};

        // сохранение существующего обработчика события
        // (если он существует)
        if (element["on" + type]) {
            handlers[0] = element["on" + type];
        }
    }

    // сохранение обработчика события в хэш-таблице
    handlers[handler.$$guid] = handler;

    // назначение глобального обработчика события для выполнения
    // всей работы
    element["on" + type] = handleEvent;
};

```

```

// счетчик, используемый для создания уникальных ID
addEvent.guid = 1;

function removeEvent(element, type, handler) {
    // удаление обработчика события из хэш-таблицы
    if (element.events && element.events[type]) {
        delete element.events[type][handler.$$guid];
    }
};

function handleEvent(event) {
    var returnValue = true;

    // захват объекта события (IE использует глобальный объект события)
    event = event || fixEvent(window.event);

    // получение ссылки на хэш-таблицу обработчиков событий
    var handlers = this.events[event.type];

    // выполнение каждого обработчика события
    for (var i in handlers) {
        this.$$handleEvent = handlers[i];
        if (this.$$handleEvent(event) === false) {
            returnValue = false;
        }
    }

    return returnValue;
};

// Добавление к объекту события IE некоторых "упущенных" методов
function fixEvent(event) {
    // добавление стандартных методов событий W3C
    event.preventDefault = fixEvent.preventDefault;
    event.stopPropagation = fixEvent.stopPropagation;
    return event;
};

fixEvent.preventDefault = function() {
    this.returnValue = false;
};

fixEvent.stopPropagation = function() {
    this.cancelBubble = true;
};

```

Преимущества addEvent

Преимущества разработанного Дином Эдвардсом метода привязки событий `addEventListener` заключаются в следующем:

- Он работает на всех браузерах, даже на старых, уже не поддерживаемых браузерах.
- Ключевое слово `this` доступно во всех функциях привязки, и указывает на текущий элемент.
- Нейтрализованы все специфические для конкретного браузера функции для пресечения исходных действий браузера и для остановки всплытия события.
- Объект события всегда передается в качестве первого параметра, независимо от типа браузера.

Недостаток `addEventListener`

В разработанном Дином Эдвардсом методе привязки событий `addEventListener` существует следующий недостаток:

- Он работает только во время фазы всплытия (поскольку в своей основе он использует традиционный метод привязки события).

Учитывая всю мощь функций `addEventListener` и `removeEventListener`, не остается абсолютно никаких причин, препятствующих их использованию в вашем коде. С высоты того, что демонстрирует разработанный Дином исходный код, становится вполне обычной задачей добавление таких свойств, как более четкая стандартизация объекта события, запуска обработки события, и тотального удаления обработчиков события, то есть всего того, чего очень трудно добиться от обычной структуры обработки событий.

Виды событий

Общие события JavaScript могут быть классифицированы на несколько различных категорий. Вероятно наиболее востребованной категорией являются события, связанные с мышью, от них не намного отстают события, связанные с клавиатурой и формами. В следующем списке представлен широкий обзор различных классов существующих событий, которые могут быть обработаны в веб-приложении. А многочисленные примеры работы с событиями приведены в приложении Б.

События, связанные с мышью: Эти события подразделяются на две категории: события, с помощью которых отслеживается текущее местоположение указателя мыши (`mouseover`, `mouseout`), и события, с помощью которых отслеживается щелчок мыши (`mouseup`, `mousedown`, `click`).

События, связанные с клавиатурой: С помощью этих событий отслеживается момент нажатия клавиш, и контекст в пределах которого он произошел — к примеру, с их помощью отслеживается нажатие клавиш внутри элемента `form` в отличие от нажатия клавиши в пределах всей страницы. Как и в случае с мышью, существует три разновидности событий, используемых для отслеживания клавиатуры: `keyup`, `keydown` и `keypress`.

События, связанные с пользовательским интерфейсом: Эти события используются для отслеживания, когда пользователи используют ту или иную часть страницы. С их помощью можно, к примеру, четко определить, когда пользователь начинает вводить данные в элемент формы. Для отслеживания этих обстоятельств используются два события: `focus` и `blur` (для тех случаев, когда объект теряет фокус).

События формы: Эти события напрямую связаны только с тем, что случается в форме и в ее элементах ввода. С помощью события отправки данных — `submit` отслеживается момент передачи данных из формы; с помощью события изменения — `change` отслеживается пользовательский ввод данных в элемент; а событие `select` возникает, когда был обновлен элемент `<select>`.

События загрузки и ошибки: И последний класс событий относится к странице как таковой, с их помощью отслеживается состояние ее загрузки. Они связаны с первоначальной загрузкой страницы пользователем (событие `load`), и с тем моментом, когда он окончательно покидает страницу (события `unload` и `beforeunload`).

Вдобавок к этому с помощью события `error` отслеживается возникновение ошибок JavaScript, позволяя проводить индивидуальную обработку ошибок.

Когда вы разберетесь с основными классами событий, я рекомендую внимательно просмотреть материал приложения Б, в котором я провожу анализ всех широко используемых событий, из работы и поведения на различных браузерах, и даю описание всех премудростей, необходимых, чтобы добиться от них желаемых результатов.

Создание ненавязчивых DOM-сценариев

Все, что изучалось до сих пор было направлено на достижение невероятно важной цели: написанию такого кода JavaScript, который смог бы взаимодействовать с вашими пользователями ненавязчивым и естественным образом. Движущая сила, положенная в основу этого стиля создания сценариев, заключается в появившейся возможности сфокусировать свою энергию на написании качественного кода, способного работать на современных браузерах, который перестает действовать, не создавая проблем в работе на устаревших (не поддерживающих его) браузерах.

Чтобы достичь этой цели, можно объединить три методики, изученные в ходе обучения созданию приложений на основе ненавязчивых сценариев:

1. Все выполняемые функции должны быть проверены. К примеру, если требуется обращение к модели HTML DOM, нужно проверить сам факт ее существования и наличия всех функций, необходимых для ее использования (например, `if (document && document.getElementById)`). Эта методика обсуждалась в главе 2.
2. DOM должна использоваться для быстрого и однообразного обращения к элементам вашего документа. Как только вы узнаете, что браузер поддерживает DOM-функции, можете свободно создавать простой код, без обходных маневров и ненужных ухищрений.
3. И, наконец, все события нужно привязывать к документу в динамическом режиме, используя DOM и функцию `addEventListener`. Теперь уже нельзя где-нибудь воспользоваться какой-нибудь конструкцией, похожей на `...`. С точки зрения ненавязчивого программирования она никуда не годится, поскольку код фактически останется не у дел, если JavaScript отключен, или если пользователь работает на старой версии браузера, не поддерживающей эту конструкцию. Поскольку вы направляете пользователей на бессмысленный URL, те из них, кто не имеет поддержки функций сценария, будут лишены интерактивности.

Если это еще не совсем очевидно, вам нужно симулировать полное отсутствие у пользователя установки JavaScript, или ущербность его браузера. Попробуйте открыть свой браузер, посетить любимую веб-страницу и отключить JavaScript, будет ли она после этого по-прежнему работать? А как насчет всех каскадных таблиц стиля — CSS, и можете ли вы по-прежнему осуществлять все необходимые переходы? И, наконец, можно ли пользоваться вашим веб-сайтом без мыши? Все это должно стать частью завершающей задачи для вашего веб-сайта. Но благодаря тому, что вы приобрели превосходное понимание того, как создается по-настоящему эффективный код JavaScript, издержки от этого перехода незначительны, и его можно достичь с минимальными усилиями.

Предупреждение отключения JavaScript

Сначала нужно выполнить задачу по удалению из ваших HTML-документов всех встроенных обработчиков событий. Есть две, часто появляющиеся проблемные области, на которые нужно обратить внимание в вашем документе:

- Если отключить на странице JavaScript и щелкнуть на любой (на всех) ссылках, смогут ли они перенести вас на нужную веб-страницу? Разработчики довольно часто используют ссылки типа href="" или href="#", предполагающие для получения нужных пользователям результатов разработку некоего дополнительного JavaScript-шаманства.
- Если отключить JavaScript, смогут ли все формы работать и должным образом посылать свои данные? Наиболее распространенная проблема возникает при использовании в качестве динамических меню элементов <select> (которые работают только при включенном JavaScript).

Теперь, последовав этим важным урокам, вы получите веб-страницу, которая будет полезна без ограничений для тех людей, у которых отключен JavaScript, и кто продолжает пользоваться ущербными браузерами.

Обеспечение независимости ссылок от JavaScript

Теперь, когда пользователь может выполнить на странице все действия, нужно обеспечить ему перед выполнением любого действия вполне адекватное извещение. Когда компания Google выпустила Google Accelerator, который проходит по всем ссылкам на странице и кэширует их для вас, пользователи обнаружили, что их электронный адрес, почтовые отправления и сообщения были магическим образом удалены без видимых причин. Это было обусловлено тем фактом, что разработчики помещали (к примеру) на свои страницы ссылки для удаления сообщений, а затем, для подтверждения удаления выводили окно подтверждения (используя JavaScript). Но Google Accelerator полностью игнорировал этот замысел со всплывающим окном, и все равно переходил по ссылке.

Этот сценарий специально упомянут, чтобы заострить ваше внимание на http-спецификации, которая используется для транспортировки всех документов и файлов по сети Интернет. Наиболее простой GET-запрос происходит при щелчке на ссылке; POST-запрос происходит при отправке данных формы. В спецификации утверждается, что никакие GET-запросы не должны иметь побочных разрушительных эффектов (таких как удаление сообщения), поэтому Google Accelerator и работал таким вот образом. Все это в первую очередь происходило из-за плохого программирования, но не со стороны Google, а со стороны разработчиков веб-приложений, создававших ссылки.

Короче говоря, все ссылки на вашем веб-сайте не должны вызывать разрушения информации. Если по щелчку можно удалить, отредактировать или модифицировать любые, принадлежащие пользователю данные, то для достижения этой цели вы, скорее всего, должны вместо этого воспользоваться формой.

Отслеживание блокировки CSS

Одна из крайне неприятных ситуаций связана с браузерами, занимающими промежуточное положение между старыми и новыми разработками, слишком устаревшими для поддержки современных JavaScript-технологий, но достаточно новыми для поддержки стилевых установок CSS. При использовании популярной технологии DHTML может быть элемент, который вначале может быть скрыт (либо за счет значения display, установленного в none, либо за счет значения visibility, установленного в hidden), а затем он постепенно проявляется (за счет использования JavaScript) при первом посещении страницы пользователем. Но если пользователь не имеет включенной поддержки JavaScript, он этот элемент никогда не увидит. Решение этой проблемы показано в листинге 6.16.

Листинг 6.16. Предоставление технологии проявления изображения после загрузки, безотказно работающей при отключенном JavaScript

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
```

```

<head>
  <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />

  <!-- Во время работы сценария к элементу <html> присоединяется новый
    класс, дающий нам возможность узнать, доступен или не JavaScript.-->
  <script>document.documentElement.className = "js";</script>

  <!-- Если JavaScript доступен, сделать текстовый блок невидимым для
    дальнейшего проявления.-->
  <style>.js #fadein { display: none }</style>
</head>
<body>
  <div id="fadein">Блок всего, что нужно проявить ...;</div>
</body>
</html>

```

Теперь эта технология может работать и без DHTML-проявления. Возможность узнать, подключен или отключен JavaScript, и применить стили, дает большой выигрыш предусмотрительным веб-разработчикам.

Доступность события

Последнее, что осталось рассмотреть в разработке по-настоящему ненавязчивого веб-приложения — это вопрос обеспечения работы событий без использования мыши. Решая эту задачу, мы помогаем двум категориям людей: нуждающимся в помощи при доступе к информации (пользователям с ослабленным зрением), и людям, которые не любят пользоваться мышью. (Как-нибудь сядьте за компьютер, отключите от него мышь, и посмотрите, как перемещаться по ресурсам Интернета, используя только клавиатуру. У вас на многое откроются глаза.)

Чтобы сделать события JavaScript более доступными, везде, где используются события `click`, `mouseover` и `mouseout`, вам нужно серьезно подумать о предоставлении альтернативных, не связанных с мышью привязок. К счастью, есть довольно простые способы, с помощью которых можно выправить эту ситуацию:

Событие щелчка (click): Разработчики браузеров предприняли один очень мудрый шаг, и привязали возникновение событие щелчка к каждому нажатию клавиши Enter. В результате этого необходимость в предоставлении каких-то альтернатив в отношении этого события полностью отпала. Тем не менее, нужно иметь в виду, что некоторым разработчикам нравится привязывать обработчики щелчка к имеющимся в формах кнопкам передачи данных — `submit`, чтобы отслеживать, когда пользователь отправляет данные с веб-страницы. Вместо использования этого события, разработчик должен привязываться к событию формы `submit`, что станет более разумной и надежно работающей альтернативой.

Событие прохождения указателя мыши над элементом (mouseover): При перемещениях по веб-странице с использованием клавиатуры вы фактически переключаете фокус с одного элемента на другой. Привязывая обработчик событий как к `mouseover`, так и к `focus`, вы сможете гарантировать наличие совместимого решения как для тех, кто использует клавиатуру, так и для тех, кто пользуется мышью.

Событие выхода указателя мыши за пределы элемента (mouseout): Если событие `focus` переключается с событием `mouseover`, то событие `blur` возникает, когда фокус выходит за пределы элемента. Поэтому событие `blur` можно использовать в качестве средства, имитирующего событие `mouseout` с помощью клавиатуры.

Теперь, узнав, какие пары элементов ведут себя нужным для нас образом, можно еще раз обратиться к листингу 6.3, чтобы создать эффект, похожий на проход указателя мыши, который будет работать и без мыши (см. листинг 6.17).

Листинг 6.17. Привязка к элементам пар событий, позволяющая получить всеобщий доступ к использованию веб-страницы

```
// Обнаружение всех элементов <a>, для привязки к ним обработчиков событий
var li = document.getElementsByTagName("a");
for ( var i = 0; i < a.length; i++ ) {

    // Привязка обработчика событий mouseover и focus к элементу <a>,
    // с помощью которого цвет фона <a> изменяется на синий, когда
    // либо перемещает над ссылкой указатель мыши, либо устанавливает на нее
    // фокус (используя клавиатуру)
    a[i].onmouseover = a[i].onfocus = function() {
        this.style.backgroundColor = 'blue';
    };

    // Привязка обработчика событий mouseout и blur к элементу <a>,
    // с помощью которого цвет фона <a> возвращается к исходному белому,
    // когда пользователь покидает ссылку
    a[i].onmouseout = a[i].onblur = function() {
        this.style.backgroundColor = 'white';
    };
}
}
```

На практике добавление возможности обработки событий клавиатуры в дополнение к обычным событиям работы с мышью представляет собой вполне обычную задачу. Ее решение по крайней мере сможет облегчить пользователям, зависящим от использования клавиатуры, работу с вашим веб-сайтом, что станет большой победой для всех.

Вывод

Теперь, когда вы узнали как перемещаться по DOM, и привязывать к DOM-элементам обработчики событий, а также узнали о всех преимуществах создания ненавязчивого кода JavaScript, можно приступить к созданию крупных приложений и интересных эффектов.

В начале этой главы я представил работу событий в JavaScript и сравнил их с моделями событий в других языках программирования. Затем вам было показано, какую информацию предоставляет модель события, и как ей можно лучше распорядиться. Затем был исследован вопрос привязки событий к DOM-элементам, и различные виды доступных событий. В заключение было показано, как при создании сценариев интегрировать некоторые эффективные и ненавязчивые технологии в вашу веб-страницу.

Далее я собираюсь рассмотреть, как выполняется ряд динамических эффектов и взаимных действий, в которых вся изученная нами технология используется наилучшим образом.

Глава 7 JavaScript и CSS

Взаимодействие JavaScript и CSS — главная опора современного JavaScript-программирования. Фактически от всех современных веб-приложений по меньшей мере требуется использование хоть какой-нибудь формы динамического взаимодействия. Когда это требование выполняется, пользователь получает возможность быстрее перемещаться по информационному наполнению и меньше тратить времени на ожидание загрузки страниц. Объединение динамической технологии с теми идеями о событиях, которые были представлены в главе 6, является основой для создания у пользователей цельного и яркого впечатления.

Каскадные таблицы стилей стали фактическим стандартом для задания стилей и компоновки удобных и привлекательных веб-страниц, и они по-прежнему предоставляют разработчикам самое большое количество возможностей, доставляя при этом пользователям наименьшее количество хлопот. Примечательно, что объединение этих возможностей с JavaScript позволяет создавать мощные интерфейсы, включая такие вещи, как анимация, элементы управления окнами или динамическое отображение данных.

Доступ к информации о стилях

Сочетание JavaScript и CSS всецело направлено на получение результата за счет взаимодействия. Для достижения нужного набора взаимодействующих структур очень важно понимать, что именно вам доступно.

Исходным инструментом для установки и получения присущих элементу свойств CSS, является его свойство `style`. К примеру, если нужно получить высоту элемента, то можно воспользоваться следующим кодом: `elem.style.height`. А если нужно установить определенный размер высоты элемента, можно воспользоваться следующим кодом: `elem.style.height = '100px'`.

При работе со свойствами CSS DOM-элементов вы столкнетесь с двумя проблемами, связанными с их неожиданным поведением. Во-первых, JavaScript при установке любого свойства, относящегося к размеру, требует, чтобы определялась единица измерения (как это было сделано в предыдущем случае). И в то же время, любое, связанное с размером свойство вместо числа также возвращает строку, представляющую свойство `style` элемента (к примеру, `100px` вместо `100`).

Во-вторых, если высота элемента составляет 100 пикселей, и вы пытаетесь получить значение его текущей высоты, то еще не факт, что будет получено ожидаемое от свойства `style` значение `100px`. Все дело в том, что любая информация о стиле, предустановленная с использованием таблиц стиля или встроенного CSS отражается в свойстве `style` не вполне достоверно.

Это вынуждает нас создать для работы с CSS в JavaScript весьма важную функцию: метод для извлечения реально существующих, текущих свойств стиля элемента, который дает точное, вполне ожидаемое значение. Для того, чтобы справиться с проблемой вычисляемых значений стиля, существует довольно надежный набор методов, которым можно воспользоваться для получения фактических, вычисляемых свойств стиля DOM-элемента. При вызове этих методов (которые представлены в W3C- и IE-специфических вариантах) вы получаете реально существующее вычисляемое значение стиля элемента. В этих методах принимаются во внимание все прошлые таблицы стилей и относящиеся к элементу свойства наряду с текущей модификацией JavaScript. Использование этих методов принесет большую пользу при разработке точного представления тех элементов, с которыми вы работаете.

Также важно принять во внимание многочисленные различия, существующие между браузерами при получении вычисляемых значений стиля элемента. Как и во многом остальном, Internet Explorer имеет одну методику получения текущего вычисляемого стилизованного значения элемента, а остальные браузеры используют другую методику, определенную консорциумом W3C.

Функция для обнаружения вычисляемых значений стиля элемента показана в листинге 7.1, а пример вашей новой функции в действии показан в листинге 7.2.

Листинг 7.1. Функция для получения реального вычисленного значение принадлежащего элементу CSS-свойства Style

```
// Получение свойства style (name) определенного элемента (elem)
function getStyle( elem, name ) {
    // Если свойство присутствует в style[], значит, оно было
    // недавно установлено (и является текущим)
    if (elem.style[name])
        return elem.style[name];

    // В противном случае, попытка воспользоваться методом IE
    else if (elem.currentStyle)
        return elem.currentStyle[name];

    // Или методом W3C, если он существует
    else if (document.defaultView && document.defaultView.getComputedStyle) {
        // Вместо text-align в нем используется традиционное правило
        // написания стиля – 'text-align'
        name = name.replace(/([A-Z])/g, "-$1");
        name = name.toLowerCase();

        // Получение объекта style и получение значения свойства
        // (если оно существует)
        var s = document.defaultView.getComputedStyle(elem, "");
        return s && s.getPropertyValue(name);

    // В противном случае, мы используем какой-то другой браузер
    } else
        return null;
}
```

Листинг 7.2. Ситуация, в которой вычисляемое CSS-значение элемента не обязательно совпадает со значением, которое доступно в объекте style

```
<html>
<head>
    <style>p { height: 100px; }</style>
    <script>
window.onload = function(){
    // Обнаружение абзаца для проверки его высоты
    var p = document.getElementsByTagName("p")[0];

    // Проверка высоты традиционным способом
    alert( p.style.height + " значение должно быть равно null" );
}
```

```

    // Проверка вычисляемого значение высоты
    alert( getStyle( p, "height" ) + " должно быть 100px" );
};
</script>
</head>
<body>
    <p>Мой рост должен быть 100 пикселей.</p>
</body>
</html>

```

В листинге 7.2 показано, как можно получить фактическое вычисленное значение CSS-свойства DOM-элемента. В данном случае вы получаете фактическую высоту элемента в пикселях, даже если эта высота установлена через CSS в заголовке файла. Важно отметить, что ваша функция игнорирует альтернативные единицы измерения (к примеру, те, в которых используется процентное отношение). И хотя это решение не является всецело приемлемым, оно заложило неплохие стартовые позиции.

Теперь, располагая этим инструментом, вы можете посмотреть, как получать и устанавливать нужные вам свойства для создания некоторых основных моментов взаимодействия DHTML.

Динамические элементы

Под динамическим понимается элемент, управляемый с использованием JavaScript и CSS для создания нестатических эффектов (простым примером может послужить флажок, указывающий на ваш интерес к информационным бюллетеням, и всплывающая область ввода сообщений электронной почты).

В создании динамических эффектов в основном используются три главных свойства: позиции, размера и видимости. Пользуясь тремя этими свойствами вы можете воспроизводить на современных браузерах наиболее часто встречающиеся моменты взаимодействия с пользователем.

Позиция элемента

Работа с позицией элемента является важной строительной составляющей для разработки на веб-странице интерактивных элементов. Обращение к CSS-свойствам позиции и изменение их значений дает возможность эффективно воспроизводить ряд распространенных анимационных эффектов и интерактивных действий (к примеру, осуществлять перетаскивание).

Важным шагом в работе с позиционированием элементов является знание такого, часто востребованного вопроса, как работа системы позиционирования в CSS. В CSS элементы позиционируются с использованием смещений. Используемое при этом измерение представляет собой значение смещения от левого верхнего угла родительского элемента. Пример системы координат, используемой в CSS, показан на рис. 7.1.



Рис. 7.1. Пример системы координат на веб-странице при использовании CSS

Элементы на странице имеют то или иное смещение сверху (вертикальная координата) и слева (горизонтальная координата). Вообще-то большинство элементов в зависимости от тех элементов, которые их окружают, имеют простое статическое позиционирование. Согласно предложениям стандарта CSS, элемент должен иметь ряд различных схем позиционирования. Чтобы в этом лучше разобраться, посмотрим на обычную HTML веб-страницу, показанную в листинге 7.3.

Листинг 7.3. HTML веб-страница, которой можно воспользоваться для демонстрации различий в позиционировании

```
<html>
<head>
<style>
p{
    border: 3px solid red;
    padding: 10px;
    width: 400px;
    background: #FFF;
}

p.odd {
    /* Сюда помещается информация о позиционировании */
    position: static;
    top: 0px;
    left: 0px;
}
</style>
</head>
<body>
```

```

<p>Lorem ipsum dolor sit amet, consectetur adipiscing elit. Etiam ...</p>
<p class='odd'>Phasellus dictum dignissim justo. Duis nec risus id
                                nunc...</p>
<p>Sed vel leo. Nulla iaculis, tortor non laoreet dictum, turpis diam
                                ...</p>
</body>
</html>

```

Посмотрим, как с установками, имеющимися в нашей простой HTML-странице, изменяется позиционирование второго параграфа при различных схемах компоновки веб-сайта:

Статическое позиционирование: Это исходный способ позиционирования элемента; он просто следует нормальному ходу формирования документа. Свойства сверху — top и слева — left при статическом позиционировании элемента не имеют никакого эффекта. На рис. 7.2 показан абзац, который имеет следующее CSS-позиционирование: position: static; top: 0px; left: 0px;.

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Etiam mi justo, aliquam id, tempus in, gravida ut, eros. Curabitur in sapien. Integer sodales. Curabitur sed tortor. Sed neque. Nulla nunc ipsum, commodo et, ultrices at, feugiat eget, dui. Curabitur nec eros sit amet quam sodales sodales. Vivamus non est. Quisque vulputate venenatis est. Vivamus at urna. Ut dolor. Curabitur vestibulum malesuada metus. Duis posuere, mi sit amet dictum vehicula, pede sem adipiscing pede, vel iaculis lorem nibh vitae justo. Integer nisl mauris, ultricies vitae, lacinia ut, varius ut, nibh.

Phasellus dictum dignissim justo. Duis nec risus id nunc ultrices eleifend. Morbi posuere lobortis massa. Morbi et urna nec pede eleifend dapibus. Curabitur sit amet nibh in tortor rutrum lobortis. Aliquam fringilla tellus nec lorem. Mauris eleifend odio in nibh. Morbi magna dui, faucibus luctus, auctor ac, imperdiet nec, sem. Praesent ullamcorper arcu ut lacus. Phasellus feugiat velit sit amet mi. Quisque scelerisque. Duis lacinia tellus semper purus. Morbi et leo. Aliquam posuere imperdiet nibh. Pellentesque quis neque. In sed velit quis orci rutrum rhoncus.

Sed vel leo. Nulla iaculis, tortor non laoreet dictum, turpis diam lacinia massa, ornare luctus leo eros sit amet sem. Integer bibendum dapibus purus. Donec magna tellus, molestie ut, dapibus sed, feugiat nec, est. Nam faucibus lorem non ante. Integer ut ipsum. Duis facilisis mi non eros. Nulla sollicitudin orci at turpis luctus pharetra. Proin lobortis purus nec tortor. Quisque non metus. Nunc enim est, placerat nec, tristique sed, aliquam in, lectus. Aliquam viverra. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Proin vehicula venenatis odio. Donec viverra commodo lectus. Suspendisse potenti.

Рис. 7.2. Абзац при обычном (статическом) формировании страницы

Относительное позиционирование: Этот способ позиционирования очень похож на статическое позиционирование, поскольку элемент будет так же следовать обычному ходу формирования документа, пока ему не будет указано что-либо иное. Но установка свойств `top` или `left` приведет к смещению элемента по отношению оригинальной (статической) позиции. Пример относительного позиционирования с использованием `CSS position: relative; top: -50px; left: 50px;` показан на рис. 7.3.

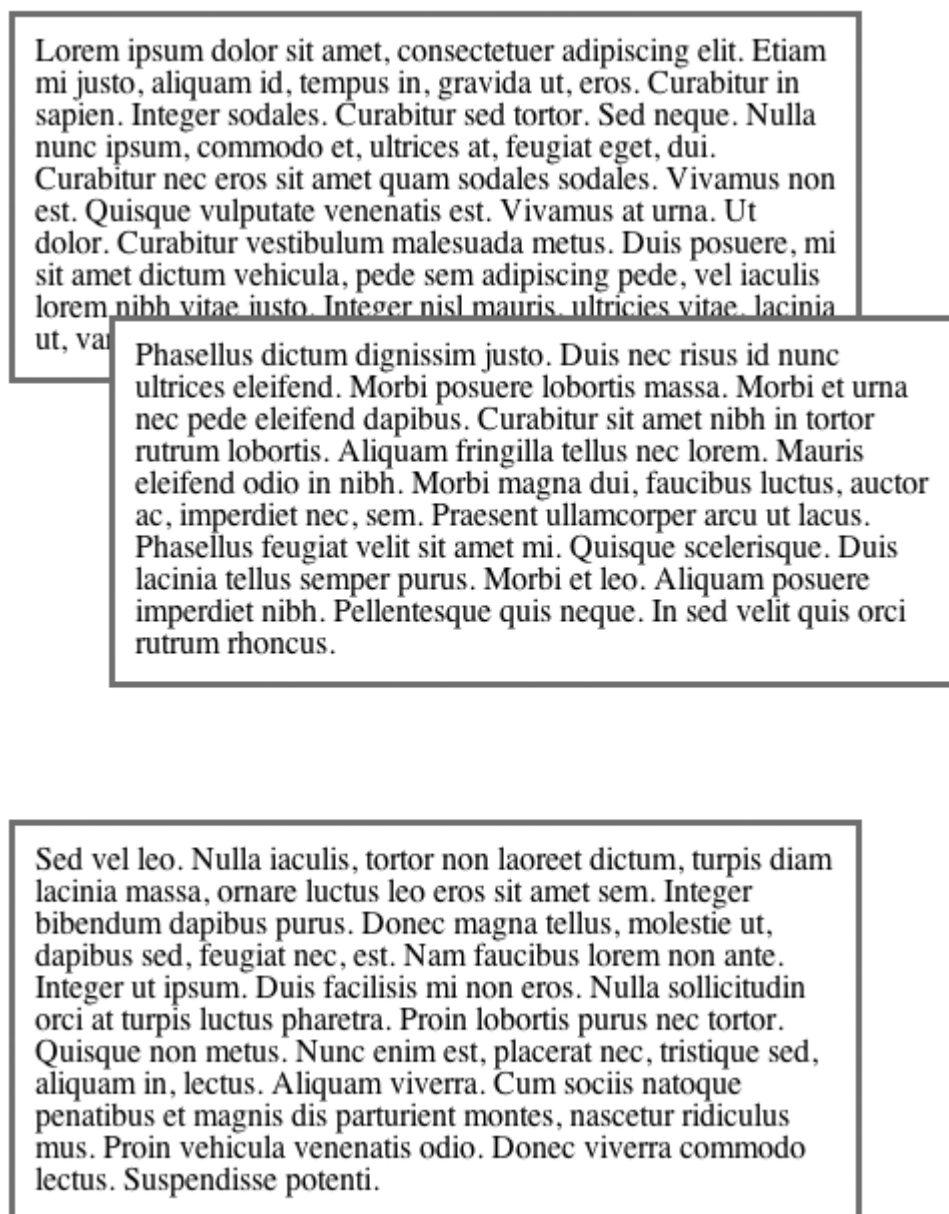


Рис. 7.3. Относительное позиционирование, при котором элемент смещен вверх и перекрывает предыдущий элемент, а не следует обычному ходу формирования документа

Абсолютное позиционирование: Абсолютное позиционирование элемента полностью выключает его из нормального хода формирования документа. При абсолютном позиционировании элемент будет отображен относительно первого родительского элемента, который не имеет статической позиции. Если родительские элементы отсутствуют, он позиционируется относительно всего документа. Пример абсолютного позиционирования с использованием `CSS position: absolute; top: 20px; left: 0px;` показан на рис. 7.4.

Phasellus dictum dignissim justo. Duis nec risus id nunc ultrices eleifend. Morbi posuere lobortis massa. Morbi et urna nec pede eleifend dapibus. Curabitur sit amet nibh in tortor rutrum lobortis. Aliquam fringilla tellus nec lorem. Mauris eleifend odio in nibh. Morbi magna dui, faucibus luctus, auctor ac, imperdiet nec, sem. Praesent ullamcorper arcu ut lacus. Phasellus feugiat velit sit amet mi. Quisque scelerisque. Duis lacinia tellus semper purus. Morbi et leo. Aliquam posuere imperdiet nibh. Pellentesque quis neque. In sed velit quis orci rutrum rhoncus.

Sed vel leo. Nulla iaculis, tortor non laoreet dictum, turpis diam lacinia massa, ornare luctus leo eros sit amet sem. Integer bibendum dapibus purus. Donec magna tellus, molestie ut, dapibus sed, feugiat nec, est. Nam faucibus lorem non ante. Integer ut ipsum. Duis facilisis mi non eros. Nulla sollicitudin orci at turpis luctus pharetra. Proin lobortis purus nec tortor. Quisque non metus. Nunc enim est, placerat nec, tristique sed, aliquam in, lectus. Aliquam viverra. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Proin vehicula venenatis odio. Donec viverra commodo lectus. Suspendisse potenti.

Рис. 7.4. Абсолютное позиционирование, при котором элемент позиционированный относительно левого верхнего угла страницы, находится поверх того элемента, который уже был отображен на этом месте

Фиксированное позиционирование: Фиксированное позиционирование работает за счет позиционирования элемента относительно окна браузера. Установка значения свойств элемента `top` и `left` в 0 пикселей приведет к отображению этого элемента в верхнем левом углу браузера на все время пребывания пользователя на этой странице, при этом все случаи использования прокрутки окна браузера этим элементом будут полностью проигнорированы. Пример фиксированного позиционирования с использованием CSS `position: fixed; top: 20px; right: 0px;` показан на рис. 7.5.

Знание того, как может быть позиционирован элемент, играет важную роль в понимании, где элемент должен быть размещен в DOM-структуре, или какое средство позиционирования нужно использовать для достижения наилучшего эффекта.

Теперь мы посмотрим, как извлечь и использовать точную позицию элемента, независимо от того, какая компоновка применяется, или какие CSS-свойства установлены.

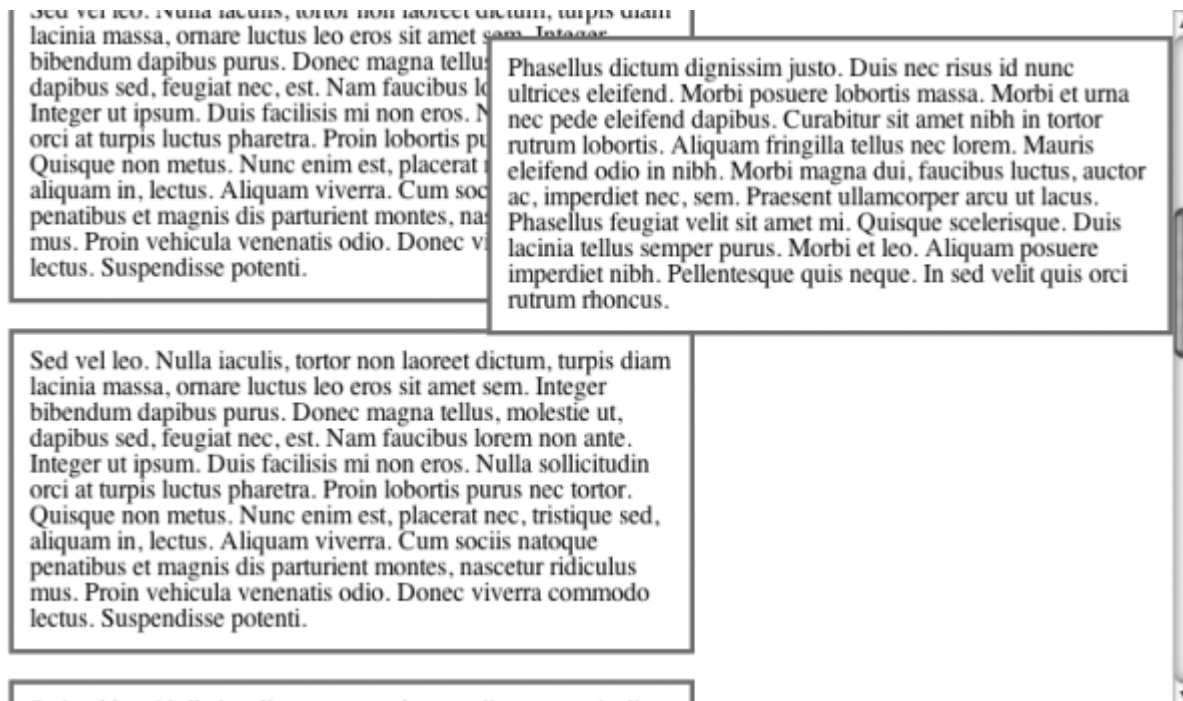


Рис. 7.5. Фиксированное позиционирование, при котором элемент позиционируется в левом верхнем углу страницы, даже если окно браузера прокручено вниз

Получение позиции

Размещение элемента варьируется в зависимости от его CSS-параметров и того контента, который непосредственно к нему примыкает. Один только доступ к CSS-свойствам или к их вычисленным значениям еще не предоставляет возможность узнать точную позицию на странице, или даже внутри другого элемента.

Для начала рассмотрим нахождение позиции элемента на странице. Для получения этой информации в вашем распоряжении имеется несколько свойств элемента. Следующие три свойства поддерживаются всеми современными браузерами, а вот как они их обрабатывают, это уже другой вопрос:

offsetParent: Теоретически это свойство указывает на родительский элемент, внутри которого осуществляется позиционирование. Но на практике элемент, на который ссылается *offsetParent* зависит от браузера (к примеру, в Firefox он ссылается на корневой узел, а в Opera — на непосредственно родительский элемент).

offsetLeft и *offsetTop*: Эти параметры являются горизонтальным и вертикальным смещением элемента в пределах контекста *offsetParent*. Хорошо, что во всех современных браузерах их назначение точно соблюдается.

Теперь самым хитрым делом будет найти способ определения подходящей кроссбраузерной системы изменений размещения элементов. Наиболее унифицированным способом решения этой задачи будет использование методов, представленных в листинге 7.4, с помощью которых осуществляется перемещение вверх по дереву DOM с использованием свойства *offsetParent*, и сложения встречающихся на этом пути значений смещения.

Листинг 7.4. Две вспомогательные функции для определения местоположения элемента (x и y) относительно всего документа

```
// Определение X (горизонтальной слева) позиции элемента
function pageX(elem) {
    // Проверка на достижение корневого элемента
    return elem.offsetParent ?
```

```

// Если не дошли до самого верха, добавление текущего смещения и
// продолжение движения вверх
elem.offsetLeft + pageX( elem.offsetParent ) :

// В противном случае, получение текущего смещения
elem.offsetLeft;
}

// Определение Y (вертикальной сверху) позиции элемента
function pageY(elem) {
// Проверка на достижение корневого элемента
return elem.offsetParent ?

// Если не дошли до самого верха, добавление текущего смещения и
// продолжение движения вверх
elem.offsetTop + pageY( elem.offsetParent ) :

// В противном случае, получение текущего смещения
elem.offsetTop;
}

```

Следующая часть головоломки с позиционированием заключается в определении горизонтальной и вертикальной позиции элемента в пределах его родителя. Важно отметить, что для этого вовсе недостаточно использовать свойства элемента `style.left` или `style.top`, поскольку вам может потребоваться определить позицию элемента, стиль которого не задавался при помощи JavaScript или CSS.

Используя позицию элемента относительно его родителя вы сможете добавлять в DOM дополнительные элементы, относительно их родительских элементов. Это, к примеру, особенно пригодится для создания контекстных подсказок.

Чтобы найти позицию элемента относительно его родительского элемента, нужно опять вернуться к свойству `offsetParent`. Поскольку это свойство не гарантирует возвращения фактического родителя для заданного элемента, для обнаружения разницы между родительским и дочерним элементами необходимо воспользоваться функциями `pageX` и `pageY`. В двух функциях, показанных в листинге 7.5, я пытаюсь сначала воспользоваться свойством `offsetParent`, если оно действительно указывает на фактического родителя текущего элемента; в противном случае я продолжаю перемещаться вверх по DOM, используя методы `pageX` и `pageY` для определения его реальной позиции.

Листинг 7.5. Две функции для определения позиции элемента относительно его родительского элемента

```

// Определение горизонтальной позиции элемента внутри его родителя
function parentX(elem) {
// Если offsetParent указывает на родителя элемента, то раннее
// завершение работы
return elem.parentNode == elem.offsetParent ?
    elem.offsetLeft :

// В противном случае нужно найти позицию относительно всей страницы
// для обоих элементов и вычислить разницу

```

```

    pageX( elem ) -pageX( elem.parentNode );
}

// Определение вертикальной позиции элемента внутри его родителя
function parentY(elem) {
    // Если offsetParent указывает на родителя элемента, то ранее
    // завершение работы
    return elem.parentNode == elem.offsetParent ?
        elem.offsetTop :

        // В противном случае нужно найти позицию относительно всей страницы
        // для обоих элементов и вычислить разницу
        pageY( elem ) -pageY( elem.parentNode );
}

```

Заключительная часть работы с позиционированием элементов заключается в определении позиции элемента относительно его CSS-контейнера. Как ранее уже было рассмотрено, элемент фактически может быть содержимым одного элемента, а позиционирован относительно другого родительского элемента (с использованием относительного и абсолютного позиционирования). Имея это в виду, вы можете вернуться обратно к функции `getStyle` и определить вычисленное значение CSS-смещения, поскольку именно ему позиция и равна.

Чтобы справиться с этой задачей есть две простые интерфейсные функции, показанные в листинге 7.6, которыми можно воспользоваться. Обе они просто вызывают функцию `getStyle`, но также еще и удаляют любую «внешнюю» (пока вы не станете использовать там, где это нужно, не пиксельный формат) информацию о единицах измерения (к примеру, 100px превратится в 100).

Листинг 7.6. Вспомогательные функции для определения CSS-позиционирования элемента

```

// Определения левой позиции элемента
function posX(elem) {
    // Получение вычисляемого значения style и извлечение числа из значения
    return parseInt( getStyle( elem, "left" ) );
}

// Определение верхней позиции элемента
function posY(elem) {
    // Получение вычисляемого значения style и извлечение числа из значения
    return parseInt( getStyle( elem, "top" ) );
}

```

Установка позиции

В отличие от получения позиции элемента, ее установка носит значительно менее гибкий характер. Но когда она используется в сочетании с различными способами задания формата (абсолютным, относительным, фиксированным), вы можете получить вполне сопоставимые и приемлемые результаты.

В настоящее время существует только один способ корректировки позиции элемента — изменение его CSS-свойств. Чтобы сохранить постоянство методологии, вам нужно подвергать изменениям лишь свойства `left` и `top`, хотя существуют и другие свойства (снизу — `bottom` и справа — `right`). Для начала вы можете просто создать

пару функций, показанных в листинге 7.7, которые можно будет использовать для установки позиции элемента независимо от его текущего положения.

Листинг 7.7. Две функции для установки x и y позиций элемента, независимо от его текущей позиции

```
// Функция для установки горизонтальной позиции элемента
function setX(elem, pos) {
    // Установка CSS-свойства 'left' с использованием единицы измерения,
    // выраженной в пикселах
    elem.style.left = pos + "px";
}

// Функция для установки вертикальной позиции элемента
function setY(elem, pos) {
    // Установка CSS-свойства 'top' с использованием единицы измерения,
    // выраженной в пикселах
    elem.style.top = pos + "px";
}
```

Тем не менее, становится вполне очевидной необходимость разработки второго набора функций, показанного в листинге 7.8, который можно будет использовать для установки позиции элемента относительно его последней позиции — к примеру, установить элемент на 5 пикселей левее его текущей позиции. Использование этих методов тесно связано с различными анимационными эффектами, являющимися основой DHTML-разработки.

Листинг 7.8. Две функции для установки позиции элемента относительно его текущей позиции

```
// Функция добавления пикселей к горизонтальной позиции элемента.
function addX(elem,pos) {
    // Получение текущей горизонтальной позиции и добавление к ней
    // смещения.
    setX( posX(elem) + pos );
}

// Функция добавления пикселей к вертикальной позиции элемента.
function addY(elem,pos) {
    // Получение текущей вертикальной позиции и добавление к ней
    // смещения.
    setY( posY(elem) + pos );
}
```

Вот теперь я полностью прошелся по всей палитре работы, связанной с позиционированием элемента. Представление о том, как работает позиционирование элемента, как получать и устанавливать его точную позицию, является основным аспектом работы с динамическими элементами. Следующей стороной этой работы, которую я собираюсь рассмотреть, является точный размер элемента.

Размер элемента

Определение высоты и ширины элемента может быть как невероятно простой, так и мучительно тяжелой задачей, в зависимости от ситуации и той цели, для которой она выполняется. В большинстве случаев для получения текущей высоты и ширины элемента нужно будет всего лишь воспользоваться модифицированной версией функции `getStyle` (см. листинг 7.9).

Листинг 7.9. Две функции для извлечения текущей высоты или ширины DOM-элемента

```
// Получение текущей высоты элемента (с использованием вычисляемого CSS)
function getHeight( elem ) {
    // Получение вычисляемого значения CSS и извлечение необходимого
    // числового значения
    return parseInt( getStyle( elem, 'height' ) );
}

// Получение текущей ширины элемента (с использованием вычисляемого CSS)
function getWidth( elem ) {
    // Получение вычисляемого значения CSS и извлечение необходимого
    // числового значения
    return parseInt( getStyle( elem, 'width' ) );
}
```

Сложности возникают при попытке сделать две вещи: первое, когда вам нужно получить полную высоту элемента, у которого есть предопределенная высота (к примеру, вы начинаете анимацию с 0 пикселей, но вам нужно узнать насколько высоким или широким может быть элемент), и второе, это значение получить невозможно, когда элемент невидим, то есть его свойство `display` имеет значение `none`. Обе эти проблемы возникают при попытке выполнения анимационного эффекта. Вы начинаете анимацию объекта (который к этому моменту вполне возможно имеет значение `display` установленное в `none`) с 0 пикселей, и вам нужно развернуть его в высоту до полного размера.

В листинге 7.10 представлены две функции, показывающие, как можно определить потенциальную высоту и ширину элемента независимо от его текущих значений высоты и ширины. Эта задача выполняется за счет обращения к свойствам `clientWidth` и `clientHeight`, предоставляющим полную возможную область прокрутки элемента.

Листинг 7.10. Две функции для получения полной потенциальной высоты или ширины элемента, даже если он скрыт

```
// Получение полной возможной высоты элемента (в отличие от фактической
// текущей высоты)

function fullHeight( elem ) {
    // Если элемент отображен на экране, то сработает свойство
    // offsetHeight а если оно не сработает, то сработает getHeight()
    if ( getStyle( elem, 'display' ) != 'none' )
        return elem.offsetHeight || getHeight( elem );

    // В противном случае нам придется иметь дело с элементом,
    // у которого display имеет значение none, поэтому
    // нужно переустановить его CSS-свойства, чтобы считать более
    // точный результат
    var old = resetCSS( elem, {
        display: '',
        visibility: 'hidden',
        position: 'absolute'
    });
```

```

// Определяем полную высоту элемента, используя clientHeight,
// а если это свойство не работает, используем getHeight
var h = elem.clientHeight || getHeight( elem );

// В завершение восстанавливаем прежние CSS-свойства
restoreCSS( elem, old );

// и возвращаем полную высоту элемента
return h;
}

// Получение полной возможной ширины элемента (в отличие от фактической
// текущей ширины)
function fullWidth( elem ) {
    // Если элемент отображен на экране, то сработает свойство
    // offsetWidth а если оно не сработает, то сработает getWidth()
    if ( getStyle( elem, 'display' ) != 'none' )
        return elem.offsetWidth || getWidth( elem );

    // В противном случае нам придется иметь дело с элементом,
    // у которого display имеет значение none, поэтому
    // нужно переустановить его CSS-свойства, чтобы считать более
    // точный результат
    var old = resetCSS( elem, {
        display: '',
        visibility: 'hidden',
        position: 'absolute'
    });

    // Определяем полную ширину элемента, используя clientWidth,
    // а если это свойство не работает, используем getWidth
    var w = elem.clientWidth || getWidth( elem );

    // В завершение восстанавливаем прежние CSS-свойства
    restoreCSS( elem, old );

    // и возвращаем полную ширину элемента
    return w;
}

// Функция, используемая для переустановки набора CSS-свойств, которые
// позже можно будет восстановить
function resetCSS( elem, prop ) {
    var old = {};

    // Перебор всех свойств

```

```

for ( var i in prop ) {
    // Запоминание старых значений свойств
    old[ i ] = elem.style[ i ];

    // и установка новых значений
    elem.style[ i ] = prop[i];
}

// возвращение набора значений для использования в функции restoreCSS
return old;
}

// Функция для устранения побочных эффектов функции resetCSS
function restoreCSS( elem, prop ) {
    // Переустановка всех свойств и возвращение им первоначальных значений
    for ( var i in prop )
        elem.style[ i ] = prop[ i ];
}

```

Имея возможность получить как текущую, так и потенциальную высоту и ширину элемента, вы можете попробовать применить некоторые анимационные эффекты, доступные благодаря использованию этих значений. Но перед тем как перейти к подробностям анимации, вам следует рассмотреть вопрос изменения видимости элемента.

Видимость элемента

Видимость элемента — довольно мощный инструмент, который может быть использован в JavaScript для создания многих вещей, от анимации и эффектов и до быстрого вывода шаблонов. Но важнее всего то, что это свойство может быть использовано для быстрого устранения элемента из поля зрения пользователя, предоставляя ему некоторые основные возможности интерактивной работы.

С использованием CSS существует два различных способа эффективного устранения элемента из поля видимости, у каждого из которых есть как свои преимущества, так и непредвиденные последствия, в зависимости от того, как именно они будут использоваться:

- Свойство `visibility` переключает видимость элемента, сохраняя при этом обычный ход формирования документа. Свойство `visibility` может иметь два значения: видимый — `visible` (по умолчанию) и скрытый — `hidden` (чтобы сделать элемент полностью невидимым). К примеру, если у вас есть текст, обрамленный тегами ``, а его свойство `visibility` установлено в `hidden`, это выразится в простом блоке пробелов в тексте, точно такого же размера, как и оригинальный текст. Сравните, к примеру две следующие текстовые строки:

```
// Обычный текст:
Hello John, how are you today?
```

```
// а теперь к элементу 'John' применено visibility: hidden
Hello      , how are you today?
```

- Свойство `display` предоставляет разработчику более широкий выбор для управления формированием элементов. Этот выбор варьируется между линейным значением — `inline` (такие теги, как `` и ``

являются линейными, потому что следуют обычному текстовому потоку), блочным значением — `block` (такие теги, как `<p>` и `<div>` являются блоками, потому, что они разбивают обычный текстовый поток), и значением `none` (которое предписывает полное устранение элемента из документа). Результат установки для элемента свойства `display` выглядит, как будто вы только что удалили элемент из документа, но не полностью, поскольку позже он может быть быстро возвращен в поле видимости. Поведение свойства `display` отображено в следующих строках:

```
// Обычный текст:
Hello John, how are you today?

// а теперь к элементу 'John' применено display: none
Hello, how are you today?
```

Хотя у свойства `visibility` есть свои особенности применения, важность свойства `display` трудно переоценить. Тот факт, что при установке свойства `visibility` в `hidden` элемент при обычном ходе формирования документа все еще существует, снижает возможность применения этого свойства в большинстве приложений. В листинге 7.11 показаны два метода, которые могут быть использованы для переключения видимости элемента с использованием свойства `display`.

Листинг 7.11. Набор функций для переключения видимости элемента с использованием его CSS-свойства `display`

```
// Функция для скрытия элемента (с использованием свойства display)
function hide( elem ) {
    // Определение текущего состояния свойства display
    var curDisplay = getStyle( elem, 'display' );

    // Запоминание состояния свойства display на будущее
    if ( curDisplay != 'none' )
        elem.$oldDisplay = curDisplay;

    // Установка display в none (скрытие элемента)
    elem.style.display = 'none';
}

// Функция показа элемента (с использованием свойства display)
function show( elem ) {
    // Возвращение свойства display к тому значению, которое им
    // использовалось, или использование
    // 'block', если предыдущее состояние этого свойства не было
    // сохранено
    elem.style.display = elem.$oldDisplay || ' ';
}
```

Второй стороной видимости элемента является степень его прозрачности — *opacity*. Корректировка степени прозрачности элемента приводит к результату, очень похожему на корректировку его видимости, но при этом предоставляет больше возможностей для управления его видимостью. Это означает, что вы можете получать элемент с 50% видимостью, и видеть элемент, который находится под ним. И опять-таки, несмотря на то, что все современные браузеры в известной степени поддерживают прозрачность, и Internet Explorer (что касается IE 5.5), и W3C-совместимые браузеры имеют различия в ее реализации. Чтобы обойти это обстоятельство, для управления

прозрачностью элемента нужно создавать стандартную функцию, показанную в листинге 7.12. Уровень 0 означает, что элемент полностью прозрачен, а уровень 100 — что он полностью непрозрачен.

Листинг 7.12. Функция, предназначенная для корректировки уровня прозрачности элемента

```
// Установка уровня прозрачности элемента
// (где уровень является числом в диапазоне 0-100)
function setOpacity( elem, level ) {
    // Если существуют какие-нибудь фильтры, значит,
    // мы имеем дело с IE, и нужно устанавливать фильтр Alpha
    if ( elem.filters )
        elem.style.filters = 'alpha(opacity=' + level + ')';
    // В противном случае мы используем W3C-свойство opacity
    else
        elem.style.opacity = level / 100;
}
```

Теперь, когда в вашем распоряжении имеются способы корректировки позиции, размера и видимости элемента, настало время приступить к исследованию некоторых забавных вещей, которые могут быть созданы при объединении всех имеющихся возможностей.

Анимация

Теперь, когда у вас уже имеются базовые знания, необходимые для осуществления DHTML-операций, давайте рассмотрим один из самых популярных и наглядных эффектов динамических веб-приложений: анимацию. Ее применение в разумных пределах может предоставить пользователю весьма полезную обратную реакцию приложения, к примеру, привлечение внимания к только что созданному на экране элементу.

Начнем с рассмотрения двух различных широко распространенных анимационных приемов, а затем еще раз вернемся к этой теме при рассмотрении наиболее популярных DHTML-библиотек.

Выплывание

Первый анимационный эффект заключается в том, что берется скрытый (с использованием свойство `display`, установленного в `none`) элемент, и вместо использования для его отображения довольно грубой функции `show()`, вы постепенно, в течение секунды, проявляете этот элемент за счет увеличения его высоты. В листинге 7.13 показана функция, которую можно применить для замены функции `show()` эффектом выпадения, который вызывает у пользователя более плавное визуальное впечатление.

Листинг 7.13. Функция, предназначенная для медленного появления скрытого элемента за счет увеличения его высоты в течение секунды

```
function slideDown( elem ) {
    // Начало выплывания вниз с 0
    elem.style.height = '0px';

    // Показ элемента (но вы его не увидите, пока высота равна 0)
    show( elem );

    // Определение полной, потенциальной высоты элемента
    var h = fullHeight( elem );
```

```

// Мы собираемся за секунду показать анимацию, состоящую из
// 20 'кадров'
for ( var i = 0; i <= 100; i += 5 ) {
    // Замкнутое выражение, гарантирующее, что у нас в распоряжении
    // находится именно та переменная 'i', которая нам нужна
    (function(){
        var pos = i;

        // Установка времени ожидания для совершения будущих
        // действий в определенное время
        setTimeout(function(){

            // Установка новой высоты элемента
            elem.style.height = ( pos / 100 ) * h ) + "px";
        }, ( pos + 1 ) * 10);
    })();
}
}

```

Проявление

Следующий анимационный эффект, который мы собираемся рассмотреть, очень похож на предыдущий, но в нем используется функция `setOpacity()`, которая встраивается вместо модификации высоты. Конкретно наша функция (показанная в листинге 7.14) показывает скрытый элемент, а затем проявляет его за счет изменения степени непрозрачности от 0 (полностью прозрачен) до 100% (полностью непрозрачен). Во многом напоминая функцию, показанную в листинге 7.13, эта функция создает у пользователей более плавное визуальное впечатление.

Листинг 7.14. Функция, предназначенная для медленного проявления скрытого элемента за счет увеличения в течение секунды его непрозрачности

```

function fadeIn( elem ) {
    // Начало непрозрачности с 0
    setOpacity( elem, 0 );

    // Отображение элемента (но вы его не увидите, пока непрозрачность
    // равна 0)
    show( elem );

    // Мы собираемся за секунду показать анимацию, состоящую из
    // 20 'кадров'
    for ( var i = 0; i <= 100; i += 5 ) {
        // Замкнутое выражение, гарантирующее, что у нас в распоряжении
        // находится именно та переменная 'i', которая нам нужна
        (function(){
            var pos = i;

            // Установка времени ожидания для совершения будущих
            // действий в определенное время
            setTimeout(function(){

```

```

        // Установка новой степени прозрачности элемента
        setOpacity( elem, pos );

    }, ( pos + 1 ) * 10);
}() );
}
}

```

Примеры этих, и других анимационных эффектов показаны в главе 9.

Браузер

После работы со специфическими DOM-элементами, знания, позволяющие изменять или отслеживать настройки браузера и его компонентов, могут значительно улучшить взаимодействие пользователя с веб-сайтом. Два наиболее важных аспекта работы с браузерами заключаются в определении позиции указателя мыши, и определении степени прокрутки страницы пользователем.

Позиция указателя мыши

Определение позиции указателя мыши является важнейшим аспектом предоставления пользователю возможностей по перетаскиванию элементов и использованию контекстных меню, и обе эти возможности появляются только при использовании взаимодействия между JavaScript и CSS.

Первые две переменные, значение которых нужно установить — это позиции *x* и *y* указателя мыши относительно всей веб-страницы (см. листинг 7.15). поскольку получить координаты указателя можно лишь с помощью событий, связанных с мышью, вам для их захвата потребуется использование общих событий мыши, в частности `MouseMove` или `MouseDown` (дополнительные примеры показаны в разделе «Перетаскивание»).

Листинг 7.15. Две универсальные функции для получения текущей позиции указателя мыши относительно всего пространства страницы

```

// Получение горизонтальной позиции указателя
function getX(e) {
    // нормализация объекта события
    e = e || window.event;

    // Сначала получение позиции из браузеров, не относящихся к IE,
    // а затем из IE
    return e.pageX || e.clientX + document.body.scrollLeft;
}

// Получение вертикальной позиции указателя
function getY(e) {
    // нормализация объекта события
    e = e || window.event;

    // Сначала получение позиции из браузеров, не относящихся к IE,
    // а затем из IE
    return e.pageY || e.clientY + document.body.scrollTop;
}

```

В заключение будет полезно узнать о вторичных, связанных с мышью переменных, позициях указателя x и y , относительно того элемента, с которым в данный момент происходит взаимодействие. В листинге 7.16 приведены две функции, которые можно использовать для извлечения этих значений.

Листинг 7.16. Две функции, предназначенные для получения позиции указателя мыши относительно текущего элемента

```
// Получение X-позиции указателя относительно целевого элемента,
// который используется в объекте события 'e'
function getElementX( e ) {
    // Определение соответствующего смещения элемента
    return ( e && e.layerX ) || window.event.offsetX;
}

// Получение Y-позиции указателя относительно целевого элемента,
// который используется в объекте события 'e'
function getElementY( e ) {
    // Определение соответствующего смещения элемента
    return ( e && e.layerY ) || window.event.offsetY;
}
```

Когда в этой главе мы перейдем к разделу «Перетаскивание», то уделим внимание взаимодействию с мышью еще раз. Дополнительно, чтобы получить больше примеров, связанных с событиями мыши, нужно обратиться к главе 6 и приложению Б, в которых приведено значительно больше примеров работы с мышью.

Область просмотра

Областью просмотра браузера можно считать все, что находится внутри его полос прокрутки. Кроме этого область прокрутки состоит из нескольких компонентов: окна области просмотра, страницы и полос прокрутки. Определение точной позиции и размеров каждого из этих компонентов нужны для разработки четкого взаимодействия с протяженными частями содержимого (среди которых, к примеру, чат-окна с автопрокруткой).

Размер страницы

Первый набор свойств, требующих рассмотрения — высота и ширина текущей веб-страницы. Чаще всего бывает так, что большая часть просматриваемой страницы лежит за пределами области просмотра (что можно определить за счет проверки размеров области просмотра и позиции полосы прокрутки). В листинге 7.17 показаны две функции, использующие вышеупомянутые свойства `scrollWidth` и `scrollHeight`, которые уточняют полную возможную ширину и высоту элемента, а не только текущие отображаемые размеры.

Листинг 7.17. Две функции, предназначенные для определения длины и ширины текущей веб-страницы

```
// Возвращение высоты веб-страницы
// (может изменяться при добавлении к странице нового содержимого)
function pageHeight() {
    return document.body.scrollHeight;
}

// Возвращение ширины веб-страницы
function pageWidth() {
    return document.body.scrollWidth;
}
```

```
}
```

Позиции полос прокрутки

Теперь мы посмотрим, как определить позиции полос прокрутки браузера (или, в другом смысле, определить насколько вниз по странице переместилась область просмотра). Их числовые значения (которые можно получить при помощи функций, показанных в листинге 7.18) необходимы для предоставления динамической прокрутки в самом приложении, не принимая в расчет того, что предоставлено браузером изначально.

Листинг 7.18. Две функции, предназначенные для определения, где находится область просмотра относительно верхней части документа

```
// Функция для определения величины горизонтальной прокрутки браузера
function scrollX() {
    // Сокращение на случай использования Internet Explorer 6 в строгом
    // (strict) режиме
    var de = document.documentElement;

    // Использование свойства браузера pageXOffset, если оно доступно
    return self.pageXOffset ||

    // в противном случае попытка получить прокрутку слева из
    // корневого узла
    ( de && de.scrollLeft ) ||

    // И наконец, попытка получить прокрутку слева из элемента body
    document.body.scrollLeft;
}

// Функция для определения величины вертикальной прокрутки браузера
function scrollY() {
    // Сокращение на случай использования Internet Explorer 6 в строгом
    // (strict) режиме
    var de = document.documentElement;

    // Использование свойства браузера pageYOffset, если оно доступно
    return self.pageYOffset ||

    // в противном случае попытка получить прокрутку сверху из
    // корневого узла
    ( de && de.scrollTop ) ||

    // И наконец, попытка получить прокрутку сверху из элемента body
    document.body.scrollTop;
}
```

Перемещение полос прокрутки

Теперь, располагая текущим смещением полос прокрутки на странице и длиной самой страницы, можно рассмотреть предоставляемый браузерами метод `scrollTo`, который можно использовать для корректировки текущей позиции области просмотра на странице.

Метод `scrollTo` существует в виде свойства объекта `window` (или любого другого элемента, имеющего полосы прокрутки содержимого или `<iframe>`) и принимает два аргумента, смещения `x` и `y`, необходимые для прокрутки области просмотра (или элемента, или `<iframe>`). В листинге 7.19 показаны два примера использования метода `scrollTo`.

Листинг 7.19. Примеры использования метода `scrollTo` для корректировки позиции области просмотра браузера

```
// Если нужно осуществить прокрутку окна браузера до самого верха, можно
// сделать следующее:
window.scrollTo(0,0);

// Если нужно осуществить прокрутку до позиции определенного элемента, можно
// сделать следующее:
window.scrollTo( 0, pageY( document.getElementById("body") ) );
```

Размер области просмотра

Заключительный аспект области просмотра является, наверное, наиболее очевидным: это размер самой области. Знание размера области просмотра дает хорошее представление о том, какую часть содержимого пользователь может в данный момент видеть, независимо от разрешения экрана или размера окна браузера. Для определения значений размера можно воспользоваться двумя функциями, представленными в листинге 7.20.

Листинг 7.20. Две функции, предназначенные для определения высоты и ширины области просмотра браузера

```
// Определение высоты области просмотра
function windowHeight() {
    // Сокращение на случай использования Internet Explorer 6 в строгом
    // (strict) режиме
    var de = document.documentElement;

    // Использование свойства браузера innerHeight, если оно доступно
    return self.innerHeight ||

        // в противном случае попытка получить высоту из корневого узла
        ( de && de.clientHeight ) ||

        // И наконец, попытка получить высоту из элемента body
        document.body.clientHeight;
}

// Определение ширины области просмотра
function windowWidth() {
    // Сокращение на случай использования Internet Explorer 6 в строгом
    // (strict) режиме
```

```

var de = document.documentElement;

// Использование свойства браузера innerWidth, если оно доступно
return self.innerWidth ||

    // в противном случае попытка получить ширину из корневого узла
    ( de && de.clientWidth ) ||

    // И наконец, попытка получить ширину из элемента body
    document.body.clientWidth;
}

```

Пользу от работы с областью просмотра трудно переоценить. Взгляните на современные веб-приложения, к примеру, на Gmail или Campfire, и увидите в них примеры того, как манипуляции с областью просмотра приводят к вполне предсказуемым результатам (Gmail предоставляет контекстные накладки, а Campfire — чаты с автопрокруткой). В главе 11 будут рассмотрены другие способы возможного использования области просмотра, улучшающие восприятие веб-приложений с высоким уровнем интерактивности.

Перетаскивание

Одно из наиболее популярных пользовательских интерактивных действий, доступных в браузере — перетаскивание элемента по странице. Теперь, используя уже изученный материал (возможность определения позиции элемента, ее корректировки, и понимание различий между разными видами позиционирования), вы будете в состоянии полностью разобраться с тем, как работает система перетаскивания элементов.

Для исследования этой технологии, я решил рассмотреть библиотеку DOM-Drag, созданную Аароном Будманом (Aaron Woodman) (<http://boring.youngpup.net/2001/domdrag>). Его библиотека предоставляет массу полезных вещей, включая следующие:

Описатели перетаскивания: У вас может быть один фактически перемещаемый родительский элемент, и другой, перетаскиваемый вместе с ним подэлемент. Это пригодится при создании элементов интерфейса, похожих на окна.

Функции обратного вызова: Вы можете отслеживать определенные события, к примеру, когда пользователь начинает перетаскивание элемента, перетаскивает его, или завершает перетаскивание, а также получать информацию о текущем местоположении элемента.

Минимальная и максимальная области перетаскивания: Вы можете в определенных пределах ограничить область перетаскивания элемента (к примеру, не дать его перетянуть за пределы экрана). Это пригодится для создания полос прокрутки.

Собственная система координат: Если вы не испытываете удобств при работе с системой координат CSS, можно выбрать работу с любой комбинацией отображения координатной системы x/y.

Собственное преобразование системы координат x и y: Вы можете заставить перетаскиваемый элемент двигаться необычными способами (по пульсирующей или круговой колебательной траектории).

Использование системы DOM-Drag не представляет особых трудностей. Сначала к элементу прикрепляется обработчик перетаскивания (кроме этого могут быть определены любые дополнительные варианты работы), а также любые дополнительные функции наблюдения. Некоторые примеры использования DOM-Drag приведены в листинге 7.21.

Листинг 7.21. Использование DOM-Drag для имитации перетаскиваемых окон

```

<html>
<head>
  <title>DOM-Drag – демонстрация перетаскиваемого окна </title>
  <script src="domdrag.js" type="text/javascript"></script>
  <script type="text/javascript">
window.onload = function(){
  // Инициализация функции DOM-Drag, при которой элемент,
  // имеющий ID 'window', становится перетаскиваемым
  Drag.init( document.getElementById("window") );
};
</script>
<style>
#window {
  border: 1px solid #DDD;
  border-top: 15px solid #DDD;
  width: 250px;
  height: 250px;
}
</style>
</head>
<body>
  <h1>Draggable Window Demo</h1>
  <div id="window">Я – перетаскиваемое окно, можете попробовать меня
                                переташить!</div>
</body>
</html>

```

Полностью документированная копия библиотеки DOM-Drag показана в листинге 7.22. Код присутствует в виде единого глобального объекта, чьи методы могут быть вызваны для объектов для инициализации процесса перетаскивания.

Листинг 7.22. Полностью документированная библиотека DOM-Drag

```

varDrag = {

  // текущий перетаскиваемый элемент
  obj: null,

  // функция инициализации для перетаскиваемого объекта
  // o = элемент, действующий в качестве описателя перетаскивания
  // oRoot = перетаскиваемый элемент, если не определено другое,
  // описатель будет перетаскиваемым элементом.
  // minX, maxX, minY, maxY = минимальные и максимальные координаты,
  // разрешенные для элемента
  // bSwapHorzRef = переключатель горизонтальной системы координат
  // bSwapVertRef = переключатель вертикальной системы координат
  // fxMapper, fyMapper = функции для преобразования координат x и y
  // в другие координаты

```



```

init: function(o, oRoot, minX, maxX, minY,
    maxY, bSwapHorzRef, bSwapVertRef, fXMapper, fYMapper) {

    // Отслеживания начала перетаскивания
    o.onmousedown = Drag.start;

    // Определение используемой системы координат
    o.hmode = bSwapHorzRef ? false : true ;
    o.vmode = bSwapVertRef ? false : true ;

    // Определение элемента, который служит описателем перетаскивания
    o.root = oRoot && oRoot != null ? oRoot : o ;

    // Инициализация указанной системы координат
    if (o.hmode && isNaN(parseInt(o.root.style.left )))
        o.root.style.left = "0px";
    if (o.vmode && isNaN(parseInt(o.root.style.top )))
        o.root.style.top = "0px";
    if (!o.hmode && isNaN(parseInt(o.root.style.right )))
        o.root.style.right = "0px";
    if (!o.vmode && isNaN(parseInt(o.root.style.bottom)))
        o.root.style.bottom = "0px";

    // Проверка предоставления пользователем минимальных и
    // максимальных значений координат x и y
    o.minX = typeof minX != 'undefined' ? minX : null;
    o.minY = typeof minY != 'undefined' ? minY : null;
    o.maxX = typeof maxX != 'undefined' ? maxX : null;
    o.maxY = typeof maxY != 'undefined' ? maxY : null;

    // Проверка на существование любых заданных преобразователей
    // x и y координат
    o.xMapper = fXMapper ? fXMapper : null;
    o.yMapper = fYMapper ? fYMapper : null;

    // Добавление оболочки для всех функций, определяемых пользователем
    o.root.onDragStart = new Function();
    o.root.onDragEnd = new Function();
    o.root.onDrag = new Function();
},

start: function(e) {
    // Определение перетаскиваемого объекта
    var o = Drag.obj = this;

    // Нормализация объекта события
    e = Drag.fixE(e);

```

```

// Получение текущих координат x и y
var y = parseInt(o.vmode ? o.root.style.top : o.root.style.bottom);
var x = parseInt(o.hmode ? o.root.style.left : o.root.style.right );

// Вызов функции пользователя с текущими координатами x и y
o.root.onDragStart(x, y);

// Запоминание начальной позиции указателя мыши
o.lastMouseX = e.clientX;
o.lastMouseY = e.clientY;

// Если используется система координат CSS
if (o.hmode) {
    // установка min и max координат там, где они применяются
    if (o.minX != null) o.minMouseX = e.clientX -x + o.minX;
    if (o.maxX != null) o.maxMouseX = o.minMouseX + o.maxX -o.minX;

    // В противном случае применение обычной математической системы
    //координат
} else {
    if (o.minX != null) o.maxMouseX = -o.minX + e.clientX + x;
    if (o.maxX != null) o.minMouseX = -o.maxX + e.clientX + x;
}

// Если используется система координат CSS
if (o.vmode) {
    // установка min и max координат там, где они применяются
    if (o.minY != null) o.minMouseY = e.clientY -y + o.minY;
    if (o.maxY != null) o.maxMouseY = o.minMouseY + o.maxY -o.minY;

    // В противном случае применение обычной математической системы
    //координат
} else {
    if (o.minY != null) o.maxMouseY = -o.minY + e.clientY + y;
    if (o.maxY != null) o.minMouseY = -o.maxY + e.clientY + y;
}

// отслеживание событий перетаскивания и завершения
// перетаскивания
document.onmousemove = Drag.drag;
document.onmouseup = Drag.end;

return false;
},

// Функция, предназначенная для отслеживания всех перемещений
// указателя мыши в ходе события перетаскивания
drag: function(e) {

```

```

// Нормализация объекта события
e = Drag.fixE(e);

// получение нашей ссылки на перетаскиваемый элемент
var o = Drag.obj;

// получение позиции указателя мыши в пределах окна
var ey = e.clientY;
var ex = e.clientX;

// Получение текущих координат x и y
var y = parseInt(o.vmode ? o.root.style.top : o.root.style.bottom);
var x = parseInt(o.hmode ? o.root.style.left : o.root.style.right );
var nx, ny;

// Если была установлена минимальная позиция X, убедиться в том,
// что она не пройдена
if (o.minX != null) ex = o.hmode ?
    Math.max(ex, o.minMouseX) : Math.min(ex, o.maxMouseX);

// Если была установлена максимальная позиция X, убедиться в том,
// что она не пройдена
if (o.maxX != null) ex = o.hmode ?
    Math.min(ex, o.maxMouseX) : Math.max(ex, o.minMouseX);

// Если была установлена минимальная позиция Y, убедиться в том,
// что она не пройдена
if (o.minY != null) ey = o.vmode ?
    Math.max(ey, o.minMouseY) : Math.min(ey, o.maxMouseY);

// Если была установлена максимальная позиция Y, убедиться в том,
// что она не пройдена
if (o.maxY != null) ey = o.vmode ?
    Math.min(ey, o.maxMouseY) : Math.max(ey, o.minMouseY);

// Вычисление координат x и y последнего перемещения
nx = x + ((ex -o.lastMouseX) * (o.hmode ? 1 : -1));
ny = y + ((ey -o.lastMouseY) * (o.vmode ? 1 : -1));

// и преобразование их с помощью x или y функции преобразования
// координат (если таковая предоставлена)
if (o.xMapper) nx = o.xMapper(y)
else if (o.yMapper) ny = o.yMapper(x)

// Установка для элемента новых x и y координат
Drag.obj.root.style[o.hmode ? "left" : "right"] = nx + "px";
Drag.obj.root.style[o.vmode ? "top" : "bottom"] = ny + "px";

```

```

// и запоминание последней позиции указателя мыши
Drag.obj.lastMouseX = ex;
Drag.obj.lastMouseY = ey;

// Вызов пользовательской функции onDrag с текущими координатами
// x и y
Drag.obj.root.onDrag(nx, ny);

return false;
},

// Функция, обрабатывающая завершение события перетаскивания
end: function() {
    // События мыши больше не отслеживать (поскольку перетаскивание
    // уже произошло)
    document.onmousemove = null;
    document.onmouseup = null;

    // В конце перетаскивания вызов нашей специальной функции
    // onDragEnd с координатами элемента x и y
    Drag.obj.root.onDragEnd(
        parseInt(Drag.obj.root.style[Drag.obj.hmode ? "left" : "right"]),
        parseInt(Drag.obj.root.style[Drag.obj.vmode ? "top" : "bottom"]));
    // No longer watch the object for drags
    Drag.obj = null;
},

// Функция, предназначенная для нормализации объекта события
fixE: function(e) {
    // Если объекта события не существует, значит это IE, и нужно
    // предоставить объект события IE
    if (typeof e == 'undefined') e = window.event;

    // Если свойство layer не установлено, получение
    // значений из эквивалентного свойства offset
    if (typeof e.layerX == 'undefined') e.layerX = e.offsetX;
    if (typeof e.layerY == 'undefined') e.layerY = e.offsetY;

    return e;
}
};

```

Если честно, то DOM-Drag, возможно, лишь одна из сотен JavaScript-библиотек перетаскивания. Тем не менее, к ней у меня особое пристрастие, благодаря ее качественному объектно-ориентированному синтаксису и относительной простоте. В следующем разделе я рассмотрю библиотеку Scriptaculous, которая обладает превосходной и мощной реализацией перетаскивания, которую я настоятельно рекомендую опробовать в работе.

Библиотеки

Если нужно разработать какой-нибудь эффект или средство взаимодействия, может получиться так же, как и со многими другими трудоемкими задачами в JavaScript — вполне вероятно, что подобная вещь уже была создана. Давайте проведем краткий обзор трех разных библиотек, предоставляющих различные средства DHTML-взаимодействия, чтобы получить представление о том, что вам доступно, как разработчику.

moо.fx и jQuery

Есть две небольшие библиотеки, которые очень хорошо подходят для управления простыми эффектами: moо.fx и jQuery. Обе они предоставляют основные комбинации эффектов, которые могут быть объединены для создания вполне впечатляющей, но несложной анимации. Дополнительная информация о каждой из этих библиотек может быть найдена на связанных с ними веб-страницах. В листинге 7.23 приведены некоторые основные примеры использования этих библиотек.

Листинг 7.23. Основные примеры анимации, получаемой с использованием библиотек moо.fx и jQuery

```
// Простая анимация, в которой скрытый элемент сначала показывается
// за счет расширения, а потом, когда этот процесс завершится,
// снова сжимается

// Реализация этого анимационного эффекта в moо.fx
new fx.Height( "side", {
  duration: 1000,
  onComplete: function() {
    new fx.Height( "side", { duration: 1000 } ).hide();
  }
}).show();

// Реализация в jQuery
$("#side").slideDown( 1000, function(){
  $(this).slideUp( 1000 );
});

// Еще одна простая анимация, в которой высота, ширина и непрозрачность
// элемента одновременно сокращаются (или уменьшаются), чем достигается
// довольно интересный эффект скрытия

// Реализация этого анимационного эффекта в moо.fx
new fx.Combo( "body", {
  height: true,
  width: true,
  opacity: true
}).hide();

// Реализация анимации в jQuery
$("#body").hide( "fast" );
```

Наверное, судя по примерам, вы сможете сказать, что библиотеки moо.fx и jQuery действительно упрощают создание некоторых, довольно тонких анимационных эффектов. В обоих проектах на их веб-сайтах представлено множество примеров их кода в действии, что может послужить великолепным способом изучения механизмов работы несложной JavaScript-анимации:

- Домашняя страница moo.fx: <http://moofx.mad4milk.net/>
- Примеры документации mootoolkit: <http://moofx.mad4milk.net/documentation/>
- Домашняя страница jQuery: <http://jquery.com/>
- Документация по эффектам и примеры jQuery: <http://jquery.com/docs/fx/>

Scriptaculous

Если потребовалось бы возвести на королевский трон какую-нибудь из DHTML-библиотек, то речь могла бы идти только о Scriptaculous. Построенная на основе популярной библиотеки Prototype, Scriptaculous предоставляет огромное количество различных интерактивных средств, то есть практически все, от анимаций и эффектов, до интерактивных действий (например, перетаскивания). На веб-сайте Scriptaculous может быть найдена масса сведений и примеров:

- Домашняя страница: <http://script.aculo.us/>
- Документация: <http://wiki.script.aculo.us/scriptaculous/>
- Демонстрационные программы: <http://wiki.script.aculo.us/scriptaculous/show/Demos/>

Одной из областей, в которой Scriptaculous предоставляет наивысший уровень возможностей, сохраняя при этом наилучший уровень простоты использования, является перетаскивание. Чтобы вы смогли составить себе представление, я привожу два простых примера.

Перестроение путем перетаскивания

Одной из задач, решаемых легко и просто с помощью Scriptaculous является перестроение списков. Если принять во внимание простоту кода (и легкость получения доступа к функциональным возможностям Ajax, продемонстрированную на веб-сайте библиотеки), это решение настоятельно рекомендуется для большинства веб-разработчиков. Пример, показанный в листинге 7.24 представляет собой перестраиваемый список, созданный с использованием библиотеки Scriptaculous.

Листинг 7.24. Способ создания списка, перестраиваемого за счет технологии, доступной в библиотеке Scriptaculous

```
<html>
<head>
  <title>script.aculo.us — демонстрация перестроения путем
    перетаскивания </title>
  <script src="prototype.js" type="text/javascript"></script>
  <script src="scriptaculous.js" type="text/javascript"></script>
  <script src="effects.js" type="text/javascript"></script>
  <script src="dragdrop.js" type="text/javascript"></script>
  <script type="text/javascript">
window.onload = function() {
  // Превращение элемента с id равным 'list' в перетаскиваемый,
  // перестраиваемый список
  Sortable.create('list');
};
</script>
</head>
<body>
  <h1>Перестроение путем перетаскивания</h1>
```

```
<p>Чтобы перестроить элемент его нужно перетащить.</p>
```

```
<ul id="list">
  <li>Элемент № 1</li>
  <li>Элемент № 2</li>
  <li>Элемент № 3</li>
  <li>Элемент № 4</li>
  <li>Элемент № 5</li>
  <li>Элемент № 6</li>
</ul>
</body>
</html>
```

Я надеюсь, что этот пример убедит вас в мощности, заключенной внутри этой библиотеки, но если этого недостаточно, вы можете посмотреть следующий пример создания управляющего элемента — ползунок для ввода данных.

Ползунок для ввода данных

Библиотека Scriptaculous предоставляет ряд управляющих элементов, которые можно использовать для решения общих вопросов разработки интерфейса. Управляющий элемент, который нетрудно получить с использованием большинства библиотек перетаскивания — это ползунок для ввода данных (сдвигающийся ползунок для получение числового ввода) и Scriptaculous в этом смысле не исключение, о чем свидетельствует код, представленный в листинге 7.25.

Листинг 7.25. Использование ползунка для ввода данных из библиотеки Scriptaculous, чтобы получить альтернативный способ ввода в форму вашего возраста

```
<html>
<head>
  <title>script.aculo.us — демонстрация ползунка для ввода данных</title>
  <script src="prototype.js" type="text/javascript"></script>
  <script src="scriptaculous.js" type="text/javascript"></script>
  <script src="effects.js" type="text/javascript"></script>
  <script src="dragdrop.js" type="text/javascript"></script>
  <script src="controls.js" type="text/javascript"></script>
  <script type="text/javascript">
window.onload = function(){
  // Превращение элемента, имеющего значение ID, равное ageHandle,
  // в ползунок, и элемента, имеющего значение ID, равное ageBar,
  // в шкалу ползунка
  new Control.Slider( 'ageHandle', 'ageBar', {
    // При перемещении или завершении перемещения ползунка
    // вызывается функция updateAge
    onSlide: updateAge
  });

  // Обработка любых перемещений ползунка
  function updateAge(v) {
```

```

        // при обновлении позиции ползунка, обновление значения
        // элемента age с целью представления текущего возраста
        // пользователя
        $('age').value = Math.floor( v * 100 );
    }
};
</script>
</head>
<body>
    <h1>Демонстрация ползунка для ввода данных </h1>

    <form action="" method="POST">
        <p>Сколько Вам лет? <input type="text" name="age" id="age" /></p>

        <div id="ageBar" style="width:200px; background: #000; height:5px;">
            <div id="ageHandle" style="width:5px; height:10px;
                background: #000; cursor:move;"></div>
        </div>

        <input type="submit" value="Submit Age"/>
    </form>
</body>
</html>

```

Я настоятельно советую, прежде чем решиться на создание какого-нибудь следующего интерактивного средства проверить его наличие в некоторых DHTML-библиотеках, следуя тому простому факту, что авторы библиотек скорее всего уже затратили больше времени и усилий на разработку именно этого средства, чем вы на создание всего своего приложения. Широкое использование библиотек наверняка позволит вам существенно сэкономить время на разработке приложения.

Вывод

Возможности, открывающиеся в веб-приложении при использовании динамических интерактивных средств, предоставляют замечательные способы достижения новых уровней скорости и удобства работы ваших пользователей. Кроме того, при использовании любых популярных библиотек, вы сможете значительно снизить сроки разработки приложения. В следующей главе мы объединим все изученные в этой главе интерактивные технологии для создания полноценного интерактивного приложения.

В этой главе мы рассмотрели все многообразие технологий, направленных на достижение необходимого уровня симбиоза между JavaScript и CSS. В результате была получена возможность создания впечатляющих анимационных эффектов и динамических средств взаимодействия с пользователем.

Необходимо помнить, что добавления на веб-страницу любых форм динамической интерактивности может отпугнуть часть вашей аудитории. Нужно всегда заботиться о том, чтобы ваше приложение не теряло удобств в работе с ним, даже если будут отключены JavaScript или CSS. Создание приложения, способного с легкостью работать в упрощенном режиме, должно быть идеалом для любого JavaScript-разработчика.

Глава 8 Усовершенствование форм

Формы являются средством получения от пользователя структурированных данных, и поэтому имеют для веб-разработчиков особую важность. По своей сути, форма имеет совсем немного ограничений на действия пользователя, на характер вводимых им данных, и на то, в каком качестве ее использовать.

На той стадии разработки, когда уже создана семантически развитая форма, настает время добавить к ней код JavaScript, предоставляющий пользователю дополнительную ответную реакцию. Зная как или почему с формой может что-то произойти, пользователь сможет быстрее ее заполнить, и получить от работы с ней лучшее впечатление.

В этой главе мы собираемся рассмотреть выполнение основной проверки формы на стороне клиента и предоставление пользователю результатов этой проверки вполне толковым и ненавязчивым образом. Затем мы собираемся рассмотреть ряд способов, позволяющих повысить общие удобства пользования формой. Объединение этих двух технологий может быть использовано для предоставления пользователям существенно улучшенных форм, заполнение которых может превратиться в сплошное удовольствие.

Проверка данных формы

Добавление к веб-странице проверки формы на стороне клиента может обеспечить пользователям ускорение работы, но без особых выгод: проверка формы на стороне клиента никогда не заменит проверку на стороне сервера, она ее может только усилить. Стало быть, добавление к веб-странице проверки формы на стороне клиента — превосходный пример уже изученной вами ненавязчивой технологии разработки сценариев.

Перед тем как приступить к составлению любого, связанного с формой сценария, нужно создать форму и убедиться в том, что она работает как и планировалось (например, проверка пользовательского ввода, выдача соответствующих сообщений об ошибках, и т. д.). В этой главе мы собираемся использовать семантически выверенную XHTML-форму. Внутри этой формы все элементы `<input>` имеют четкую классификацию (например, элементы, по своему типу относящиеся к тексту, имеют в атрибуте `class` значение `text`), и содержатся внутри соответствующих наборов полей с точными обозначениями. Все это можно увидеть в листинге 8.1.

Листинг 8.1. Простая XHTML-форма, которую можно улучшить за счет JavaScript

```
<html>
<head>
  <title>Простая форма</title>
</head>
<body>
<form action="" method="POST">
  <fieldset class="login">
    <legend>Регистрационные данные</legend>
    <label for="username" class="hover">Имя пользователя</label>
    <input type="text" id="username" class="required text"/>

    <label for="password" class="hover">Пароль</label>
    <input type="password" id="password" class="required text"/>
  </fieldset>
  <fieldset>
    <legend>Личные сведения</legend>

    <label for="name">Имя</label>
```

```

    <input type="text" id="name" class="required text"/><br/>

    <label for="email">Адрес электронной почты</label>
    <input type="text" id="email" class="required email text"/><br/>

    <label for="date">Дата</label>
    <input type="text" id="date" class="required date text"/><br/>

    <label for="url">Веб-сайт</label>
    <input type="text" id="url" class="url text" value="http://"/><br/>

    <label for="phone">Телефон</label>
    <input type="text" id="phone" class="phone text"/><br/>

    <label for="age">Вам уже исполнилось 13 лет?</label>
    <input type="checkbox" id="age" name="age" value="yes"/><br/>

    <input type="submit" value="Submit Form" class="submit"/>
  </fieldset>
</form>
</body>
</html>

```

Следующий шаг заключается в применении к форме некоторых основных CSS-стилей, чтобы придать ей более приглядный вид. Это поможет вам в следующих разделах главы подобающим образом отобразить сообщения об ошибках и ответную реакцию. Используемая в форме CSS показана в листинге 8.2.

Листинг 8.2. Таблица стилей CSS, используемая для улучшения внешнего вида вашей формы

```

form {
    font-family: Arial;
    font-size: 14px;
    width: 300px;
}

fieldset {
    border: 1px solid #CCC;
    margin-bottom: 10px;
}

fieldset.login input {
    width: 125px;
}

legend {
    font-weight: bold;
    font-size: 1.1em;
}

```

```

label {
    display: block;
    width: 60px;
    text-align: right;
    float: left;
    padding-right: 10px;
    margin: 5px 0;
}

input {
    margin: 5px 0;
}

input.text {
    padding: 0 0 0 3px;
    width: 172px;
}

input.submit {
    margin: 15px 0 0 70px;
}

```

Копия экрана, показанная на рис. 8.1, даст вам достаточное представление о внешнем виде формы (готовой к наслению новых режимов работы за счет использования JavaScript).

Login Information

Username

Password

Personal Information

Name

Email

Date

Website

Phone

Over 13?

Рис. 8.1. Копия экрана стилизованной формы, к которой будет добавляться новый режим работы за счет применения JavaScript

Теперь, имея в своем распоряжении хорошо стилизованную форму, можно приступить к углубленному рассмотрению вопросов проверки формы на стороне клиента. Для этой проверки существует ряд технологий, наиболее часто применяемых к формам. Все эти технологии вращаются вокруг обеспечения того, что данные введенные пользователем в форму, соответствуют ожиданиям программы на серверной стороне.

Главное преимущество обеспечение проверки на стороне клиента состоит в том, что у пользователей появляется практически мгновенная ответная реакция относительно введенных ими данных, которая помогает лишь улучшить общее впечатление от ввода информации в форму. Должно быть абсолютно ясно, что решение о реализации проверки формы на стороне клиента, не означает, что нужно убрать или проигнорировать проверку на стороне сервера. Проверка формы должна продолжаться и при выключенном JavaScript, обеспечивая пользователям, не имеющим включенного JavaScript, возможность ее дальнейшего использования.

В этом разделе мы собираемся рассмотреть определенный код, необходимый для проверки ряда различных элементов ввода данных, чтобы удостовериться, что они содержат определенные данные, востребованные формой. По отдельности каждая из этих проверочных процедур может и не играть особой роли, но в совокупности они могут обеспечить полный набор для проверки и тестирования, показанный в следующем разделе.

Обязательные поля

Возможно, самая важная из всех проводимых проверок полей, относится к проверке обязательного поля (это значит, что пользователь обязательно должен ввести в него данные). В большинстве случаев это требование может быть сведено к проверке того, что это поле не пустое. Но иногда у поля может быть уже введенное в него значение по умолчанию, а это значит, что у вас должна быть проверка, учитывающая такую возможность, и позволяющая убедиться в том, что пользователь как минимум внес в предоставленные полем данные хотя бы какие-нибудь изменения. Эти две проверки охватывают большинство полей формы, включая `<input type="text">`, `<select>` и `<textarea>`.

Но проблемы возникают при попытке обнаружить, изменял ли пользователь значения обязательных полей флажков или переключателей. Чтобы обойти эту проблему нужно найти все поля с таким же названием (которые составляют совокупность элементов поля), а затем проверить, устанавливал ли пользователь любой из них.

Пример проверки обязательных полей показан в листинге 8.3.

Листинг 8.3. Проверка на модификацию обязательного поля (включая флажки и переключатели)

```
// Универсальная функция проверки элемента ввода на наличие введенной
// информации
function checkRequired( elem ) {
    if ( elem.type == "checkbox" || elem.type == "radio" )
        return getInputsByName( elem.name ).numChecked;
    else
        return elem.value.length > 0 && elem.value != elem.defaultValue;
}

// Обнаружение всех элементов ввода с определенным именем (для обнаружения
// и работы с флажками и переключателями)
function getInputsByName( name ) {
    // Массив для подходящих входных элементов
    var results = [];
    // Отслеживание, сколько из них было установлено
    results.numChecked = 0;

    // Обнаружение всех элементов ввода в документе
    var input = document.getElementsByTagName("input");
    for ( var i = 0; i < input.length; i++ ) {
```

```

// Обнаружение всех полей с определенным именем
if ( input[i].name == name ) {
    // Сохранение результатов, чтобы впоследствии их можно было
    // вернуть
    results.push( input[i] );
    // Запоминание количества полей, подвергавшихся
    // установке
    if ( input[i].checked )
        results.numChecked++;
}
}
// Возвращение набора подходящих полей
return results;
}

// Ожидание окончания загрузки документа
window.onload = function()
    // Получение формы и отслеживание попытки отправки данных.
    document.getElementsByTagName("form")[0].onsubmit = function(){

        // Получение проверяемого элемента ввода
        var elem = document.getElementById("age");

        // Определение установки флажка в поле age
        if ( ! checkRequired( elem ) ) {
            // Отображение сообщения об ошибке и предотвращение отправки
            // данных формы.
            alert( "Обязательное поле не отмечено - " +
                "для использования сайта Вам должно быть свыше 13 лет". );
            return false;
        }

        // Получение проверяемого элемента ввода
        var elem = document.getElementById("name");

        // Определение ввода в поле name какого-нибудь текста
        if ( ! checkRequired( elem ) ) {
            // Если текст не введен, отображение сообщения об ошибке и
            // предотвращение отправки данных формы.
            alert( "Обязательное поле не заполнено - пожалуйста,
                введите Ваше имя". );
            return false;
        }
    };
};
};

```

Справившись с проверкой заполнения обязательных полей, нужно убедиться в том, что введенные поля содержат вполне ожидаемые значения. В следующем разделе мы собираемся рассмотреть, как осуществляется проверка содержимого полей.

Соответствие шаблону

Второй составляющей проверки большинства элементов ввода (особенно текстовых полей) является определение соответствия шаблону, чтобы проверить, что содержимое полей отвечает определенным предположениям.

При использовании следующей технологии важно понять, что ваши требования к содержимому поля должны быть точно и ясно определены. В противном случае не все пользователи смогут понять, что именно от них требуется. Неплохим примером подобных требования может послужить запрос дат в определенном формате, поскольку даты меняются в зависимости от сложившихся традиций и даже от разных подходов в спецификациях.

В этом разделе мы собираемся рассмотреть ряд различных технологий, которые могут быть использованы для проверки содержимого полей, включая адреса электронной почты, URL, телефонные номера и даты.

Адреса электронной почты

Поле запроса адреса электронной почты встречается в веб-формах довольно часто, поскольку это практически повсеместная форма идентификации и связи. Но провести подлинную проверку на истинность электронного адреса (в соответствии со спецификациями на которых он основан) очень трудно. Вместо этого можно обеспечить упрощенную проверку, которая сможет работать со всеми встречающимися примерами. В листинге 8.4 показан пример проверки поля ввода на наличие в нем адреса электронной почты.

Листинг 8.4. Проверка на наличие в определенном элементе ввода адреса электронной почты

```
// Универсальная функция для проверки, похоже ли содержимое элемента ввода
// на адрес электронной почты
function checkEmail( elem ) {
    // Определение, что в поле что-то введено, и что введенное значение
    // похоже на приемлемый адрес электронной почты
    return elem.value == '' ||
        /^[a-z0-9_+.-]+\@([a-z0-9-]+\.)+[a-z0-9]{2,4}$/i.test( elem.value );
}

// Получение проверяемого элемента ввода
var elem = document.getElementById("email");

// Проверка приемлемости содержимого поля
if ( ! checkEmail( elem ) ) {
    alert( "Поле не содержит адреса электронной почты". );
}
```

URL

Во многих формах ввода комментариев (и других сетевых областях) довольно часто запрашивается адрес пользовательского веб-сайта в форме URL. Это еще один пример (наряду с адресом электронной почты) когда очень трудно полностью задать технические требования по его определению. Но существуют и другие ситуации, в которых все, что необходимо на самом деле — это небольшое подмножество полных технических требований. В действительности вам нужны лишь веб-адреса, основанные на протоколах http или https (если нужно что-либо другое, то изменения внести совсем не трудно). Кроме того, для поля URL типичным началом будет строка http://, поэтому вам при проверке формы следует убедиться, что это обстоятельство было принято во внимание. Пример проверки приемлемости предоставленных в форме URL показан в листинге 8.5.

Листинг 8.5. Проверка на наличие URL в элементе ввода

```
// Универсальная функция, предназначенная для проверки наличия URL в
// элементе ввода
function checkURL( elem ) {
    // Определение, что в поле что-то введено, и что введенное значение
    // не является уже введенным по умолчанию текстом http://
    return elem.value == '' || !elem.value == 'http://' ||
    // Определение, что введенное значение похоже на приемлемый URL
    /^https?:\/\:\/\/([a-z0-9-]+\.)+[a-z0-9]{2,4}.*$/i.test( elem.value );
}

// Получение проверяемого элемента ввода
var elem = document.getElementById("url");

// Проверка, содержит ли поле приемлемый URL
if ( ! checkURL( elem ) ) {
    alert( "Поле не содержит URL". );
}
```

Телефонные номера

Теперь мы рассмотрим два разных поля, содержимое которых различается в зависимости от места вашего пребывания: поле телефонных номеров и поле дат. Чтобы упростить задачу, я воспользовался телефонными номерами (и датами) центральной части США; приспособить задачу под условия другой страны не представляет особой сложности.

Теперь рассмотрим несколько вариантов поля телефонных номеров. Номера могут быть записаны несколькими различными способами, следовательно это нужно учесть (например, 123-456-7890, или (123) 456-7890).

Мы собираемся не только проверить приемлемость телефонного номера, но и привести его к определенному формату. Эта задача решается за счет весьма универсального поиска в значении поля телефонного номера, направленного на простое обнаружение двух групп по три цифры и одной группы из четырех цифр, при этом все дополнительное форматирование, введенное пользователем вокруг этих цифр, игнорируется.

Код этой проверки и приведения значения к определенному формату, показан в листинге 8.6.

Листинг 8.6. Проверка поля на наличие телефонного номера

```
// Универсальная функция, предназначенная для проверки наличия в элементе
// ввода телефонного номера
function checkPhone( elem ) {
    // Проверка на наличие чего-либо похожего на приемлемый телефонный
    // номер
    var m = /(\d{3}).*(\d{3}).*(\d{4})/.exec( elem.value );

    // Если похоже что, номер приемлем, приведение его к
```

```

// определенному желаемому формату: (123) 456-7890
if ( m !== null)
    elem.value = "(" + m[1] + ") " + m[2] + "-" + m[3];

return elem.value == '' || m !== null;
}

// Получение проверяемого элемента ввода
var elem = document.getElementById("phone");

// Проверка наличия в поле приемлемого телефонного номера
if ( ! checkPhone( elem ) ) {
    alert( "Поле не содержит телефонного номера". );
}

```

Дата

И теперь нам осталось рассмотреть проверку дат. Мы опять обратимся к Центральноамериканскому формату, на этот раз для даты (ММ/ДД/ГГГГ). И опять, как в случае с телефонными номерами или другими полями, чье заполнение зависит от конкретного региона, регулярное выражение, используемое для проверки, при необходимости может быть легко перенастроено под региональные особенности. Простая проверка содержимого поля даты может быть произведена с помощью функции, показанной в листинге 8.7.

Листинг 8.7. Проверка поля на наличие даты

```

// Универсальная функция, предназначенная для проверки наличия даты в
// элементе ввода
function checkDate( elem ) {
    // Определение, что в поле что-то введено, и что введенное значение
    // похоже на приемлемую дату в формате ММ/ДД/ГГГГ
    return !elem.value || /^\\d{2}\\/\\d{2}\\/\\d{2,4}$/.test(elem.value);
}

// Получение проверяемого элемента ввода
var elem = document.getElementById("date");

// Проверка наличия в поле приемлемой даты
if ( ! checkDate( elem ) ) {
    alert( "Поле не содержит даты". );
}

```

Набор правил

Теперь, используя различные функции проверки из предыдущего раздела, вы можете создать универсальную структуру для работы со всем многообразием проверочных технологий. Важно, чтобы все тесты обрабатывались однообразно, с использованием общих названий и общей семантики сообщений об ошибках. Полная структура набора правил изложена в листинге 8.8.

Листинг 8.8. Стандартный набор правил и описательных сообщений об ошибках для построения основного механизма проверки


```

var errMsg = {
  // Проверка обязательности определенного поля
  required: {
    msg: "Это обязательное поле".,
    test: function(obj,load) {
      // Обеспечение отсутствия в поле введенного текста
      // и задержки начала работы на загружаемой странице
      // (демонстрация сообщения 'Это обязательное поле' при загрузке
      // страницы будет только раздражать пользователя)
      return obj.value.length > 0 || load || obj.value ==
          obj.defaultValue;
    }
  },

  // Определение наличия в поле приемлемого адреса электронной почты
  email: {
    msg: "Введенный адрес неприемлем".,
    test: function(obj) {
      // Определение, что в поле что-то введено, и что введенное
      // значение похоже на адрес электронной почты
      return !obj.value ||
        /^[a-z0-9_+.-]+\@([a-z0-9-]+\.)+[a-z0-9]{2,4}$/i.test(
          obj.value );
    }
  },

  // Определение, что поле содержит телефонный номер, и
  // его автоформатирование в случае положительного результата
  phone: {
    msg: "Введенный номер неприемлем".,
    test: function(obj) {
      // Определение, что введенное значение похоже
      // на приемлемый телефонный номер
      var m = /(\d{3}).*(\d{3}).*(\d{4})/.exec( obj.value );

      // Если похоже на то, что, номер приемлем, приведение
      // его к определенному желаемому формату: (123) 456-7890
      if ( m ) obj.value = "(" + m[1]+") " + m[2]+"-"+m[3];

      return !obj.value || m;
    }
  },

  // Определение, что поле содержит приемлемую дату
  // формата ММ/ДД/ГГГГ
  date: {
    msg: "Дата неприемлема".,
    test: function(obj) {

```

```

    // Определение, что в поле что-то введено, и что введенное
    // значение похоже на приемлемую дату формата ММ/ДД/ГГГГ
    return !obj.value || /^^\d{2}\d{2}\d{2,4}$/.test(obj.value);
  }
},

// Определение, что поле содержит приемлемый URL
url: {
  msg: "URL неприемлем".,
  test: function(obj) {
    // Определение, что в поле введен какой-нибудь текст, и он
    // отличается от уже введенного по умолчанию текста
    // http://
    return !obj.value || obj.value == 'http://' ||
      // Определение, что введенное значение похоже на
      // приемлемый URL
      /^https?:\/\/([a-z0-9-]+\.)+[a-z0-9]{2,4}.*$/i.test(
        obj.value );
  }
}
};

```

Теперь, используя эту новую структуру набора правил, вы можете создать общие, согласованные средства проверки формы и отображения сообщений об ошибках, которые я рассматриваю в следующем разделе.

Отображение сообщений об ошибках

Если процесс проверки данных формы не вызывает особых затруднений, то отображения контекстно-зависимых сообщений об ошибках, способных помочь пользователю более качественно заполнить форму, зачастую вызывает определенные трудности. Теперь все созданное в предыдущем разделе мы собираемся использовать для создания полноценной системы проверки и отображения сообщений об ошибках. Мы собираемся рассмотреть, как осуществляется проверка формы и отображение сообщений об ошибках, и когда все это должно происходить, чтобы пользователь все понял наилучшим образом.

Проверка приемлемости данных

Используя новую структуру данных можно построить согласованную, расширяемую пару функций, которая может быть использована для проверки приемлемости данных формы или отдельного поля, и отображения на основе этой проверки контекстно-зависимого сообщения об ошибке.

Чтобы добиться динамической проверки формы используются две технологии. Первая из них предоставляется браузерами и представляет собой часть HTML DOM-спецификации. Все <form>-элементы (в DOM) обладают дополнительным свойством под названием `elements`. Это свойство содержит массив всех полей, присутствующих в форме, и это значительно облегчает переход по всем имеющимся полям для проверки ошибок ввода.

Вторая важная составляющая заключается во включении во все поля дополнительных классов для переключения различных правил проверки. К примеру, наличие класса — `required` (обязательное) потребует от поля ввода какой-нибудь формы ввода данных. Каждый из классов должен соответствовать тем, которые были предоставлены в наборе правил, показанном в листинге 8.8.

Теперь, используя эти две технологии, вы можете создать две универсальные функции для проверки приемлемости данных всех форм и отдельных полей (и обе эти задачи потребуют полноценного проверочного сценария). Эти две функции показаны в листинге 8.9.

Листинг 8.9. Функции, предназначенные для проверки приемлемости данных, введенных в форму, и отображения сообщений об ошибках

```
// Функция проверки всех полей внутри формы.
// Аргумент form должен быть ссылкой на элемент формы
// Аргумент load должен быть булевой ссылкой на то, что функция проверки
// запускается не в динамическом режиме, а после загрузке страницы
function validateForm( form, load ) {
    var valid = true;

    // Последовательный перебор всех имеющихся в форме элементов полей
    // form.elements — массив всех имеющихся в форме полей
    for ( var i = 0; i < form.elements.length; i++ ) {

        // Скрытие любых сообщений об ошибках, если они были показаны
        hideErrors( form.elements[i] );

        // Проверка содержимого поля на приемлемость
        if ( ! validateField( form.elements[i], load ) )
            valid = false;
    }

    // Возвращение false, если содержимое поля неприемлемо,
    // и true, если значение всех полей приемлемо
    return valid;
}

// Проверка приемлемости содержимого отдельного поля
function validateField( elem, load ) {
    var errors = [];

    // Последовательный перебор всех имеющихся технологий проверки
    // приемлемости
    for ( var name in errMsg ) {
        // Определение, имеет ли поле класс, определенный типом ошибки
        var re = new RegExp("(^|\\s)" + name + "(\\s|$)");

        // Определение, имеет ли элемент класс, и передан ли он тесту
        // на приемлемость данных
        if ( re.test( elem.className ) && !errMsg[name].test( elem, load ) )
            // Если проверка не удалась, добавление сообщения об ошибке к
            // списку
            errors.push( errMsg[name].msg );
    }
}
```

```

// Отображение сообщений об ошибке, если таковые имеются
if ( errors.length )
    showErrors( elem, errors );

// Возвращение false, если поле не прошло какую-нибудь процедуру
// проверки
return errors.length > 0;
}

```

Возможно, в предыдущем коде вы заметили отсутствие двух функций, относящихся к скрытию и отображению сообщений об ошибках, обнаруженных во время проверки. Возможно, эти функции потребуют некоторой настройки в соответствии с вашими желаниями по характеру отображения сообщений об ошибках. Но для данной конкретной формы я избрал вариант отображения сообщений об ошибках внутри самой формы, сразу после отображения каждого из полей. Две функции, предназначенные для выполнения этого замысла, показаны в листинге 8.10.

Листинг 8.10. Функции для отображения и скрытия сообщениях об ошибках, обнаруженных во время проверки определенного поля формы

```

// Скрытие любых отображаемых на данный момент сообщений об ошибках,
// обнаруженных во время проверки
function hideErrors( elem ) {
    // Обнаружение следующего за текущим полем элемента
    var next = elem.nextSibling;

    // Если следующий элемент ul и имеет класс errors
    if ( next && next.nodeName == "UL" && next.className == "errors" )
        // его следует удалить (в нашем смысле – 'скрыть')
        elem.parentNode.removeChild( next );
}

// Отображение набора сообщений об ошибках для определенного поля
// внутри формы
function showErrors( elem, errors ) {
    // Обнаружение следующего за полем элемента
    var next = elem.nextSibling;

    // Если поле не является одним из наших специальных контейнеров
    // для сообщений об ошибке
    if ( next && ( next.nodeName != "UL" || next.className
        != "errors" ) ) {
        // То такое поле нужно создать
        next = document.createElement( "ul" );
        next.className = "errors";

        // а затем вставить его в нужное место в DOM-структуре
        elem.parentNode.insertBefore( next, elem.nextSibling );
    }
}

```

```

// Теперь, имея ссылку на контейнер сообщений об ошибках — UL
// осуществляем последовательный перебор всех сообщений
for ( var i = 0; i < errors.length; i++ ) {
    // создаем новый контейнер li для каждого из них
    var li = document.createElement( "li" );
    li.innerHTML = errors[i];

    // и вставляем его в DOM
    next.appendChild( li );
}
}

```

Теперь, когда работа с кодом JavaScript завершена, осталось лишь добавить сообщениям об ошибках какое-нибудь дополнительное стилевое оформление, чтобы придать им вполне приглядный вид. Код CSS, предназначенный для решения этой задачи, показан в листинге 8.11.

Листинг 8.11. Дополнительный код CSS, предназначенный для придания сообщениям об ошибках соответствующего вида

```

ul.errors {
    list-style: none;
    background: #FFCECE;
    padding: 3px;
    margin: 3px 0 3px 70px;
    font-size: 0.9em;
    width: 165px;
}

```

И наконец, когда все составляющие сложены воедино, можно посмотреть на конечный результат работы JavaScript и таблицы стилей, показанный на рис. 8.2 (полученный после того, как все это было связано с отслеживателями событий, рассматриваемыми в следующем разделе).

Теперь, когда вы точно знаете, как осуществить проверку приемлемости данных формы (и полей, которые в ней содержатся) и отобразить сообщения об ошибках, основанные на любых неудачах, настало время определить момент запуска ваших проверочных процедур. Одновременное проведение проверки всех полей не всегда является лучшим вариантом, зачастую больше подходит ее постепенное проведение. Мы рассмотрим преимущества всех остальных случаев, в которых используется проверка приемлемости в следующем разделе.

Login Information

Username

Password

Personal Information

Name

Email
Not a valid email address.

Date
Not a valid date.

Website
Not a valid URL.

Phone

Over 13?

Рис. 8.2. Пример приемлемого и неприемлемого ввода данных в вашей заново стилизованной и обработанной сценарием форме

Когда следует проводить проверку

Один из наиболее трудных аспектов проведения проверки формы является определение подходящего момента для отображения сообщений об ошибках. Для проверки формы (или поля) есть три различных момента времени: непосредственно перед отправкой данных формы, после внесения изменений в поле, и после загрузки страницы. У каждого из них есть свои преимущества и недостатки, требующие отдельного рассмотрения. Использование разработанных в предыдущем разделе функций упрощает и облегчает осмысление этого процесса.

Проверка, предшествующая отправке данных формы

Наибольшее распространение получила проверка, осуществляемая непосредственно перед отправкой формы, поскольку она лучше всего вписывается в стандартные проверочные технологии. Чтобы отследить момент, наступающий непосредственно перед передачей формы, нужно осуществить привязку обработчика события, который будет ожидать завершения заполнения формы и щелчка на кнопке отправки — Submit (или нажатия клавиши Enter). При этом не ставится обязательное условие, что во все поля пользователь уже ввел какие-нибудь значения, но раз форма направлена на отправку, она проходит проверку на соответствие всем правилам определенного набора. Если какое-то из полей не соответствует какому-нибудь правилу, форма не будет отправлена, и пользователю придется разбираться в предъявленных ему сообщениях об ошибках (что делается за счет подавления исходных действий обработчика события submit). Необходимы для реализации этой технологии код показан в листинге 8.12.

Листинг 8.12. Ожидание события отправки для запуска функции проверки формы

```
function watchForm( form ) {
```

```

// Отслеживание события отправки формы
addEventListener( form, 'submit', function(){

    // Обеспечение прохождения формой проверки приемлемости данных
    return validateForm( form );

});
}

// Обнаружение первой формы на странице
var form = document.getElementsByTagName( "form" )[0];

// и отслеживание ее события отправки для того, чтобы подвергнуть проверке
watchForm( form );

```

Проверка после внесения в поле изменений

Другая технология, применяемая для проверки приемлемости данных формы, заключается в отслеживании внутри отдельных полей формы. Для этого можно воспользоваться событием нажатия клавиши — `keypress`, но это приведет к нежелательным результатам. Постоянное отслеживание ошибки при каждом нажатии клавиши в пределах поля будет сбивать пользователей с толку. Они (к примеру) могут приступить к вводу адреса своей электронной почты, и увидят сообщение об ошибке, в котором утверждается, что их адрес некорректен. Но это будет неверно, поскольку они еще не завершили ввод данных в поле. В общем, нам такая технология не подойдет, поскольку пользователь будет от нее не в восторге.

Второй способ отслеживания изменений в поле заключается в ожидании, пока пользователь его не покинет (в надежде на то, что он ввел всю необходимую информацию). Проведение такой проверки воспринимается пользователем намного легче, поскольку он получает возможность ввести всю желаемую информацию, и ему по прежнему выдается оперативное сообщение об ошибке, вызванной неприемлемостью данных.

Пример реализации этой технологии показан в листинге 8.13.

Листинг 8.13. Отслеживание изменений в поле перед запуском любых проверочных функций

```

function watchFields( form ) {
    // Последовательный перебор всех элементов полей формы
    for ( var i = 0; i < form.elements.length; i++ ) {

        // и прикрепление к ним обработчика события 'change' (отслеживающего
        // потерю фокуса элементом ввода)
        addEventListener( form.elements[i], 'change', function(){
            // Как только фокус утрачен, перепроверка поля
            return validateField( this );
        });
    }
}

// Обнаружение первой формы на странице
var form = document.getElementsByTagName( "form" )[0];

```

```
// Отслеживание изменений во всех полях формы
watchFields( form );
```

Проверка после загрузки страницы

Проверка формы после загрузки страницы не столь востребована, как предыдущие две технологии, но если вы хотите охватить и особые случаи, ее тоже нужно включить в свой арсенал. Если пользователь вводит в форму информацию, а затем повторно загружает окно браузера (или если браузер или само приложение заранее заполняет форму пользовательской информацией), существует вероятность, что ошибка будет допущена в информации, используемой для предварительного заполнения. Эта технология разработана для запуска проверки приемлемости данных формы при каждой загрузке страницы для оценки качества уже введенных данных.

При этом пользователь получает возможность немедленной работы над ошибками, не дожидаясь проверки данных, запущенной по событию их отправки.

Код, необходимый для проверки приемлемости данных формы после загрузки страницы, показан в листинге 8.14.

Листинг 8.14. Осуществление проверки формы после загрузки страницы

```
addEventListener( window, "load", function() {
    // Обнаружение всех форм на странице
    var forms = document.getElementsByTagName("form");

    // Последовательный перебор всех форм на странице
    for ( var i = 0; i < forms.length; i++ ) {

        // Проверка каждой из форм, установив аргумент 'load'
        // в true, чтобы остановить появление определенных, совершенно
        // ненужных ошибок
        validateForm( forms[i], true );
    }
});
```

Освоив всевозможные формы проверки, способы отображения сообщений об ошибках, и даже рассмотрев вопрос, когда следует проводить проверку формы, мы подошли к достойному финишу: полной проверке формы на стороне клиента. После того, как мы справились с этой задачей, можно приступить к исследованию двух дополнительных технологий, повышающих качество работы с формами и с определенными типами ее полей.

Повышение качества работы с формами

Учитывая, что формы относятся к наиболее часто используемым элементам веб-страниц, повышение качества работы с ними принесет пользователю несомненные выгоды. В этом разделе я собираюсь рассмотреть две разные, довольно широко распространенные технологии, которые довольно часто используются для повышения качества работы с формами.

Кроме этого вам представится еще одна возможность воспользоваться библиотекой JavaScript для упрощения трудоемких DOM-перемещений и изменений, необходимых для выполнения намеченной нами задачи. Для двух представленных здесь технологий я выбрал использование JavaScript-библиотеки jQuery (<http://jquery.com/>), которая, в частности, хорошо подходит для осуществления DOM-перемещений и изменений.

Накладные надписи

Первое из рассматриваемых усовершенствований, касается установочных (накладных) надписей поверх связанных с ними полей, и их скрытия, как только соответствующее им поле получает фокус. Эта технология имеет двойное предназначение. Она абсолютно точно объясняет пользователю, что предполагается вводить в конкретное поле (поскольку то, что предполагается в него вводить написано поверх самого поля). И к тому же она позволяет уменьшить общее пространство, необходимое для поля и соответствующей ему надписи.

В нашей исходной форме мы добавим эти две накладные надписи к полям имени пользователя — username и пароля — password, чтобы получить результат, показанный на рис. 8.3.

The image shows a web form with two main sections. The first section, titled 'Login Information', contains two input fields: 'Username' and 'Password'. The labels 'Username' and 'Password' are positioned directly above their respective input fields, overlapping the top edge of the text boxes. The second section, titled 'Personal Information', contains five input fields: 'Name', 'Email', 'Date', 'Website', and 'Phone', each with its label to the left. Below these is a checkbox labeled 'Over 13?'. At the bottom right of the form is a button labeled 'Submit Form'.

Рис. 8.3. Использование накладных надписей для полей имени пользователя и пароля

Код JavaScript, необходимый для получения этого специфического эффекта, будет сравнительно непростым. Для гладкой работы в него потребуется включить множество мелких подробностей. Рассмотрим две особенности, необходимые для получения конечного результата.

Во-первых, чтобы позиционировать надписи поверх самих элементов ввода, сначала нужно поместить и надпись, и элемент ввода в div-контейнер. Этот div используется таким образом, чтобы можно было абсолютно позиционировать надпись поверх поля.

Во-вторых, это нужно сделать так, чтобы когда поле получало или утрачивало фокус, надпись соответственно скрывалась (или отображалась). Кроме этого, когда пользователь покидает поле, нужно проверить, имеет ли оно какое-нибудь значение, и если имеет, больше не показывать надпись.

И наконец, нужно обеспечить, чтобы надпись не появлялась, если значение помещается в поле по умолчанию (иначе у вас получится полная мешанина).

Памятуя обо всем этом, давайте взглянем на код, необходимый для получения накладных надписей внутри формы, который представлен в листинге 8.15.

Листинг 8.15. Накладные надписи, появляющиеся поверх полей, реализованные с помощью JavaScript-библиотеки jQuery

```
// Обнаружение всех элементов ввода, следующих за надписями, имеющими
// класс hover
$("label.hover+input")
```

```

// Заключение элемента ввода в div (имеющий класс hover-wrap),
// чтобы получить следующий HTML:
// <div class='hover-wrap'><input type="text" .../></div>
.wrap("<div class='hover-wrap'></div>")

// Скрытие надписи при каждом получении элементом ввода фокуса
// (либо за счет щелчка, либо за счет клавиатурных манипуляций)
.focus(function(){
    $(this).prev().hide();
})

// Повторный показ надписи при выходе пользователя за пределы
// элемента ввода (без ввода в него какого-либо текста).
.blur(function(){
    if ( !this.value ) $(this).prev().show()
})

// Индивидуальный перебор всех элементов ввода
.each(function(){
    // Внедрение надписи в <div class='hover-wrap'></div>
    $(this).before( $(this).parent().prev() );

    // Обеспечение автоматического скрывтия надписи, если
    // значение уже введено
    if ( this.value ) $(this).prev().hide();
});

```

Но для достижения желаемого результата одного JavaScript недостаточно. Все-таки для того, чтобы надписи и поля встали на правильные позиции, нужно обязательно включить дополнительные стили CSS. Необходимый для этого код показан в листинге 8.16.

Листинг 8.16. Стили CSS, необходимые, чтобы заставить надписи накладываться на связанные с ними поля

```

div.hover-wrap {
    position: relative;
    display: inline;
}

div.hover-wrap input.invalid {
    border: 2px solid red;
}

div.hover-wrap ul.errors {
    display: none;
}

div.hover-wrap label.hover {
    position: absolute;

```

```

top: -0.7em;
left: 5px;
color: #666;
}

```

Вот так, без особого труда, мы создали очень полезное усовершенствование, повышающее качество работы с формой. Использование этой специализированной технологии позволяет убить сразу двух зайцев: сэкономить место на экране, и сохранить необходимые для пользователя указания.

Пометка обязательных полей

Вторая технология, которую мы собираемся рассмотреть, касается пометки обязательных полей каким-нибудь знаком. Большинство веб-разработчиков выбрали для пометки обязательных полей на своих веб-сайтах красную звездочку. Но дополнительная разметка, необходимая для включения этих звездочек, выходит за рамки обычной семантики, и может стать обескураживающим моментом. Но для нас это станет лишь прекрасной возможностью воспользоваться для добавления знака JavaScript. Пример этой технологии показан на рис. 8.4.

The image shows a web form with two main sections. The first section, titled "Login Information", contains two input fields: "Username" and "Password". The second section, titled "Personal Information", contains five input fields: "Name *", "Email *", "Date *", "Website", and "Phone". Below these is a checkbox labeled "Over 13?". At the bottom right of the form is a button labeled "Submit Form".

Рис. 8.4. Результат добавления контекстно-зависимых звездочек к обязательным полям формы

Одним из аспектов добавления этих знаков к надписям обязательных полей заключается в дополнении формы специальным вспомогательным текстом, который служит пояснением для пользователей. Использование атрибута `title` позволит предоставить пользователям сообщение, поясняющее значение красной звездочки (на тот случай, если они с ним не знакомы). В общем, реализация этого усовершенствования не отличается сложностью, и показана в листинге 8.17.

Листинг 8.17. Добавление контекстно-зависимых звездочек (*) и пояснительных сообщений к обязательным полям формы с использованием JavaScript-библиотеки jQuery

```

// обнаружение всех полей ввода, помеченных обязательными (required)
$("input.required")
    // затем обнаружение предшествующей им надписи
    .prev("label")

// Изменение курсора при прохождении над надписью на более
// полезный элемент

```

```

.css("cursor", "help")

// Обеспечение появления поясняющей надписи для звездочки при
// прохождении над ней указателя мыши
.title( errMsg.required )

// И наконец, добавление звездочки (*) к надписи, чтобы обозначить
// обязательное поле
.append(" <span class='required'*</span>");

```

Для получения стилового оформления, необходимо добавить к значку красную расцветку (см. листинг 8.18).

Листинг 8.18. Дополнительный код CSS для стилового оформления звездочки (*)

```

label span.required {
    color: red;
}

```

Добавление пометок и накладных надписей — это существенное повышение качества работы с формами, которое можно получить за счет использования JavaScript в ненавязчивой и полезной форме. Я уверен, что в своих приложениях вы всегда найдете массу возможностей для усовершенствования работы с формами и полями за счет использования простого кода JavaScript.

Выводы

Показав вам сначала ряд аспектов, затрудняющих использование форм в веб-приложениях, я надеюсь, что сумел поднять ваше настроение научив, как с помощью простого дополнительного кода на JavaScript существенно улучшить общее качество работы с формами. Пример всего, что удалось достичь в этой главе, показан на рис. 8.5.

Login Information

Personal Information
Name *
Email *

Date *

Website

Phone
Over 13?

Рис. 8.5. Законченный вид формы, усовершенствованной за счет JavaScript

Мы начали главу с рассмотрения, как наиболее точно провести проверку приемлемости данных на стороне клиента, оставив при этом у пользователя наилучшие впечатления от ее работы. Этого удалось добиться за счет создания набора проверочных правил, проверки приемлемости данных в полях формы в самые подходящие для этого моменты, и отображения полезных для пользователя сообщений об ошибках. Вдобавок мы рассмотрели две технологии, повышающие качество работы пользователя с формой за счет накладных надписей и пометки обязательных полей.

Я надеюсь, что все вместе взятые представленные здесь технологии, могут быть широко задействованы вами в тех формах, которые вам еще предстоит разработать.

Глава 9 Создание галерей изображений

Работа с DOM, перемещение по элементам и динамическое использование CSS — все это направлено на то, что бы создать у конечного пользователя положительное впечатление о веб-сайте о простоте и легкости управления его работой. Галерея изображений (позволяющая осуществлять из просмотр и перебор) является одним из тех приложений, которые получают от применения этих технологий явное преимущество. Повысившееся качество браузеров позволяет использовать динамические сценарии и утилиты. В последнее время эти усовершенствования привели к созданию ряда высококачественных галерей изображений.

В этой главе мы рассмотрим две такие галереи, и посмотрим, в чем заключается их конкретная уникальность, а затем создадим свою собственную галерею, используя динамический, ненавязчивый код JavaScript. При этом будет рассмотрен ряд вопросов, касающихся подробностей конструкции и реализации галереи. Конечным результатом станет эффективно работающий сценарий создания галереи изображений, который может без труда размещен на любом веб-сайте. Кроме этого у нас появится отличная возможность для применения функций, разработанных в пятой и седьмой главах, в которых для создания простого и качественного кода будет использована совместная работа DOM, JavaScript и CSS.

Примеры галерей

Имеется несколько превосходных, современных сценариев галерей изображений, оставляющих яркое впечатление, удобных в работе и полностью отвечающих принципам ненавязчивости. Два сценария, которые мы собираемся рассмотреть подробнее, обладают очень похожими визуальными эффектами, но используют в основе своего кода разные библиотеки.

Работу галерей можно кратко представить следующим образом:

- Когда на одном из изображений галереи происходит щелчок мышью, вместо перенаправления пользователя на просмотр этого изображения, происходит его наложение на экран.
- Когда отображается наложенное на экран изображение, поверх страницы накладывается прозрачный серый фильтр (снижая яркость всего, что под него попало).
- Отображаемое в данный момент изображение каким-либо образом помечается в галерее изображений.
- Существует какой-нибудь способ перехода по галерее от одного изображения к другому.

В этом разделе мы собираемся рассмотреть галереи, которые создаются двумя очень известными библиотеками — Lightbox и ThickBox.

Lightbox

Lightbox является первой из DOM-галерей «нового стиля». Ее выпуск подтолкнул на создание ряда других галерей подобного стиля, положенных в основу материалов этой главы.

Эта галерея разрабатывалась с нуля (без использования в качестве основы какой-то конкретной JavaScript-библиотеки). Но с тех пор она была приспособлена к использованию различных библиотек (что привело к сокращению общего объема ее кода). Дополнительная информация об этом сценарии может быть найдена по адресам <http://www.huddletogether.com/projects/lightbox/> и <http://particletree.com/features/lightbox-gone-wild/>, где представлены сведения о Lightbox, использующей JavaScript-библиотеку Prototype.

На рис. 9.1 показан пример копии экрана галереи Lightbox в действии, с ее уникальным наложением прозрачного затемнения и размещенным по центру изображением.



Рис. 9.1. Lightbox отображает отдельное изображение, принадлежащее галереи

Lightbox всецело работает в ненавязчивой манере. Чтобы воспользоваться этой библиотекой нужно просто включить сценарий в заголовок вашего HTML-файла и модифицировать HTML изображения, которые вы хотите отображать с ее использованием, и тогда код сценария сделает все остальное:

```
<a href="images/image-1.jpg" rel="lightbox" title="сопроводительная  
надпись">фото №1</a>
```

К сожалению, ненавязчивая природа кода не столь совершенна, как могла бы быть (вместо того, чтобы отслеживать готовность DOM, библиотека ожидает загрузки всех изображений). Тем не менее, задействованные DOM-сценарии (см. листинг 9.1) совершенно рациональны и пригодны к использованию.

Листинг 9-1. Обнаружение всех элементов ссылки (anchor) и их преобразование для правильного отображения

```
// Обнаружение на странице всех тегов anchor
var anchors = document.getElementsByTagName("a");

// Последовательный перебор всех тегов anchor
for ( var i=0; i < anchors.length; i++ ) {
    var anchor = anchors[i];

    // Проверка на принадлежность ссылки к "lightbox"
    if ( anchor.href && anchor.rel == "lightbox" ) {
```

```

// Обеспечение отображения Lightbox по щелчку
anchor.onclick = function () {
    showLightbox(this);
    return false;
};
}
}

```

Под влиянием отзывов новых пользователей Lightbox постепенно развивалась, в нее добавлялись новые свойства, к примеру, переходы с помощью клавиатуры и анимационные эффекты. Тем не менее в своем самом простом варианте, Lightbox вдохновляет на создание вашей собственной подобной галереи изображений.

ThickBox

Вторая галерея изображений, которую мне захотелось вам показать — это ThickBox, созданная Коди Линдли (Cody Lindley), в которой используется JavaScript-библиотека jQuery. Эта реализация очень похожа на Lightbox, но значительно меньше по размерам и поддерживает с использованием Ajax загрузку внешних HTML-файлов. Дополнительные сведения об этой библиотеке могут быть найдены на его веб-сайте (<http://codylindley.com/Javascript/257/thickbox-one-box-to-rule-them-all>), могут быть найдены и примеры, демонстрирующее ее в действии (<http://jquery.com/demo/thickbox/>).

Как видно на копии экрана (см. рис. 9.2), результат отображения картинок с помощью ThickBox очень похож на результаты работы Lightbox. Как и в Lightbox, в ThickBox используются ненавязчивые средства ее загрузки и выполнения. После простого включения сценария в заголовок страницы, он пройдет по всей DOM-структуре и отыщет все ссылки, имеющие класс thickbox, подобные показанной в следующем коде:

```
<a href="ajaxLogin.htm?height=100&width=250" class="thickbox">ThickBox login</a>
```




Рис. 9.2. ThickBox отображает отдельное изображение поверх всей остальной страницы

В листинге 9.2 показан код, который ThickBox использует для динамического и ненавязчивого применения своих функциональных возможностей, как только DOM будет готов к работе (что произойдет до того, как будут загружены все имеющиеся на странице изображения, создавая у пользователей более благоприятное впечатление).

Листинг 9.2. Применение функциональных возможностей ко всем элементам anchor, имеющим класс "thickbox"

```
// Обнаружение всех элементов thickbox, когда DOM будет готов к работе
$(document).ready(function(){

// добавление thickbox к элементам href, имеющим класс .thickbox
$("a.thickbox").click(function(){
// Определение надписи для thickbox
var t = this.title || this.name || this.href || null;

// Отображение thickbox
TB_show(t,this.href);

// Удаление фокуса со ссылки
this.blur();

// Обеспечение блокировки обычной работы ссылки
```

```

    return false;
  });
});

```

Благодаря количеству дополнительных возможностей, включенных в ThickBox, а также более компактному коду, эта библиотека несомненно предпочтительнее Lightbox.

Далее мы собираемся рассмотреть, как создать свой собственный клон галереи, с учетом всех хитросплетений, необходимых для того, чтобы выполнить эту работу вполне корректно.

Создание галереи

Первым шагом к созданию галереи изображений должен стать набор изображений, по которым можно будет осуществлять переходы. Я собираюсь выдвинуть предположение, что на странице может быть любое количество галерей, и в каждой галерее может быть любое количество изображений. Кроме этого очень важно, чтобы перед любым применением вашего JavaScript, изображения были отображены в понятной и семантически непротиворечивой манере. Это поможет обеспечить пользователям, имеющим выключенный JavaScript (или имеющим ущербную поддержку CSS) получать тем не менее вполне приемлемое впечатление от страницы.

Основной HTML-код, который мы собираемся использовать для нашей галереи, показан в листинге 9.3.

Листинг 9.3. Основной HTML-код страницы, содержащей нашу галерею изображений

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html>
<head>
  <title>Произвольные изображения кошек</title>
</head>
<body>
  <h1>Произвольные изображения кошек</h1>

  <p>Lorem ipsum dolor . . . </p>

  <!--
    Наша галерея, которая должна содержать <ul>, class которого
    равен "gallery", а title связан с этой галереей.
  --->
  <ul class="gallery" title="Произвольные изображения кошек">

    <!--
      Каждое изображение галереи должно быть заключено в элементы
      <li>, и иметь ссылку, указывающую на текущее место изображения.
      Если изображение большое, в него к тому же нужно включить class
      "tall".
    -->

    <li><a href="image1.jpg"></a></li>
    <li><a href="image2.jpg"></a></li>

```

```

<li class="tall"><a href="image3.jpg"></a></li>
<li class="tall"><a href="image4.jpg"></a></li>
<li><a href="image5.jpg"></a></li>

</ul>

<p>Lorem ipsum dolor . . . </p>
</body>
</html>

```

Затем нужно применить к изображениям дополнительное стилевое оформление, чтобы получить более приглядный вид и более наглядное перемещение. Соответствующий код CSS показан в листинге 9.4 9-4.

Листинг 9.4. CSS для соответствующего стилевого оформления страницы

```

body {
  font-family: Arial;
  font-size: 14px;
}

/* Включает красивое обрамление вокруг галереи изображений. */
ul.gallery {
  list-style: none;
  padding: 5px;
  background: #EEE;
  overflow: auto;
  border: 1px solid #AAA;
  margin-top: 0px;
}

/* Создает вокруг каждого изображения прямоугольник стандартной ширины и высоты. */
ul.gallery li {
  float: left;
  margin: 6px;
  width: 110px;
  height: 110px;
  background: #FFF;
  border: 2px solid #AAA;
}

/* Изображения горизонтального формата шириной 100px */
ul.gallery img {
  width: 100px;
  margin: 5px;
  border: 0px;
  margin-top: 17px;
}

```

```

}

/* Изображения вертикального формата высотой 100px */
ul.gallery li.tall img {
    height: 100px;
    width: auto;
    margin-top: 5px;
    margin-left: 17px;
}

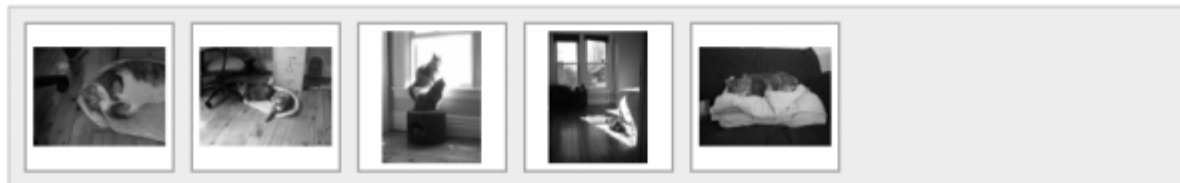
```

Получившийся результат работы базового HTML и CSS показан на рис. 9.3. Теперь, после установки на странице базового HTML, мы готовы приступить к сведению воедино компонентов, необходимых для создания красивой, приводимой в действие кодом JavaScript галереи изображений.

luctus, massa vitae placerat rhoncus, neque mauris lobortis enim, id ullamcorper dui nunc et enim. Integer faucibus rhoncus elit. Vestibulum ut ante. Morbi a sem. Vivamus accumsan.

Maecenas bibendum tellus at ante. Maecenas pharetra volutpat mauris. Vivamus vulputate. Pellentesque nec pede. Pellentesque aliquet, tellus nec placerat bibendum, lacus augue mattis tellus, id iaculis enim augue ac sapien. Maecenas commodo ante quis tellus. Phasellus tempor, massa non malesuada cursus, tortor justo mollis diam, eget sagittis dolor ipsum at velit. Curabitur at pede eu magna sollicitudin accumsan. Duis quis velit nec justo condimentum aliquet. Mauris dignissim mi. Phasellus non mauris. Aliquam sagittis blandit magna.

Etiam quam. Sed nisi. Maecenas viverra pellentesque ante. Fusce vulputate porta metus. Maecenas turpis uma, porta vitae, tempor vel, dapibus vestibulum, felis. Pellentesque sit amet nisl. Curabitur blandit. Sed in nisl et neque condimentum lacinia. Morbi arcu dui, dignissim in, consectetur quis, gravida ac, felis. Proin lacinia aliquet augue. Mauris nec odio. Vestibulum eu orci nec ligula consequat rhoncus. Phasellus nunc nunc, vulputate posuere, semper in, cursus nec, orci. Curabitur sem justo, ullamcorper vitae, adipiscing in, malesuada vel, elit. Maecenas eleifend, justo sed ornare consequat, sem ipsum pellentesque turpis, in rutrum orci neque sed massa. Vestibulum laoreet dui eget arcu. Quisque in lorem. Morbi quis lacus. Maecenas nonummy metus auctor nunc.



Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nullam at sem. Ut ultrices, dolor sit amet pharetra pretium, leo metus accumsan uma, eu ultrices purus augue at ligula. Nam gravida. Mauris neque lacus, vehicula et, hendrerit nec, semper quis, nibh. Nulla facilisi. Curabitur id ante. Proin felis ipsum, commodo non, hendrerit et, ultrices in, eros. Nulla tincidunt vulputate felis. Morbi convallis libero sed erat. Etiam lectus pede, dictum et, rutrum id, commodo nec, augue. Vestibulum massa. Fusce nec felis. Vivamus consectetur elementum elit.

Etiam mattis egestas felis. Proin sed magna at orci aliquet mattis. Vivamus ligula metus, interdum quis, viverra nec, vehicula vitae, mauris. Pellentesque venenatis pede vel augue. Aliquam fringilla nunc eget eros. Pellentesque eget ipsum. Morbi augue. Praesent sit amet ipsum. Ut vitae sem et diam mattis imperdiet. Phasellus a pede. Suspendisse potenti. Maecenas in sapien sit amet uma ultrices dignissim. Duis aliquet faucibus odio. Nunc a turpis vitae sem porta viverra. Vestibulum convallis dignissim nisl. Fusce enim. Morbi quam. Praesent tortor. Suspendisse eu dolor. Ut tempus tortor in dui.

Aenean justo. Integer egestas pharetra quam. Nulla faucibus, mi vel mollis tempus, tortor libero fringilla odio, ac iaculis eros quam eget lorem. Pellentesque nec enim. Cras porta. Duis ornare libero vel risus. In hac habitasse platea dictumst. Quisque luctus, massa vitae placerat rhoncus, neque mauris lobortis enim, id ullamcorper dui nunc et enim. Integer faucibus rhoncus elit. Vestibulum ut ante. Morbi a sem. Vivamus accumsan.

Maecenas bibendum tellus at ante. Maecenas pharetra volutpat mauris. Vivamus vulputate. Pellentesque nec pede. Pellentesque aliquet, tellus nec placerat bibendum, lacus augue mattis tellus, id iaculis enim augue ac sapien. Maecenas commodo ante quis tellus. Phasellus tempor, massa non malesuada cursus, tortor justo mollis diam, eget sagittis dolor ipsum at velit. Curabitur at pede eu magna sollicitudin accumsan. Duis quis velit nec justo condimentum aliquet. Mauris dignissim mi. Phasellus non mauris. Aliquam sagittis blandit magna.

Рис. 9.3. Веб-страница, содержащая изображения с простым стилевым оформлением

Ненавязчивая загрузка

Одной из сторон, которой согласно нашим требованиям должна обладать галерея изображений, должна быть ненавязчивая работа сценария. Нам не хотелось бы заставлять пользователей сценария включать какой-нибудь лишней (и не отвечающей семантике) HTML на их веб-страницы, только ради того, чтобы придать им

более приглядный вид. Поэтому мы собираемся начать со вставки некоторого количества HTML-элементов в DOM веб-страницы, как только закончится ее загрузка. Соответствующий код показан в листинге 9.5.

Листинг 9.5. Вставка исходного HTML в DOM и привязка всех необходимых обработчиков событий к каждому элементу

```
// Отслеживания изображения, которое рассматривается в данный момент
var curImage = null;

// Ожидание окончания загрузки страницы перед тем, как вносить
// изменения или перемещаться по DOM
window.onload = function() {
    /*
    * создание следующей DOM-структуры:
    * <div id="overlay"></div>
    * <div id="gallery">
    * <div id="gallery_image"></div>
    * <div id="gallery_prev"><a href="">&laquo; Предыдущее</a></div>
    * <div id="gallery_next"><a href="">Следующее &raquo;</a></div>
    * <div id="gallery_title"></div>
    * </div>
    */

    // Создание контейнера для всей галереи
    var gallery = document.createElement("div");
    gallery.id = "gallery";

    // И добавление в него всех управляющих контейнеров div
    gallery.innerHTML = '<div id="gallery_image"></div>' +
        '<div id="gallery_prev"><a href="">&laquo; Предыдущее</a></div>' +
        '<div id="gallery_next"><a href="">Следующее &raquo;</a></div>' +
        '<div id="gallery_title"></div>';

    // Добавление gallery в DOM
    document.body.appendChild( gallery );

    // Поддержка обработки каждой ссылки Следующее и Предыдущее
    // на которых был щелчок в пределах галереи
    id("gallery_next").onclick = nextImage;
    id("gallery_prev").onclick = prevImage;

    // Обнаружение всех галерей на странице
    var g = byClass( "gallery", "ul" );

    // Последовательные перебор всех галерей
    for ( var i = 0; i < g.length; i++ ) {
        // и обнаружение всех ссылок на демонстрируемые изображения
        var link = tag( "a", g[i] );
```

```

// Последовательный перебор ссылок на изображения
for ( var j = 0; j < link.length; j++ ) {
    // Обеспечение, чтобы по щелчку на ссылке вместо перехода
    // к изображению осуществлялась демонстрация галереи
    // изображений
    link[j].onclick = function(){
        // Отображение серого фона
        showOverlay();

        // Отображение изображения в галерее
        showImage( this.parentNode );

        // Блокировка обычных действий браузера по переходу на
        // изображение
        return false;
    };
}
// Добавление к галерее средств перехода в режиме демонстрации
// изображений
addSlideShow( g[i] );
}
};

```

Позабывшись об этом важном шаге, вы можете приступить к созданию различных компонентов самой галереи.

Наложение затемнения

Сначала мы собираемся создать наложение затемнения, которое используется как Lightbox, так и в ThickBox. Мы собираемся убедиться в том, что по большей части эта довольно легкая задача, имеющая только один сложный аспект: налагаемое затемнения должно соответствовать высоте и ширине текущей страницы. Но для нас все сложилось удачно, ведь в главе 7 мы уже разрабатывали необходимые для этого функции: `pageWidth()` и `pageHeight()`.

Начнем с создания простого `div`-элемента, имеющего атрибут ID (чтобы позже можно было получить к нему доступ) и добавления его к DOM. Все это показано в листинге 9.6.

Листинг 9.6. Создание простого `div`-элемента и добавление его к DOM

```

// Создание полупрозрачного затемнения
var overlay = document.createElement("div");
overlay.id = "overlay";

// Обеспечение скрытия фона и галереи по щелчку на сером фоне
overlay.onclick = hideOverlay;

// Добавление затемнения к DOM
document.body.appendChild( overlay );

```

Далее нужно создать две функции, необходимые для скрытия и проявления затемнения. Именно здесь и возникают сложности. Сам процесс скрытия и проявления сравнительно прост, но определение правильной ширины и высоты затемнения — дело непростое. В обычной ситуации было бы достаточно определить размеры высоты и ширины в 100%, но этот подход неприемлем, поскольку пользователь мог для отображения галереи прокрутить страницу вниз (и сдвинуть затемнение). Это можно исправить, если сделать высоту и ширину затемнения как и у всей страницы. Чтобы определить нужные параметры, можно воспользоваться функциями `pageHeight()` и `pageWidth()`, разработанными в главе 7.

Полный код для скрытия и проявления затемнения показан в листинге 9.7.

Листинг 9.7. Две функции, необходимые для скрытия и проявления затемнения, используемого в галерее изображений

```
// Скрытие затемнения и текущей галереи
function hideOverlay() {
    // Обеспечение перезапуска значения текущего изображения
    curImage = null;

    // и скрытия затемнения и галереи
    hide( id("overlay") );
    hide( id("gallery") );
}

// Проявление затемнения
function showOverlay() {
    // Обнаружение затемнения
    var over = id("overlay");

    // Установка его размеров по высоте и ширине текущей страницы
    // (что будет полезным при использовании прокрутки)
    over.style.height = pageHeight() + "px";
    over.style.width = pageWidth() + "px";

    // и проявление
    fadeIn( over, 50, 10 );
}
```

И наконец, давайте присовокупим к этому CSS, необходимый для правильного отображения затемнения. Соответствующий код показан в листинге 9.8.

Листинг 9.8. Код CSS, необходимый для правильного отображения затемнения

```
#overlay {
    background: #000;
    opacity: 0.5;
    display: none;
    position: absolute;
    top: 0px;
    left: 0px;
    width: 100%;
```

```

height: 100%;
z-index: 100;
cursor: pointer;
cursor: hand;
}

```

Собранные воедино, встраиваемый HTML и CSS приведут к внешнему виду страницы, показанному на рис. 9.4.

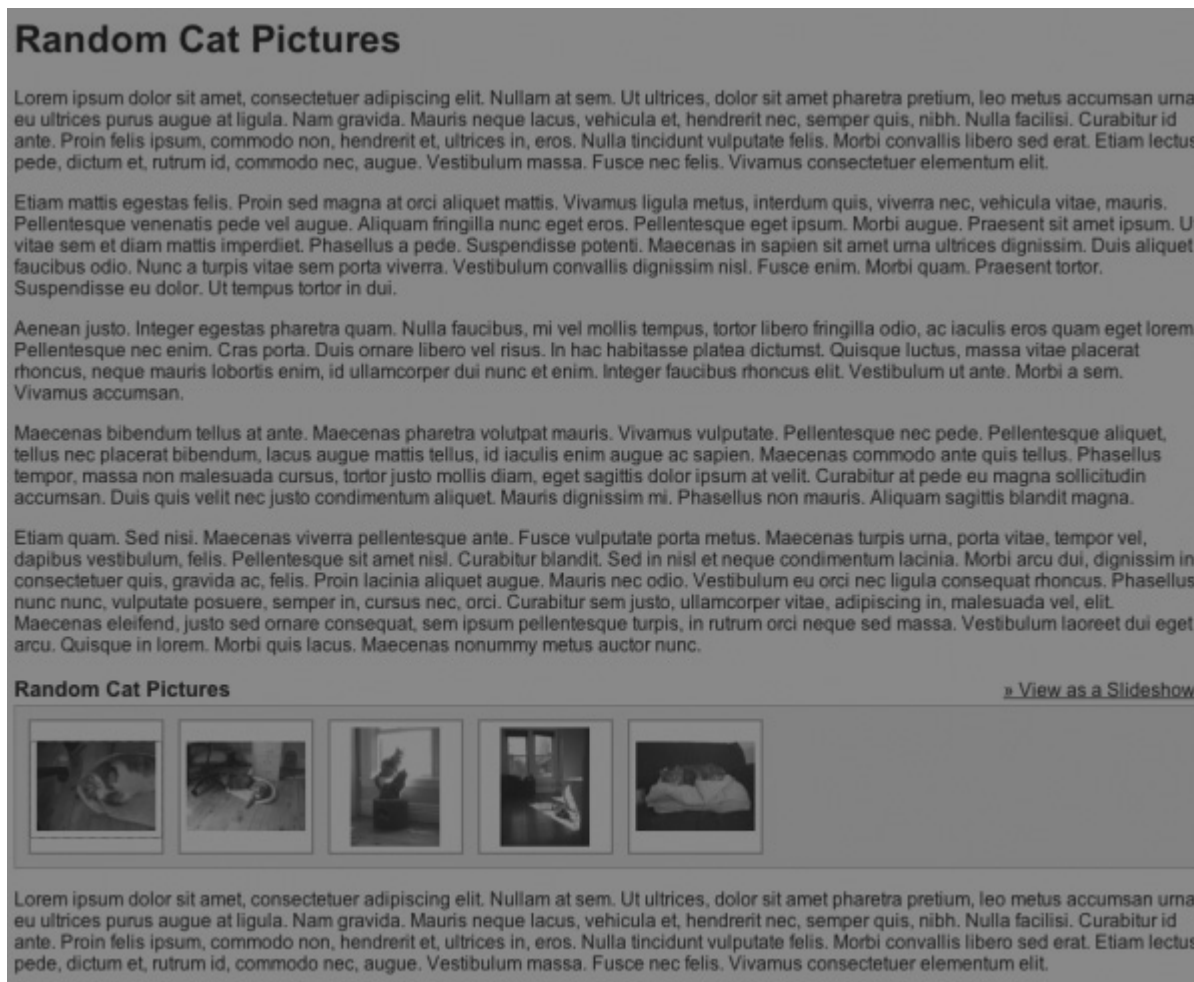


Рис. 9.4. Результат отображения затемнения, наложенного поверх страницы

Создав затемнение, и вставив его в страницу, вы будете готовы приступить к отображению изображения поверх этого затемнения.

Позиционируемый контейнер

Нам нужно создать компонент галереи изображений, который станет контейнером, всплывающим над затемнением и содержащим текущее изображение. К сожалению, из-за недостаточной поддержки CSS 2 в некоторых современных браузерах, реализация этого шага представляет определенные трудности. При должной поддержке CSS, изображение просто будет иметь фиксированную позицию (создавая тем самым иллюзию позиционирования поверх всего остального, независимо от текущей позиции прокрученной страницы).

Начнем с предположения, что у нас на странице уже есть DOM-структура вроде той, что показана в листинге 9.9 (и что вы ее уже добавили к той странице, что была показана в листинге 9.5).

Листинг 9.9. HTML для отображения позиционированной галереи изображений поверх затемнения


```

<div id="gallery">
  <div id="gallery_image"></div>
  <div id="gallery_prev"><a href="">&laquo; Предыдущее</a></div>
  <div id="gallery_next"><a href="">Следующее &raquo;</a></div>
  <div id="gallery_title"></div>
</div>

```

Обладая основной HTML-структурой, нужно создать соответствующую функцию для проявления div-контейнера галереи и добавления к нему изображения. Существует несколько способов запуска такой функции, но наиболее очевидным будет запуск при щелчке пользователя на одном из изображений вашей галереи (показанном на основной HTML-странице), после которого поверх всего остального проявляется его увеличенная версия. Функция, осуществляющая такое отображение, показана в листинге 9.10.

Листинг 9.10. Показ галереи на основе выбранного изображения

```

// Отображение текущей галереи изображений
function showImage(cur) {
  // Запоминание текущего рабочего изображения
  curImage = cur;

  // Обнаружение изображения галереи
  var img = id("gallery_image");

  // Удаление изображения, если таковое уже было отображено
  if ( img.firstChild )
    img.removeChild( img.firstChild );

  // и добавление вместо него нового изображения
  img.appendChild( cur.firstChild.cloneNode( true ) );

  // Установка надписи изображения галереи в качестве содержимого
  // аргумента 'alt' обычного изображения
  id("gallery_title").innerHTML = cur.firstChild.firstChild.alt;

  // Обнаружение основной галереи
  var gallery = id("gallery");

  // Установка правильного класса (чтобы был получен правильный размер)
  gallery.className = cur.className;

  // Постепенное проявление
  fadeIn( gallery, 100, 10 );

  // Обеспечение позиционирования галереи в правильном месте
  // экрана
  adjust();
}

```

Последним шагом в функции showImage вызывается функция adjust. Эта функция отвечает за перепозиционирование галереи изображений на точный центр пользовательского окна (даже если пользователь

осуществил прокрутку или изменил размеры окна). Этот важный шаг, который позволяет галереи выглядеть и вести себя вполне естественно, показан в листинге 9.11.

Листинг 9.11. Перепозиционирование галереи на основе высоты и ширины изображения, и того места, до которого была осуществлена пользовательская прокрутка

```
// Перепозиционирование галереи по центру видимой части страницы
// даже после ее прокрутки
function adjust(){
    // Обнаружение галереи Locate the gallery
    var obj = id("gallery");

    // Определение существования галереи
    if ( !obj ) return;

    // Определение ее текущей высоты и ширины
    var w = getWidth( obj );
    var h = getHeight( obj );

    // Вертикальное позиционирование контейнера по середине окна

    var t = scrollY() + ( windowHeight() / 2 ) - ( h / 2 );

    // Но не выше верхней части страницы
    if ( t < 0 ) t = 0;

    // Горизонтальное позиционирование контейнера по середине окна
    var l = scrollX() + ( windowWidth() / 2 ) - ( w / 2 );

    // Но не левее, чем левый край страницы
    if ( l < 0 ) l = 0;

    // Установка выверенной позиции элемента
    setY( obj, t );
    setX( obj, l );
};

// Корректировка позиции галереи после каждого применения прокрутки страницы
// или изменения размеров окна браузера
window.onresize = document.onscroll = adjust;
```

И наконец, в листинге 9.12 показан код CSS, необходимый для сохранения правильного позиционирования галереи. Можно заметить, что на самом деле это ничто иное, как абсолютно спозиционированный div-контейнер с большим показателем стилового свойства z-index, позволяющим поместить его поверх всего остального, что есть на странице.

Листинг 9.12. Код CSS для правильного позиционирования галереи

```
#gallery {  
    position: absolute;  
    width: 650px;  
    height: 510px;  
    background: #FFF;  
    z-index: 110;  
    display: none;  
}
```

```
#gallery_title {  
    position: absolute;  
    bottom: 5px;  
    left: 5px;  
    width: 100%;  
    font-size: 16px;  
    text-align: center;  
}
```

```
#gallery img {  
    position: absolute;  
    top: 5px;  
    left: 5px;  
    width: 640px;  
    height: 480px;  
    border: 0px;  
    z-index: 115;  
}
```

```
#gallery.tall {  
    width: 430px;  
    height: 590px;  
}
```

```
#gallery.tall img {  
    width: 420px;  
    height: 560px;  
}
```

Теперь, добившись совместной работы CSS, HTML и JavaScript, мы получили позиционированную галерею изображений, которая, судя по рис. 9.5, выглядит довольно неплохо.



Рис. 9.5. Позиционированная галерея изображений, помещенная поверх затемнения

После того как важный шаг по созданию красивой галереи изображений уже позади, нужно сосредоточить усилия на облегчении пользователю переходов по различным изображениям галереи.

Переходы

Когда изображение отображено поверх всей остальной страницы (а между ним и страницей наложено затемнение), ко всему этому нужно добавить лучшие средства для осуществления переходов между различными изображениями галереи. Чуть раньше, когда мы задавали HTML для затемнения галереи, туда были включены ссылки, которые можно было использовать для осуществления переходов. Эти ссылки позволяют пользователю перемещаться по галерее взад-вперед, сохраняя затемнение.

Отслеживая, какое именно изображение просматривается в данный момент, вы можете добавить это функциональное свойства к своей галерее (в данном случае ссылка на изображение хранится в переменной `curImage`). Опирируя этими сведениями, можно легко определить, в какое изображение галереи просматривается, и осуществить перемещение в нужном пользователю направлении (см. листинг 9.13).

Листинг 9.13. Две функции, необходимые для того, чтобы направить пользователей на желаемую ими позицию в галерее

```
// Обнаружение и отображение предыдущего изображения
function prevImage() {
```

```

// Определение местоположения и демонстрация предыдущего изображения
// галереи
showImage( prev( curImage ) );

// Блокировка обычных действий ссылки
return false;
}

// Обнаружение и отображение следующего изображения
function nextImage() {
    // Определение местоположения и демонстрация следующего изображения
    // галереи
    showImage( next( curImage ) );

    // Блокировка обычных действий ссылки
    return false;
}

```

Особенность ссылок перемещений состоит в том, что нужно определить, когда именно уместно вывести их отображение. Нужно обеспечить, чтобы ссылки показывались только тогда, когда в галерее до и после текущего изображения есть изображения, на которые можно перейти. Контекстуальное скрытие или появление ссылок в галерее изображений будет использовано из функции `showImage()` для отображения этих ссылок там, где это уместно. Код, управляющий состоянием средств управления переходами показан в листинге 9.14.

Листинг 9.14. Определение, когда должны быть показаны или скрыты ссылки перехода Следующее и Предыдущее

```

// Скрытие ссылки Следующее, если мы дошли до конца показа
if ( !next(cur) )
    hide( id("gallery_next" ) );

// Если нет, обеспечение ее показа
else
    show( id("gallery_next" ) );

// Скрытие ссылки Предыдущее, если мы дошли до начала показа
if ( !prev(cur) )
    hide( id("gallery_prev" ) );

// Если нет, обеспечение ее показа
else
    show( id("gallery_prev" ) );

```

И наконец, в листинге 9.15 показан код CSS, необходимый для правильного позиционирования ссылок перехода.

Листинг 9.15. Код CSS, необходимый для позиционирования ссылок перехода

```

#gallery_prev, #gallery_next {

```

```

position: absolute;
bottom: 0px;
right: 0px;
z-index: 120;
width: 60px;
text-align: center;
font-size: 12px;
padding: 4px;
}

#gallery_prev {
  left: 0px;
}

#gallery_prev a, #gallery_next a {
  color: #000;
  text-decoration: none;
}

```

На рис. 9.5 показан пример использования ссылок перехода. Обратите внимание, что в нижней части галереи изображений показана ссылка, направляющая пользователя на просмотр следующего изображения галереи. Ссылка показывается и скрывается в соответствии с тем, в каком месте галереи находится просматриваемое пользователем изображение.

Демонстрация изображений

Финальная часть создания галереи изображений придаст ей черты привлекательности, которые могут обрадовать многих пользователей: динамическая, изящная демонстрация всех изображений галереи. По сравнению с предыдущей работой по созданию переходов по изображениями, это дополнение не потребует особых усилий. Все очень просто. Процесс создания демонстрационного режима разбивается на два шага:

- Создание в документе дополнительной ссылки, по щелчку на которой пользователь мог бы запустить демонстрацию.
- Создание демонстрационного процесса как такового (управляющего тем, какие изображения показывать, и как переключаться между ними).

Первый шаг показан в листинге 9.16.

Листинг 9.16. Добавление к DOM дополнительного навигационного элемента для запуска демонстрации изображений

```

function addSlideshow( elem ) {
  // Мы собираемся создать некоторую дополнительную контекстную
  // информацию, сопровождающую демонстрацию

  // создание заголовка демонстрации и его контейнера
  var div = document.createElement("div");
  div.className = "slideshow";
}

```

```

// Отображение имени демонстрации на основе названия галереи
var span = document.createElement("span");
span.innerHTML = g[i].title;
div.appendChild( span );

// Создание ссылки, позволяющей увидеть демонстрацию всех
// изображений галереи
var a = document.createElement("a");
a.href = "";
a.innerHTML = "&raquo; Просмотреть демонстрацию изображений";

// Обеспечение запуска демонстрации по щелчку на ссылке
a.onclick = function(){
    startShow( this.parentNode.nextSibling );
    return false;
};

// Добавление нового управляющего элемента и заголовка к странице
div.appendChild( a );
elem.parentNode.insertBefore( div, elem );
}

```

А теперь настало время создать средство управления всей серией демонстрационной анимации. Все управление построено на серии временных задержек, инициализация которых происходит одновременно (поскольку они настроены так, что их сроки истекают с разном по времени). Конечный результат выражается в плавной, изящной демонстрации, создающей весьма цельное представление. Код, запускающий демонстрацию, показан на рис. 9.17.

Листинг 9.17. Код, запускающий демонстрацию конкретной галереи

```

// Запуск демонстрации всех изображений, находящихся в конкретной галерее
function startShow(obj) {
    // Определение местонахождения всех отдельных изображений галереи
    var elem = tag( "li", obj );

    // Определение местонахождения всей демонстрируемой галереи
    var gallery = id("gallery");

    // Последовательный перебор всех, принадлежащих галереи изображений
    for ( var i = 0; i < elem.length; i++ ) new function() {
        // Запоминание, на какой текущий элемент была ссылка
        var cur = elem[i];

        // Мы собираемся показывать новое изображение каждые 5 секунд
        setTimeout(function(){
            // Отображение отдельного изображения
            showImage( cur );

            // И начала его растворения после 3,5 секунд

```

```

        // (со временем растворения в 1 секунду)
        setTimeout(function(){
            fadeOut( gallery, 0, 10 );
        }, 3500 );
    }, i * 5000 );
};

// А затем скрытие затемнения, когда все закончится
setTimeout( hideOverlay, 5000 * elem.length );

// Но появление затемнения при запуске демонстрации
showOverlay();
}

```

В завершение нужно не забыть добавить код CSS для ссылки на запуск демонстрации. Соответствующий код показан в листинге 9.18.

Листинг 9.18. Дополнительный код CSS для отображения ссылки на запуск демонстрации

```

div.slideshow {
    text-align: right;
    padding: 4px;
    margin-top: 10px;
    position: relative;
}

div.slideshow span {
    position: absolute;
    bottom: 3px;
    left: 0px;
    font-size: 18px;
    font-weight: bold;
}

div.slideshow a {
    color: #000;
}

```

Изготовить копию экрана, показывающую демонстрацию в действии очень сложно, но на рис. 9.6 можно как минимум посмотреть на ссылку запуска, добавленную на страницу.

Представленные ранее демонстрация и переходы по изображениям послужили реальным началом показа возможностей по созданию своих собственных динамических веб-приложений (к примеру, части презентационного программного обеспечения). Чтобы получить более четкое представление о том, как работает демонстрация изображений, я рекомендую вам установить кол, представленный в этой главе, и посмотреть на простые, но убедительные результаты его работы.

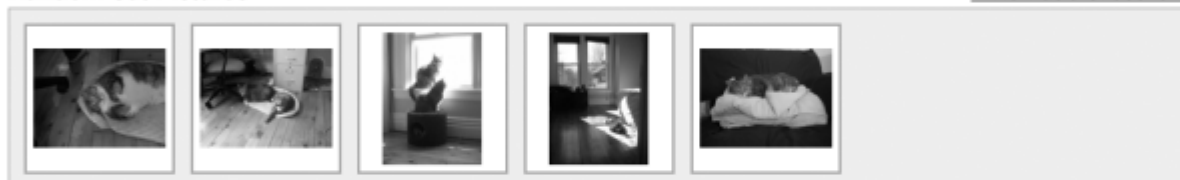
ructus, massa vitae placerat morcus, neque mauns iobortis enim, id ullamcorper dui nunc et enim. Integer faucibus morcus elit. Vestibulum ut ante. Morbi a sem. Vivamus accumsan.

Maecenas bibendum tellus at ante. Maecenas pharetra volutpat mauris. Vivamus vulputate. Pellentesque nec pede. Pellentesque aliquet, tellus nec placerat bibendum, lacus augue mattis tellus, id iaculis enim augue ac sapien. Maecenas commodo ante quis tellus. Phasellus tempor, massa non malesuada cursus, tortor justo mollis diam, eget sagittis dolor ipsum at velit. Curabitur at pede eu magna sollicitudin accumsan. Duis quis velit nec justo condimentum aliquet. Mauris dignissim mi. Phasellus non mauris. Aliquam sagittis blandit magna.

Etiam quam. Sed nisi. Maecenas viverra pellentesque ante. Fusce vulputate porta metus. Maecenas turpis urna, porta vitae, tempor vel, dapibus vestibulum, felis. Pellentesque sit amet nisl. Curabitur blandit. Sed in nisl et neque condimentum lacinia. Morbi arcu dui, dignissim in, consectetur quis, gravida ac, felis. Proin lacinia aliquet augue. Mauris nec odio. Vestibulum eu orci nec ligula consequat rhoncus. Phasellus nunc nunc, vulputate posuere, semper in, cursus nec, orci. Curabitur sem justo, ullamcorper vitae, adipiscing in, malesuada vel, elit. Maecenas eleifend, justo sed ornare consequat, sem ipsum pellentesque turpis, in rutrum orci neque sed massa. Vestibulum laoreet dui eget arcu. Quisque in lorem. Morbi quis lacus. Maecenas nonummy metus auctor nunc.

Random Cat Pictures

[» View as a Slideshow](#)



Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nullam at sem. Ut ultrices, dolor sit amet pharetra pretium, leo metus accumsan urna, eu ultrices purus augue at ligula. Nam gravida. Mauris neque lacus, vehicula et, hendrerit nec, semper quis, nibh. Nulla facilisi. Curabitur id ante. Proin felis ipsum, commodo non, hendrerit et, ultrices in, eros. Nulla tincidunt vulputate felis. Morbi convallis libero sed erat. Etiam lectus pede, dictum et, rutrum id, commodo nec, augue. Vestibulum massa. Fusce nec felis. Vivamus consectetur elementum elit.

Etiam mattis egestas felis. Proin sed magna at orci aliquet mattis. Vivamus ligula metus, interdum quis, viverra nec, vehicula vitae, mauris. Pellentesque venenatis pede vel augue. Aliquam fringilla nunc eget eros. Pellentesque eget ipsum. Morbi augue. Praesent sit amet ipsum. Ut vitae sem et diam mattis imperdiet. Phasellus a pede. Suspendisse potenti. Maecenas in sapien sit amet urna ultrices dignissim. Duis aliquet faucibus odio. Nunc a turpis vitae sem porta viverra. Vestibulum convallis dignissim nisl. Fusce enim. Morbi quam. Praesent tortor. Suspendisse eu dolor. Ut tempus tortor in dui.

Aenean justo. Integer egestas pharetra quam. Nulla faucibus, mi vel mollis tempus, tortor libero fringilla odio, ac iaculis eros quam eget lorem. Pellentesque nec enim. Cras porta. Duis ornare libero vel risus. In hac habitasse platea dictumst. Quisque luctus, massa vitae placerat rhoncus, neque mauris lobortis enim, id ullamcorper dui nunc et enim. Integer faucibus rhoncus elit. Vestibulum ut ante. Morbi a sem. Vivamus accumsan.

Maecenas bibendum tellus at ante. Maecenas pharetra volutpat mauris. Vivamus vulputate. Pellentesque nec pede. Pellentesque aliquet, tellus nec placerat bibendum, lacus augue mattis tellus, id iaculis enim augue ac sapien. Maecenas commodo ante quis tellus. Phasellus tempor, massa non malesuada cursus, tortor justo mollis diam, eget sagittis dolor ipsum at velit. Curabitur at pede eu magna sollicitudin accumsan. Duis quis velit nec justo condimentum aliquet. Mauris dignissim mi. Phasellus non mauris. Aliquam sagittis blandit magna.

Рис. 9.6. Дополнительная ссылка на запуск демонстрации, добавленная на страницу

Вывод

Галерея изображений, переходы по изображениям и их автоматическая демонстрация, представленные в этой главе, реально показывают пользу от применения DOM-сценариев для создания дополнительных функциональных возможностей на веб-странице, не создающих каких-либо существенных трудностей или неразберихи. Опираясь на все ранее изученное, можно прийти к очевидному выводу, что для динамического, ненавязчивого применения DOM-сценариев нет практически ничего невозможного.

В этой главе были рассмотрены две другие галереи изображений, вдохновившие нас на создание своего собственного экземпляра. Затем мы определили для галереи стандартный HTML-синтаксис и порядок ее отображения, а также набор ее основных комплектующих (включая затемнение, позиционируемый контейнер и переходы по изображениям). В качестве заключительного аккорда, мы добавили запускаемую пользователем автоматическую демонстрацию изображений. Таким образом без лишней суеты и с минимальным объемом кода мы создали мощный фрагмент динамического сценария, использующего объектную модель документа.

Глава 10 Введение в Ajax

Аjax — это термин, придуманный *Джесси Джеймсом Гарретом* (Jesse James Garrett) из компании Adaptive Path для объяснения асинхронной связи между клиентом и сервером, открывающей возможности использования объекта XMLHttpRequest, предоставляемого всеми современными браузерами. Ajax — это всего лишь термин, означающий Asynchronous JavaScript и XML, который используется для краткого изложения технологии, необходимой для создания динамического веб-приложения. Кроме того, отдельные компоненты технологии Ajax являются полностью взаимозаменяемыми — вполне приемлемо, к примеру, вместо XML использовать HTML.

В этой главе мы собираемся рассмотреть детали формирования полноценного Ajax-процесса (который концентрируется на осуществлении запроса от браузера к серверу). Будет рассмотрено все, от физического запроса как такового, до JavaScript-взаимодействия и работы с данными, необходимой для выполнения всей работы. Сюда включаются:

- Исследование различных типов HTTP-запросов и определение, как наилучшим образом отправить объекты данных на сервер.
- Рассмотрение всего http-ответа и попытка обработки всех ошибок, которые могут с ним произойти, включая истечение времени отклика сервера.
- Чтение данных, присланных сервером в ответ, перемещение по ним и их обработка.

Полностью разобравшись с тем, как осуществляется Ajax-процесс и как он может быть реализован, вы поймете, как он может быть использован во всем, от типичных ситуаций, до полноценных приложений. В главах 11, 12 и 13, мы также будем рассматривать ряд случаев использования Ajax-технологий.

Использование Ajax

Для создания простой Ajax-реализации не требуется большого объема кода, но ее предоставляемые ею возможности впечатляют. К примеру, вместо того, чтобы заставлять пользователя после отправки формы целиком запрашивать всю страницу, процесс отправки может быть обработан асинхронно, и после его завершения будет отображена только небольшая часть страницы с желаемыми результатами. Например, процесс поиска доступных доменных имен (с целью их приобретения) может быть медленным и трудоемким. Как только вам захочется найти новое имя, нужно набирать запрос в форме, отправлять его, и наблюдать за перезагрузкой страницы. С использованием Ajax можно получить мгновенный результат, такой, как, к примеру, предоставляет веб-приложение Instant Domain Search ([http:// instantdomainsearch.com/](http://instantdomainsearch.com/)), показанное на рис. 10.1.

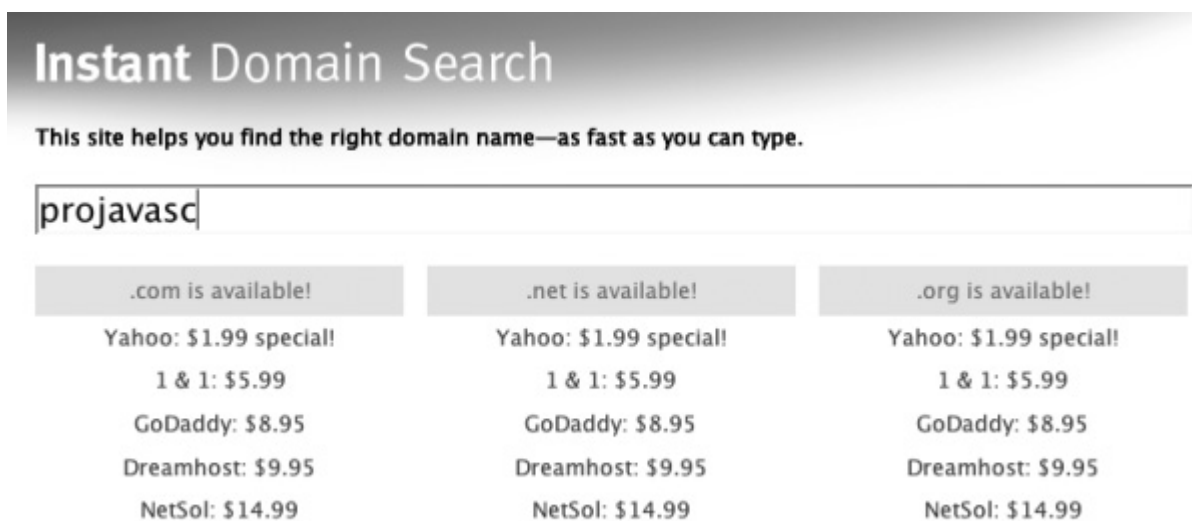


Рис. 10.1. Пример использования Instant Domain Search для поиска доменных имен по мере набора текста

HTTP-запросы

Пожалуй наиболее важным и, наверное, самым совместимым аспектом Ajax является та часть процесса, которая относится к HTTP-запросу. Hypertext Transfer Protocol (HTTP) был разработан для простой передачи HTML-документов и однородных файлов. К счастью все современные браузеры поддерживают средства для динамической установки HTTP-соединения с использованием JavaScript. Как показала практика, эти средства очень полезны для разработки более «отзывчивых» веб-приложений.

Асинхронная отправка данных на сервер и получение в ответ дополнительных данных является основным назначением Ajax. Как именно отформатированы данные в конечном счете зависит от определенных вами требований, которые будут рассмотрены разделе «Обработка данных ответа».

В следующих разделах мы собираемся посмотреть, как форматировать данные, передаваемые на сервер с использованием HTTP-запросов. Затем мы собираемся рассмотреть, как устанавливаются основные соединения с сервером, и посмотреть на подробности их реализации в кроссбраузерной среде.

Установка соединения

Первоначальным аспектом Ajax-процесса является открытие соединения с сервером. Чтобы этого добиться существует несколько различных способов, но мы рассмотрим специфические средства, с помощью которых можно легко не только послать, но и получить данные. Эта технология обычно называется «использование объекта XMLHttpRequest».

В зависимости от пользовательского браузера передача данных производится двумя различными способами применения объекта XMLHttpRequest:

- Internet Explorer, которые впервые проложил путь этому средству браузерной передачи данных, устанавливает все свои соединения с помощью ActiveXObject (конкретная версия которого изменяется в зависимости от версии Internet Explorer). К счастью, Internet Explorer 7 обладает собственной поддержкой объекта XMLHttpRequest.
- Все остальные современные браузеры локализовали все возможности XMLHttpRequest в объекте под тем же названием. К их числу относятся Firefox, Opera и Safari.

Хорошо, что несмотря на отличия существующего в Internet Explorer метода создания объекта XMLHttpRequest от метода, используемого во всех остальных современных браузерах, у этого объекта такой же набор полезных функциональных свойств. Объект XMLHttpRequest обладает рядом методов, используемых для установки соединения и чтения данных с сервера. В листинге 10.1 показано, как послать на сервер основной GET-запрос.

Листинг 10.1. Кроссбраузерное средство создания HTTP GET-запроса на сервер

```
// Если используется IE, создание для объекта XMLHttpRequest программного
// обрамления
if ( typeof XMLHttpRequest == "undefined" )
    XMLHttpRequest = function() {
        // В Internet Explorer для создания нового объекта XMLHttpRequest
        // используется ActiveXObject
        return new ActiveXObject(
            // IE 5 использует XMLHTTP объект, отличающийся от объекта,
            // используемого в IE 6
            navigator.userAgent.indexOf("MSIE 5") >= 0 ?
            "Microsoft.XMLHTTP" : "Msxml2.XMLHTTP"
```

```

    );
};
// Создание объекта запроса
var xml = new XMLHttpRequest();

// Открытие сокета
xml.open("GET", "/some/url.cgi", true);

// Установка соединения с сервером и отправка любых дополнительных
// данных
xml.send();

```

Как видите, код, необходимый для установки соединения с сервером, совсем простой, и его создание не требует практически никаких усилий. Сложности возникают, когда дело доходит до дополнительных возможностей (например, до проверки истечения срока запроса или измененных данных); но эти детали мы рассмотрим в разделе «HTTP ответ».

Самая важная особенность всей методологии Ajax состоит в передаче данных от клиента (например, веб-браузера) к серверу. С учетом этого давайте рассмотрим детали, необходимые для упаковки данных и отправки их на сервер.

Преобразование данных в последовательную форму

Первой стадией отправки данных на сервер является приведение их к такому формату, который может быть легко прочтен сервером; этот процесс называется преобразованием в последовательную форму (сериализацией). Существует два варианта сериализации, которые могут предоставить вам самый широкий диапазон возможностей передачи:

- Передача обычного объекта JavaScript, который может быть использован для хранения пар ключ-значение (где значения представлены либо числами, либо строками).
- Передача значений из нескольких элементов ввода формы (этот вариант отличается от первого тем, что в нем имеет значение порядок передаваемых элементов, тогда как порядок значений, передаваемых в первом случае, может быть абсолютно произвольным).

Давайте посмотрим в листинге 10.2 несколько примеров типов данных, которые можно передать на сервер, а также на получившееся из них на выходе последовательное представление, воспринимаемое сервером.

Листинг 10.2. Примеры простых JavaScript-объектов, преобразованных в последовательную форму

```

// Простой объект, хранящий пары ключ-значение
{
    name: "John",
    last: "Resig",
    city: "Cambridge",
    zip: 02140
}

// Последовательная форма
name=John&last=Resig&city=Cambridge&zip=02140

```

```
// Другой набор данных, имеющий множество значений
[
  { name: "name", value: "John" },
  { name: "last", value: "Resig" },
  { name: "lang", value: "JavaScript" },
  { name: "lang", value: "Perl" },
  { name: "lang", value: "Java" }
]

// И последовательная форма этих данных
name=John&last=Resig&lang=JavaScript&lang=Perl&lang=Java

// В завершение найдем несколько элементов ввода (используя метод
// id(), который мы создали в главе, посвященной DOM)
[
  id( "name" ),
  id( "last" ),
  id( "username" ),
  id( "password" )
]
// И превратим их в строку данных
name=John&last=Resig&username=jeresig&password=test
```

Формат, использованный для сериализации данных является стандартным форматом для передачи в HTTP-запросе. Вы, наверное, их уже видели в стандартном HTTP GET-запросе, который выглядит следующим образом:

```
http://someurl.com/?name=John&last=Resig
```

Эти данные также могут быть переданы в POST-запрос (и в намного большем количестве, чем в простое отправление). Мы рассмотрим эти отличия в будущем разделе.

На данный момент давайте создадим стандартные средства сериализации структур данных, представленных в листинге 10.2. Специальная функция, выполняющая эту задачу, показана в листинге 10.3. Она способна преобразовать в последовательную форму большинство элементов ввода формы, за исключением элементов, имеющих множественный выбор.

Листинг 10.3. Стандартная функция для сериализации структур данных в совместимую с HTTP схему параметров

```
// Сериализация набора данных. Может воспринимать два различных типа
// объектов:
// - массив элементов ввода
// - хэш, составленный из пар ключ-значение
// Функция возвращает последовательную строку данных
function serialize(a) {
  // Набор результатов сериализации
  var s = [];

  // Если передан массив, предположение, что он является массивом
  // элементов формы
```

```

if ( a.constructor == Array ) {

    // Сериализация элементов формы
    for ( var i = 0; i < a.length; i++ )
        s.push( a[i].name + "=" + encodeURIComponent( a[i].value ) );
    // Если нет, предположение, что это объект, состоящий
    // из пар ключ-значение
} else {

    // Сериализация пар ключ-значение
    for ( var j in a )
        s.push( j + "=" + encodeURIComponent( a[j] ) );
}

// возврат результатов сериализации
return s.join("&");
}

```

Получив последовательную форму наших данных (в виде простой строки), мы может посмотреть, как отправить эти данные на сервер, используя GET- или POST-запрос.

Создание GET-запроса

Вернемся к созданию HTTP GET-запроса на сервер с использованием XMLHttpRequest, но на этот раз уже с отправкой дополнительных данных, преобразованных в последовательную форму. Простой пример такого запроса показан в листинге 10.4.

Листинг 10.4. Кроссбраузерное средство создания HTTP GET-запроса к серверу (не приспособленное для чтения ответных данных)

```

// Создание объекта запроса
var xml = new XMLHttpRequest();
// Открытие асинхронного GET-запроса
xml.open("GET", "/some/url.cgi?" + serialize( data ), true);

// Установка соединения с сервером
xml.send();

```

Важно отметить, что сериализованные данные добавляются к URL сервера (с использованием в качестве разделителя знака вопроса — ?). Все веб-серверы и находящиеся на них веб-приложения знают, что данные, включенные после знака вопроса являются последовательным набором пар ключ-значение. Вопрос обработки возвращаемого сервером ответа (основанного на переданных ему данных) будет рассмотрен в разделе «Обработка ответных данных».

Создание POST-запроса

Другая форма создания HTTP-запроса к серверу, с использованием XMLHttpRequest, относится к POST, и в ней задействуются совершенно другие способы отправки данных на сервер. Прежде всего POST-запрос способен отправлять данные любого формата и любой длины (не ограничиваясь лишь сериализованной строкой данных).

При передаче на сервер сериализованный формат, используемый для данных обычно снабжается типом содержимого `application/x-www-form-urlencoded`. Это означает, что вы также можете посылать на сервер чистый XML (с типом контента `text/xml` или `application/xml`) или даже объект JavaScript (используя тип контекста `application/json`).

В листинге 10.5 показан простой пример создания запроса и отправки дополнительных сериализованных данных.

Листинг 10.5. Кроссбраузерное средство создания HTTP POST-запроса к серверу (не приспособленное для чтения ответных данных)

```
// Создание объекта запроса
var xml = new XMLHttpRequest();

// Открытие асинхронного POST-запроса
xml.open("POST", "/some/url.cgi", true);

// Отправка заготовка типа контекста (content-type),
// позволяющего серверу узнать, как интерпретировать посланные данные
xml.setRequestHeader(
    "Content-Type", "application/x-www-form-urlencoded");

// Обеспечение отправки правильной длины сериализованных данных —
// браузеры, основанные на движке Mozilla иногда испытывают с этим
// проблемы
if ( xml.overrideMimeType )
    xml.setRequestHeader("Connection", "close");

// Установка соединения с сервером и отправка сериализованных
//данных
xml.send( serialize( data ) );
```

Чтобы более подробно рассмотреть предыдущее утверждение, обратимся к случаю отправки на сервер данных, не приведенных к сериализованному формату. Соответствующий пример показан в листинге 10.6.

Листинг 10.6. Пример отправки на сервер POST-запроса, содержащего данные в формате XML

```
// Создание объекта запроса
var xml = new XMLHttpRequest();

// Открытие асинхронного POST-запроса
xml.open("POST", "/some/url.cgi", true);

// Установка заголовка типа контекста, чтобы сервер знал,
// как интерпретировать отправляемые XML-данные
xml.setRequestHeader( "Content-Type", "text/xml");

// Обеспечение отправки правильной длины данных — браузеры,
// основанные на движке Mozilla иногда испытывают с этим проблемы
if ( xml.overrideMimeType )
```

```
xml.setRequestHeader("Connection", "close");
```

```
// Установка соединения с сервером и отправка данных
```

```
xml.send( "<items><item id='one'/><item id='two'/></items>" );
```

Весьма существенной является возможность посылать большое количество данных (в отличие от GET-запроса, предел для которого, в зависимости от браузера, составляет всего пару килобайт, ограничений на количество передаваемых данных нет). Это позволяет создавать реализации различных протоколов обмена данными, например, XML-RPC или SOAP.

Но, чтобы не усложнять материал, мы ограничимся наиболее распространенными и полезными форматами данных, которые делают доступным HTTP ответ.

HTTP ответ

Аспектом создания и использования XMLHttpRequest, который ставит его выше всех остальных упрощенных форм односторонней связи, является возможность чтения различных текстовых форматов данных, посылаемых сервером. Сюда включается и один из краеугольных камней Ajax: XML (хотя этим еще не утверждается, что при создании Ajax-приложений может быть использован только XML. Кстати, в разделе «Обработка ответных данных» показан ряд других альтернативных форматов).

Для начала взглянем на листинг 10.7 с очень простым примером обработки данных ответа, полученного от сервера.

Листинг 10.7. Установка соединения с сервером и чтение результирующих данных

```
// создание объекта запроса
```

```
var xml = new XMLHttpRequest();
```

```
// Открытие асинхронного POST-запроса
```

```
xml.open("GET", "/some/url.cgi", true);
```

```
// Отслеживание момента обновления статуса документа
```

```
xml.onreadystatechange = function(){
```

```
    // Ожидание, пока не завершится загрузка данных
```

```
    if ( xml.readyState == 4 ) {
```

```
        // xml.responseXML содержит XML-документ (если таковой был
        // возвращен)
```

```
        // xml.responseText содержит текст ответа
```

```
        // (если не был предоставлен XML-документ)
```

```
        // Подчистка для экономии пространства памяти
```

```
        xml = null;
```

```
    }
```

```
};
```

```
// Установка соединения с сервером
```

```
xml.send();
```


В этом примере можно увидеть, как получать доступ к различным блокам данных, полученных в ответе HTTP. Каждое из двух свойств, `responseXML` и `responseText`, будет содержать данные, отформатированные соответствующим образом. Например, если сервер возвращает XML-документ, то DOM-документ будет находиться в `responseXML`; любой другой ответ и его результаты будут находиться в `responseText`.

Перед тем, как заняться обработкой, перемещением и управлением полученными данными, давайте поработаем над более сложной версией функции `onreadystatechange` (из листинга 10.7), способной обрабатывать ошибки сервера и истечение срока соединения.

Обработка ошибок

К сожалению, объект `XMLHttpRequest` не имеет какого-нибудь встроенного механизма обработки ошибок сервера, наличие которого сэкономило бы массу времени. Но вы можете без особого труда создать свой собственный механизм. При отправке запроса бывают случаи, требующие отслеживания ситуации для определения, столкнулся ли сервер с проблемами при его обработке:

- *Код ответа успешно обработанного запроса:* Предполагаемый способ проверки на наличие ошибок заключается в отслеживании кода состояния ответа HTTP, который включен в HTTP-спецификацию как средство оповещения клиента о том, что делает сервер. Успешным считается такой запрос, чей код состояния находится в диапазоне 200.
- *Ответ, не подвергшийся изменениям:* Возвращенный сервером документ может иметь пометку «Not Modified» (код состояния 304). Это означает, что данные, полученные с сервера, не подверглись изменениям, и были загружены не с него, а из персональной кэш-памяти браузера. Поскольку данные по-прежнему могут быть считаны клиентом, важно не считать этот ответ за ошибку.
- *Локально размещенные файлы:* Если Ajax-приложение запущено на вашем локальном компьютере (без участия веб-сервера), код состояния возвращаться не будет, даже если запрос будет успешным. Значит, ситуацию, при которой запрос не имеет кода состояния и осуществляется просмотр локального файла нужно рассматривать как успешный ответ.
- *Ответ, не подвергшийся изменениям и Safari:* Если документ не подвергся изменениям со времени последнего запроса (и если вы не отправляли явным образом серверу заголовок `IF-MODIFIED-SINCE`). Этот весьма странный случай может позже помешать процессу отладки.

Помня об этом, рассмотрим листинг 10.8, в котором представлена реализация проверки ответа, которую я ранее уже в общих чертах обрисовал.

Листинг 10.8. Функция, которая может быть использована для проверки состояния успешности полученного с сервера ответа HTTP

```
// Проверка, имеет ли объект XMLHttpRequest состояние успешности.
// Функция принимает один аргумент — объект XMLHttpRequest
function httpSuccess(r) {
    try {
        // Если состояние сервера предоставлено не было, и мы фактически
        // сделали запрос к локальному файлу, значит, он прошел успешно
        return !r.status && location.protocol == "file:" ||

            // Нас устраивает любой код состояния в диапазоне 200
            ( r.status >= 200 && r.status < 300 ) ||

            // Запрос прошел успешно, если документ не подвергся изменениям
            r.status == 304 ||
    }
}
```

```

        // Если файл не подвергался изменениям, Safari возвращает пустое
        // состояние
        navigator.userAgent.indexOf("Safari") >= 0 &&
            typeof r.status == "undefined";
    } catch(e){}

// Если проверка состояния не удалась, следует предположить,
// что запрос тоже закончился неудачей
    return false;
}

```

Проверка состояния ответа HTTP — шаг очень важный; если его не сделать, можно получить какие-нибудь неприятные и весьма непредсказуемые результаты (например, вместо XML-документа будет возвращена страница ошибки HTML).

В разделе «Полноценный Ajax-пакет» мы встроим функцию в законченное Ajax-решение.

Проверка истечения времени запроса

Еще одной полезной технологией, не включенной в исходную реализацию XMLHttpRequest, является определение момента когда время запроса к серверу истекло, и он уже потерял всякий смысл.

Реализация этого свойства Implementing this feature isn't as cut-and-dry, but determining the success state of the request (as you did in the previous section) is possible with a little bit of work.

Listing 10-9 shows how you would go about checking for a request time-out in an application of your own.

Листинг 10.9. Пример проверки истечения времени запроса

```

// Создание объекта запроса
var xml = new XMLHttpRequest();

// Открытие асинхронного POST-запроса
xml.open("GET", "/some/url.cgi", true);

// Мы собираемся дать на ожидание ответа 5 секунд
var timeoutLength = 5000;

// Отслеживание успешного выполнения запроса
var requestDone = false;

// Инициализация функции обратного вызова, которая будет запущена через
// 5 секунд, отменяя запрос (если он не будет к тому времени выполнен)
setTimeout(function(){
    requestDone = true;
}, timeoutLength);

// Отслеживание обновления состояния документа
xml.onreadystatechange = function(){
    // Ожидание, полной загрузки данных,

```

```

// и проверка, не истекло ли время запроса
if ( xml.readyState == 4 && !requestDone ) {

    // xml.responseXML содержит XML-документ (если он был возвращен)
    // xml.responseText содержит текст
    // (если XML-документ не был предоставлен)

    // Подчистка для экономии пространства памяти
    xml = null;
}
};

// Установка соединения с сервером
xml.send();

```

Когда учтены все детали обмена данными с сервером, включая все множество возможных ошибок, наступает время обратиться к подробностям обработки ответных данных, полученных с сервера.

Обработка ответных данных

Во всех ранее приводимых примерах место для ответных данных, полученных с сервера пустовало по той простой причине, что существует поистине бесчисленное множество всевозможных форматов данных, которые могут быть возвращены сервером. Но на самом деле XMLHttpRequest работает только с форматами данных, имеющими текстовую основу. Но даже при этом с некоторыми из них он работает лучше (с XML), чем с другими (с JSON). В этой главе мы собираемся рассмотреть три различных формата данных, которые могут быть возвращены сервером, а затем считаны и обработаны клиентом:

- *XML*: Хорошо, что все современные браузеры изначально обеспечивают обработку XML-документов, автоматически превращая их в полезные DOM-документы.
- *HTML*: Этот формат отличается от XML-документа тем, что обычно представляет собой простую текстовую строку, содержащую фрагмент HTML-кода.
- *JavaScript/JSON*: Он охватывает два формата данных — простой, исполняемый код JavaScript, и представление объекта JavaScript — JSON (JavaScript Object Notation).

У каждого из этих форматов данных есть различные случаи применения, в которых они могут быть особенно полезны. К примеру, существует масса примеров, когда имеет больше смысла вместо XML-документа возвращать фрагменты кода HTML. Важный аспект извлечения данных из ответа HTTP заключается в двух свойствах объекта XMLHttpRequest:

- *responseXML*: Это свойство будет содержать ссылку на заранее сгенерированный DOM-документ (представляющий документ XML) если с сервера был возвращен XML-документ. Это случается только если сервер явным образом указал в заголовке контента «Content-type: text/xml», или подобный этому тип данных XML.
- *responseText*: Это свойство содержит ссылку на простую текстовую строку возвращенных сервером данных. На этот метод для доступа к своим данным полагаются два типа данных: HTML и JavaScript.

Имея в распоряжении эти два свойства, можно без особого труда создать универсальную функцию для детерминированного извлечения данных из ответа HTTP (и даже определить с чем вы имеете дело, с XML-ответом или с обычным текстом). В листинге 10.10 показана функция, которую можно использовать именно в этом качестве.

Листинг 10.10. Функция, предназначенная для извлечения правильных данных из ответа HTTP-сервера

```

// Функция для извлечения данных из ответа HTTP
// Она принимает два аргумента, объект XMLHttpRequest и
// необязательный аргумент - тип данных, ожидаемых с сервера
// Приемлемы следующие значения: xml, script, text или html - по
// умолчанию - "", что устанавливает тип данных на основе заголовка
// content-type
function httpData(r, type) {
    // Получение заголовка content-type
    var ct = r.getResponseHeader("content-type");

    // Если не предоставлен тип по умолчанию, определение
    // не возвращена ли с сервера какая-либо форма XML
    var data = !type && ct && ct.indexOf("xml") >= 0;

    // Получение объекта XML-документа, если сервер вернул XML,
    // если нет - возвращение полученного с сервера текстового содержимого
    data = type == "xml" || data ? r.responseXML : r.responseText;

    // Если указан тип "script", выполнение возвращенного текста,
    // реагируя на него, как на JavaScript
    if ( type == "script" )
        eval.call( window, data );

    // Возвращение данных, полученных в ответе (или XML-документа, или
    // текстовой строки)
    return data;
}

```

Теперь, располагая этой функцией извлечения данных, у нас имеются все компоненты, необходимые для построения законченной функции, создающей обычный Ajax-вызов данных с сервера. Полная реализация функции показана в следующем разделе.

Полноценный Ajax-пакет

Используя все изученные до сих пор понятия, можно создать универсальную функцию для обработки всех Ajax-запросов и связанных с ними ответов. В основном эта функция станет в будущих главах фундаментом для нашей Ajax-разработки, позволяющей осуществлять быстрые запросы к серверу для получения дополнительной информации.

Законченная Ajax-функция показана в листинге 10.11.

Листинг 10.11. Законченная функция, способная осуществлять необходимые задачи, связанные с использованием Ajax

```

// Универсальная функция, предназначенная для осуществления Ajax-запросов
// Она принимает один аргумент, представляющий собой объект, содержащий
// набор параметров, каждый из которых имеет краткое описание в последующих
// комментариях
function ajax( options ) {

```

```

// Загрузка объекта параметров по умолчанию, если пользователем не
// представлено никаких значений
options = {
  // Тип http-запроса
  type: options.type || "POST",

  // URL на который должен быть послан запрос
  url: options.url || "",

  // Время ожидания ответа на запрос
  timeout: options.timeout || 5000,

  // Функция, вызываемая, когда запрос неудачен, успешен
  // или завершен (успешно или нет)
  onComplete: options.onComplete || function(){},
  onError: options.onError || function(){},
  onSuccess: options.onSuccess || function(){},

  // Тип данных которые будут возвращены с сервера
  // по умолчанию просто определить, какие данные были
  // возвращены, и действовать соответственно.
  data: options.data || ""
};

// Создание объекта запроса
var xml = new XMLHttpRequest();

// Открытие асинхронного запроса
xml.open(options.type, options.url, true);

// Ожидание отклика на запрос в течение 5 секунд
// перед тем, как от него отказаться
var timeoutLength = options.timeout;

// Отслеживание факта успешного завершения запроса
var requestDone = false;

// Инициализация функции обратного вызова, которая будет запущена через
// 5 секунд, отменяя запрос (если он не будет к тому времени выполнен)
setTimeout(function(){
  requestDone = true;
}, timeoutLength);

// Отслеживание обновления состояния документа
xml.onreadystatechange = function(){
  // Ожидание, полной загрузки данных,
  // и проверка, не истекло ли время запроса

```

```

if ( xml.readyState == 4 && !requestDone ) {

    // Проверка успешности запроса
    if ( httpSuccess( xml ) ) {

        // Выполнение в случае успеха функции обратного вызова
        // с данными, возвращенными с сервера
        options.onSuccess( httpData( xml, options.type ) );

        // В противном случае произошла ошибка, поэтому нужно
        // выполнить функцию обратного вызова для обработки ошибки
    } else {
        options.onError();
    }

    // Выполнение функции обратного вызова, связанной с завершением
    // запроса
    options.onComplete();

    // Подчистка для экономии пространства памяти
    xml = null;
}
};

// Установка соединения с сервером
xml.send();

// Определение успешности получения ответа HTTP
function httpSuccess(r) {
    try {
        // Если состояние сервера предоставлено не было, и мы
        // фактически сделали запрос к локальному файлу,
        // значит, он прошел успешно
        return !r.status && location.protocol == "file:" ||

            // Нас устраивает любой код состояния в диапазоне 200
            ( r.status >= 200 && r.status < 300 ) ||

            // Запрос прошел успешно, если документ не подвергся
            // изменениям
            r.status == 304 ||

            // Если файл не подвергался изменениям, Safari возвращает
            // пустое состояние
            navigator.userAgent.indexOf("Safari") >= 0
            && typeof r.status == "undefined";
    } catch(e) {}
}

```

```

    // Если проверка состояния не удалась, следует предположить,
    // что запрос тоже закончился неудачей
    return false;
}

// Извлечение правильных данных из ответа HTTP
function httpData(r,type) {
    // Получение заголовка content-type
    var ct = r.getResponseHeader("content-type");

    // Если не предоставлен тип по умолчанию, определение
    // не возвращена ли с сервера какая-либо форма XML
    var data = !type && ct && ct.indexOf("xml") >= 0;

    // Получение объекта XML-документа, если сервер вернул XML,
    // если нет – возвращение полученного с сервера текстового
    // содержимого
    data = type == "xml" || data ? r.responseXML : r.responseText;

    // Если указан тип "script", выполнение возвращенного текста,
    // реагируя на него, как на JavaScript
    if ( type == "script" )
        eval.call( window, data );

    // Возвращение данных, полученных в ответе (или XML-документа, или
    // текстовой строки)
    return data;
}
}

```

Важно отметить, что запрос страниц, которые находятся не в том же домене, что и страница, с которой производится запрос, невозможно. Это обусловлено ограничениями, налагаемыми из соображений безопасности всеми современными браузерами (чтобы пресечь попытки кражи вашей персональной информации). Теперь, когда мы располагаем столь мощной функцией, настало время проработать несколько примеров, демонстрирующих новоприобретенную силу Ajax.

Примеры различного использования данных

По сути ситуации создания простых Ajax-запросов мало отличаются друг от друга, но, что действительно изменяется, так это данные, которые сервер посылает в ответ. В зависимости от цели, которую вы пытаетесь достичь, наличие различных форматов данных может стать очень полезным обстоятельством. Именно по этому я и собираюсь вам показать, как выполняются некоторые стандартные задачи с применением ряда различных форматов.

RSS-поток, основанный на формате XML

Несомненно наиболее популярным форматом для возвращаемых сервером данных является XML, и для этой популярности есть весьма серьезные основания. Все современные браузеры обладают своей собственной поддержкой XML-документов, конвертируя их на лету в DOM-представление. Поскольку всю тяжелую работу по синтаксическому разбору берет на себя браузер, все, что остается сделать — пройтись по нему, как по любому

другому DOM-документу. При этом важно отметить, что перемещаться по восстановленному из удаленного источника XML-документу с помощью функции `getElementById` в принципе невозможно. Просто потому, что обычные не-HTML XML-документы не имеют у себя предварительно запрограммированного уникального атрибута ID. Но не смотря на сказанное, все же способ эффективного перемещения по XML-документам есть.

В листинге показан простой пример использования возвращенного XML для создания на веб-сайте элемента отображения RSS-потока.

Листинг 10.12. Загрузка заголовков новостей, содержащихся в удаленном RSS-потоке, основанном на формате XML

```
<html>
<head>
  <title>Динамический элемент отображения RSS-потока</title>
  <!--загрузка нашей универсальной Ajax-функции -->
  <script src="ajax.js"></script>
  <script>
    // Ожидание полной загрузки документа
    window.onload = function(){
      // И загрузка RSS-потока с использованием Ajax
      ajax({
        // URL RSS-потока
        url: "rss.xml",

        // Это XML-документ
        type: "xml",

        // Эта функция будет выполнена, когда запрос будет завершен
        onSuccess: function( rss ) {
          // Все заголовки у которых id равны "feed",
          // мы собираемся поместить в теги <ol>
          var feed = document.getElementById("feed");

          // Использование всех заголовков RSS XML-документа
          var titles = rss.getElementsByTagName("title");

          // последовательный перебор всех соответствующих
          // заголовков новостей
          for ( var i = 0; i < titles.length; i++ ) {
            // Создание <li>-элемента для размещения заголовка
            // новости
            var li = document.createElement("li");

            // Установка содержимого заголовка в элемент
            li.innerHTML = titles[i].firstChild.nodeValue;

            // и добавление его в DOM, в <ol>-элемент
            feed.appendChild( li );
          }
        }
      });
    }
  </script>
</head>
</html>
```



```

        }
    }
    });
};
</script>
</head>
<body>
    <h1> Динамический элемент отображения RSS-потока</h1>
    <p>Ознакомление с RSS-потоком:</p>
    <!-- Именно сюда мы собираемся вставить RSS-поток -->
    <ol id="feed"></ol>
</body>
</html>

```

Как видите, после того, как все сложности, связанные с Ajax-процессом запроса-ответа перемещены в другое место, для решения этой задачи особых усилий прикладывать не пришлось. Кроме того, поскольку браузеры существенно упростили проход по XML-документу, этот способ действительно стал великолепным средством быстрой передачи данных от сервера к клиенту.

Вставка HTML

Еще одной технологией, которая может быть выполнена с использованием Ajax является загрузка в документ HTML-фрагментов. Эта технология отличается от только что рассмотренной методики работы с XML-документом тем, что для нее не требуется проводить синтаксический разбор или осуществлять проход по данным, получаемым с сервера; она используется только для вставки этих данных в документ. Использование этого метода — действительно быстрый и впечатляющий способ простого и немедленного получения обновления вашей веб-страницы. Пример использования этого метода показан в листинге 10.13.

Листинг 10.13. Загрузка HTML-фрагмента из удаленного файла и вставка его в текущую веб-страницу

```

<html>
<head>
    <title> Спортивный счет в формате HTML, загруженный Ajax </title>
    <!-- Загрузка нашей универсальной Ajax-функции -->
    <script src="ajax.js"></script>
    <script>
        // Ожидание полной загрузки документа
        window.onload = function(){
            // Затем загрузка спортивного счета с использованием Ajax
            ajax({
                // URL, где находятся спортивный счет в формате HTML
                url: "scores.html",

                // Это HTML-документ
                type: "html",

                // Эта функция будет выполнена, когда запрос будет завершен
                onSuccess: function( html ) {
                    // Мы собираемся вставить HTML в div-элемент,
                    // значение id которого равно 'scores'

```

```

        var scores = document.getElementById("scores");

        // Вставка нового HTML в документ
        scores.innerHTML = html;
    }
    });
};
</script>
</head>
<body>
    <h1> Спортивный счет в формате HTML, загруженный с помощью Ajax </h1>
    <!-- Сюда будет вставлен спортивный счет -->
    <div id="scores"></div>
</body>
</html>

```

Наиболее важный аспект динамической HTML-технологии заключается в том, что вы по-прежнему можете хранить все шаблоны уровня приложения в коде на стороне сервера, что позволяет централизованно содержать и без проблем обслуживать весь базовый код шаблонов.

Простоту загрузки обычных HTML-файлов трудно переоценить, поскольку она предоставляет самый легкий способ придать веб-приложению лучшую отзывчивость на действия пользователя.

JSON и JavaScript: Удаленное выполнение

В заключение я собираюсь рассмотреть формат данных (один из тех, к которому мы еще вернемся в главе 13, при изучении wiki), который относится к передаче строк данных JSON и простого кода JavaScript. Передача сериализованных JSON-данных может служить облегченной альтернативой передаче XML-документов от сервера к клиенту. Кроме того, предоставление сервером простого кода JavaScript служит превосходным способом построения динамических многопользовательских приложений. Чтобы ничего не усложнять, давайте рассмотрим удаленную загрузку в приложение файла JavaScript, показанную в листинге 10.14.

Листинг 10.14. Динамическая загрузка и исполнение удаленного JavaScript-файла

```

<html>
<head>
    <!-- Загрузка нашей универсальной Ajax-функции -->
    <script src="ajax.js"></script>
    <script>
        // Загрузка удаленного JavaScript-файла
        ajax({
            // URL JavaScript-файла
            url: "myscript.js",

            // Принуждение его к выполнению в качестве кода JavaScript
            type: "script"
        });
    </script>
</head>
<body></body>

```

</html>

Вывод

При всей своей обманчивой простоте, концепция веб-приложений Ajax обладает большими возможностями. Динамическая загрузка дополнительных информационных частей в уже запущенное приложение, использующее JavaScript, позволяет создавать более отзывчивый интерфейс, способный порадовать пользователей.

В этой главе мы на изучили основных концепций работы Ajax, включая особенности http-запроса и ответа, обработку ошибок, форматирование данных и их синтаксический разбор. В результате этого была разработана универсальная функция, которую можно многократно использовать для упрощения работы по приданию большей динамичности любому веб-приложению. Эта функция будет использована в следующих трех главах для создания динамических элементов взаимодействия, основанных на технологии Ajax.

Глава 11 Усовершенствование блогов с помощью Ajax

Одной из возможностей, предоставляемой технологией Ajax, является предоставление дополнительных уровней взаимодействия с пользователями при работе со статическими веб-страницами. Это означает, что вы можете приступать к изменениям характера работы статической веб-страницы, не нарушая целостности пользовательского представления.

Одна из областей, которую можно усовершенствовать — это *сетевые журналы*, или блоги. Если взглянуть на него с точки зрения данных, то блог — это ничто иное, как перечень статей, в котором есть текстовая статья, заголовок и ссылка на полную версию статьи. Но средства просмотра старых статей и проверки наличия новых оставляют желать лучшего.

В этой главе мы рассмотрим два разных способа улучшения типичного блога за счет применения кода JavaScript, в котором используется технология Ajax. Один из них представляет собой средство простой прокрутки большого перечня статей без оставления текущей страницы, а другой — способ отслеживания новых статей блога без непрерывной перезагрузки веб-страницы.

Бесконечный блог

Первым усовершенствованием, которое мы собираемся применить к блогу — возможность осуществления обратной прокрутки архивов без каких-либо щелчков на ссылках переходов. Общей особенностью блогов, или иных веб-сайтов, содержимое которых имеет хронологический характер, является возможность перехода к более ранним статьям. Зачастую она предоставляется за счет ссылок Следующая и (или) Предыдущая в нижней части страницы, которые позволяют пользователю перемещаться по архивным статьям.

Мы будем исследовать способ, при котором весь этот процесс будет проведен с помощью Ajax. Чтобы создать утилиту, способную выполнить эту задачу, нужно сделать несколько допущений:

- У нас есть веб-страница с серией публикаций, изложенных в хронологическом порядке.
- Когда пользователь приближается к концу страницы, у него возникает желание прочитать предыдущие публикации.
- У нас есть источник данных, из которого можно брать публикации. В данном случае мы собираемся воспользоваться WordPress, программным обеспечением для создания блогов (<http://wordpress.org/>), которое хорошо справляется с этой задачей.

Тогда замыслом нашего сценария будет автоматическая подгрузка дополнительных публикаций как только пользователь прокрутит изображение ближе к нижней части страницы, что позволит ему продолжить прокрутку и перемещаться по архиву, при этом возникнет иллюзия бесконечной веб-страницы. Этот сценарий будет создан с использованием функциональных возможностей программного обеспечения WordPress, предназначенного для создания блогов. У вас появится возможность включить этот сценарий и в свой собственный блог, построенный на основе WordPress, придав ему дополнительные функциональные возможности.

Шаблон блога

Сначала мы собираемся воспользоваться основным шаблоном, предоставляемым по умолчанию при установке WordPress. Обычно этот шаблон называется Kubrik, и пользуется немалой популярностью. На рис. 11.1 показан пример обычной страницы, использующей тему Kubrik.



Рис. 11.1. Пример темы Kubrik, используемой в WordPress по умолчанию

Как видно из иллюстрации, у страницы есть основной столбец, заголовок и боковая панель. Самой главной областью является заголовок; именно в нем идет поиск публикаций и добавление новой информации. Рассмотрим приведенную в листинге 11.1 упрощенную версию HTML, используемую в качестве структуры этого блога.

Листинг 11.1. Упрощенная версия HTML, сгенерированного темой Kubrik и WordPress

```
<html>
<head>
  <title>Бесконечный Wordpress</title>
  <script>
    <!-- Сюда будет помещен наш сценарий -->
  </script>
</head>
<body>
  <div id="page">
    <div id="header">
      <!--Содержимое заголовка -->
    </div>
    <div id="content">

      <!-- Первая публикация -->
      <div class="post">
        <!-- Заголовок публикации -->
```

```

<h2><a href=" /test/?p=1">Тестовая публикация</a></h2>
<small>24 октября 2006 года</small>

<div class="entry">
  <!-- Содержимое публикации -->
</div>

<p class="postmetadata">
  <a href="/test/?p=1#comments">Комментарии</a></p>
</div>

<!-- Другие публикации ... -->

</div>
</div>
</body>
</html>

```

Следует заметить, что все публикации блога помещены в `<div>`-контейнеры с ID, значение которого равно «content». Кроме этого, каждая публикация обладает специфическим форматом, который придает ей определенную структуру. С расчетом на это нужно создать простой набор DOM-функций, которые можно будет выполнить для получения некоторых данных и внедрения их на страницу блога. В листинге 11.2 показаны DOM-операции, необходимые для завершения страницы.

Листинг 11.2. Операции DOM для добавления HTML, необходимого для завершения страницы

```

// Загрузка новых публикаций производится в <div>-контейнер с ID, значение
// которого равно "content"
var content = document.getElementById("content");

// Мы собираемся осуществить последовательный перебор всех публикаций
// в RSS-потоке
var items = rss.getElementsByTagName("item");
for ( var i = 0; i < items.length; i++ ) {

  // Извлечение ссылки, заголовка и описательных данных из каждого
  // элемента потока, относящегося к публикации
  var data = getData( items[i] );

  // Создание нового <div>-контейнера для хранения публикации
  var div = document.createElement("div");
  div.className = "post";

  // Создание заголовка публикации
  var h2 = document.createElement("h2");

  // Здесь содержится заголовок элемента потока и имеется ссылка,
  // указывающая на публикацию.
  h2.innerHTML = "<a href='" + data.link + "'>" + data.title + "</a>";

```

```

// Добавление этого содержимого к <div>-контейнеру публикации
div.appendChild( h2 );

// Теперь создадим <div>, в котором будет содержаться публикация
var entry = document.createElement("div");
entry.className = "entry";

// Добавим в <div> содержимое публикации
entry.innerHTML = data.desc;
div.appendChild( entry );

// В завершение добавим нижнюю часть, содержащую ссылку возврата
var meta = document.createElement("p");
meta.className = "postmetadata";

var a = document.createElement("a");
a.href = data.link + "#comments";
a.innerHTML = "Комментарий";
meta.appendChild( a );

div.appendChild( meta );

// Помещение новой публикации в документ
content.appendChild( div );
}

```

Но все эти DOM-операции мало что значат, если вы не знаете, с какими данными имеете дело. В следующем разделе мы рассмотрим данные, которые будут переданы нам с сервера, и как их можно будет вставить в документ, используя DOM-операции.

Источник данных

WordPress предоставляет простые средства доступа к данным публикаций. Все блоги WordPress включают исходный RSS-поток, который можно использовать для просмотра десяти наиболее свежих публикаций. Но только этого одного будет не достаточно, нам нужен доступ ко всем публикациям, для чего следует вернуться к самому началу веб-сайта. К счастью, именно для этого имеется одно скрытое свойство, которым можно воспользоваться.

URL для RSS-потока обычно выглядит в WordPress следующим образом: `/blog/?feed=rss`; но если к нему добавить дополнительный параметр, `/blog/?feed=rss&paged=N`, то можно углубиться в исторические данные блога (с `N` равным 1 отображаются десять самых последних публикаций, с 2 — десять предыдущих, и т. д.). В листинге 11.2 показан пример того, как выглядит RSS-поток, содержащий данные публикаций.

Листинг 11.3. XML RSS-поток, возвращаемый WordPress, который содержит десять публикаций в четко отформатированной структуре

```

<?xml version="1.0" encoding="UTF-8"?>
<rss version="2.0">

```

```

<channel>
  <title>Test Wordpress Web log</title>
  <link>http://someurl.com/test/</link>
  <description>Test Web log.</description>
  <pubDate>Fri, 08 Oct 2006 02:50:23 +0000</pubDate>
  <generator>http://wordpress.org/?v=2.0</generator>
  <language>en</language>

  <item>
    <title>Test Post</title>
    <link>http://someurl.com/?p=9</link>
    <pubDate>Thu, 07 Sep 2006 09:58:07 +0000</pubDate>
    <dc:creator>John Resig</dc:creator>
    <category>Uncategorized</category>
    <description><![CDATA[Сюда помещается содержимое публикации...]]>
    </description>

  </item>

  <!-- Множество других публикаций -->
</channel>
</rss>

```

Благодаря подключению к RSS-поток вы получаете четко отформатированный XML-файл, с которым можно работать (и по которому очень удобно перемещаться с помощью JavaScript). В листинге 11.4 показан код, необходимый для прохода по RSS XML-документу и извлечения из него всей важной информации.

Листинг 11.4. Извлечение информации о публикации из XML RSS-потока

```

// Мы собираемся осуществить последовательный перебор всех публикаций,
// имеющих в RSS-потоке
var items = rss.getElementsByTagName("item");

for ( var i = 0; i < items.length; i++ ) {

  // Извлечение из <item>-элемента RSS-потока заголовка, описания
  // и ссылки
  var title = elem.getElementsByTagName("title")[0].firstChild.nodeValue;
  var desc =
    elem.getElementsByTagName("description")[0].firstChild.nodeValue;
  var link = elem.getElementsByTagName("link")[0].firstChild.nodeValue;
}

```

Теперь, когда у нас есть цельный источник данных и четкая структура для вставки результатов в HTML-документ, настало время склеить все вместе с помощью Ajax-запроса и определения наступления некоторых основных событий.

Определение наступления событий

Основным действием, которое пользователь должен совершить для активации нашего сценария, будет простая прокрутка ближе к концу страницы. При каждом смещении области просмотра браузера нам нужно проверять, не приблизилось ли оно к нижней части страницы.

Сценарий управления этим процессом относительно прост; нужно лишь привязать отдельный обработчик события к событию прокрутки окна. Тогда будет известно о каждом случае перемещения пользователем окна просмотра по странице (что может происходить и возле нижней части окна). Все, что нужно будет сделать сводится к применению нескольких методов, показанных в главе 7 с целью определения, где именно находится пользовательская область просмотра относительно страницы: `pageHeight` (для определения высоты всей страницы), `scrollTop` (для определения текущего положения, до которого прокручена вершина области просмотра), и `windowHeight` (для определения высоты области просмотра). Все они показаны в листинге 11.5.

Листинг 11.5. Определение положения пользовательской области просмотра

```
// Мы собираемся определить, не пора ли подгружать дополнительное
// содержимое в зависимости от того, где на странице находится
// пользовательская область просмотра
window.onscroll = function(){
    // Проверка положения области просмотра на странице
    if ( curPage >= 1 && !loading &&
        pageHeight() -scrollTop() -windowHeight() < windowHeight() ) {
        // Запрос RSS XML-потока с использованием Ajax
    }
};
```

Теперь у нас есть все необходимые компоненты, осталось только добавить Ajax-запрос, чтобы извлечь данные, необходимые для того, чтобы все как следует заработало.

Запрос

Ядро всего приложения связано с использованием Ajax-запросов для динамической загрузки новых блоков публикаций, которые затем можно будет вставлять в страницу. Нужный нам запрос довольно прост: отправляем на определенный URL (указывающий на следующий блок публикаций) HTTP GET-запрос и извлекаем находящийся там XML-документ. Именно это и делает код, показанный в листинге 11.6, в котором используется полноценная Ajax-функция (из главы 10).

Листинг 11.6. Ajax-запрос для загрузки нового блока публикаций

```
// Загрузка публикаций с использованием доступной нам функции ajax()
ajax({

    // Мы запрашиваем простую веб-страницу, поэтому используем GET
    type: "GET",

    // Ожидается RSS-поток, представленный XML-файлом
    data: "xml",

    // Получение RSS-потока N-ной страницы. При первоначальной загрузке
    // нашей страницы мы находимся на странице '1', поэтому при переходе
    // к предыдущему периоду времени мы начинаем со страницы 2
    url: " ./?feed=rss&paged=" + ( ++curPage ),
```

```

// Отслеживание успешного завершения извлечения RSS-потока
onSuccess: function( rss ){
    // Проход по RSS XML-документу, используя его DOM
}
});

```

Теперь после создания механизма составления запроса страницы, следует связать все вместе в единый пакет, который без особого труда можно будет поместить непосредственно в ваш WordPress-блог.

Результат

Объединение кода построения DOM с кодом прохода по RSS XML, наверное, самая простая часть этого приложения, но при объединении с определителем события прокрутки и Ajax-запросом, вы получаете все компоненты, необходимые для привлекательного дополнения вашего блога — возможность непрерывной прокрутки по всем публикациям блога не покидая при этом страницу. В листинге 11.7 показан полный код, необходимый для усовершенствования блога WordPress путем добавления этих функциональных возможностей.

Листинг 11.7. Код JavaScript, необходимый для предоставления блогу WordPress возможностей бесконечной страницы

```

// Отслеживание на какой "странице" содержимого мы находимся в данный момент
var curPage = 1;

// Предотвращение двойной загрузки страницы за один сеанс работы
var loading = false;

// Отслеживание необходимости загрузки дополнительного содержимого в
// зависимости от того места страницы, которое просматривается
// пользователем
window.onscroll = function(){
    // Перед загрузкой дополнительного содержимого нужно проверить
    // 1) Что мы не на последней странице содержимого.
    // 2) Что мы только что уже не загружали каких-нибудь новых
    //    публикаций.
    // 3) Что мы собираемся загружать только новые для нас публикации,
    //    прокрутка достигла нижней части страницы
    if ( curPage >= 1 && !loading
        && pageHeight() -scrollY() -windowHeight() < windowHeight() ) {

        // Запоминание того, что мы приступили к загрузке новых публикаций.
        loading = true;

        // Загрузка публикаций, с использованием доступной нам
        // функции ajax()
        ajax({

            // Мы запрашиваем простую веб-страницу, поэтому используем GET
            type: "GET",

```

```

// Ожидается RSS-поток, представленный XML-файлом
data: "xml",

// Получение RSS-потока N-ной страницы. При первоначальной
// загрузке нашей страницы мы находимся на странице '1',
// поэтому при переходе к предыдущему периоду времени мы
// начинаем со страницы 2
url: "../?feed=rss&paged=" + ( ++curPage ),

// Отслеживание успешного завершения извлечения RSS-потока
onSuccess: function( rss ){

    // Загрузка новых публикаций в <div>,
    // у которого ID имеет значение "content"
    var content = document.getElementById("content");

    // Мы собираемся осуществить последовательный перебор
    // всех публикаций, имеющихся в RSS-потоке
    var items = rss.getElementsByTagName("item");
    for ( var i = 0; i < items.length; i++ ) {

        // помещение новой публикации в документ
        content.appendChild( makePost( items[i] ) );

    }

    // Если из XML-документа уже больше нечего извлекать,
    // мы должны вернуться назад, насколько это возможно
    if ( items.length == 0 ) {
        curPage = 0;
    }
},

// Как только запрос будет завершен, можно будет
// снова осуществить попытку загрузки новых публикаций
onComplete: function(){
    loading = false;
}
});
}
};

// Функция, предназначенная для создания полной DOM-структуры отдельной
// публикации
function makePost( elem ) {
    // Извлечение из каждого элемента потока публикаций ссылки, заголовка
    // и описания данных
    var data = getData( elem );

```

```

// Создание нового <div>-контейнера для содержимого публикации
var div = document.createElement("div");
div.className = "post";

// Создание заголовка публикации
var h2 = document.createElement("h2");

// Здесь содержится заголовок элемента потока и имеется ссылка,
// указывающая на публикацию.
h2.innerHTML = "<a href='" + data.link + "'>" + data.title + "</a>";

// Добавление этого содержимого к <div>-контейнеру публикации
div.appendChild( h2 );

// Теперь создадим <div>, в котором будет содержаться публикация
var entry = document.createElement("div");
entry.className = "entry";

// Добавим в <div> содержимое публикации
entry.innerHTML = data.desc;
div.appendChild( entry );

// В завершение добавим нижнюю часть, содержащую ссылку возврата
var meta = document.createElement("p");
meta.className = "postmetadata";
meta.innerHTML = "<a href='" + data.link + "#comments'" + ">Комментарий</a>";
div.appendChild( meta );

return div;
}

// Простая функция для извлечения данных из DOM-элемента
function getData( elem ) {
    // Мы собираемся вернуть данные в виде четко отформатированного объекта
    return {
        // Извлечение из элемента <item> RSS-потока заголовка, описания и
        // ссылки
        title: elem.getElementsByTagName("title")[0].firstChild.nodeValue,
        desc:
            elem.getElementsByTagName("description")[0].firstChild.nodeValue,
        link: elem.getElementsByTagName("link")[0].firstChild.nodeValue
    };
}

```

Будет вполне достаточно добавить этот код в верхнюю часть заголовка вашего файла шаблона WordPress, чтобы получить результат, похожий на тот, что показан на рис. 11.2. Заметьте, что полоса прокрутки имеет довольно небольшой запас (что может свидетельствовать о завершении динамической загрузки на страницу дополнительных публикаций).

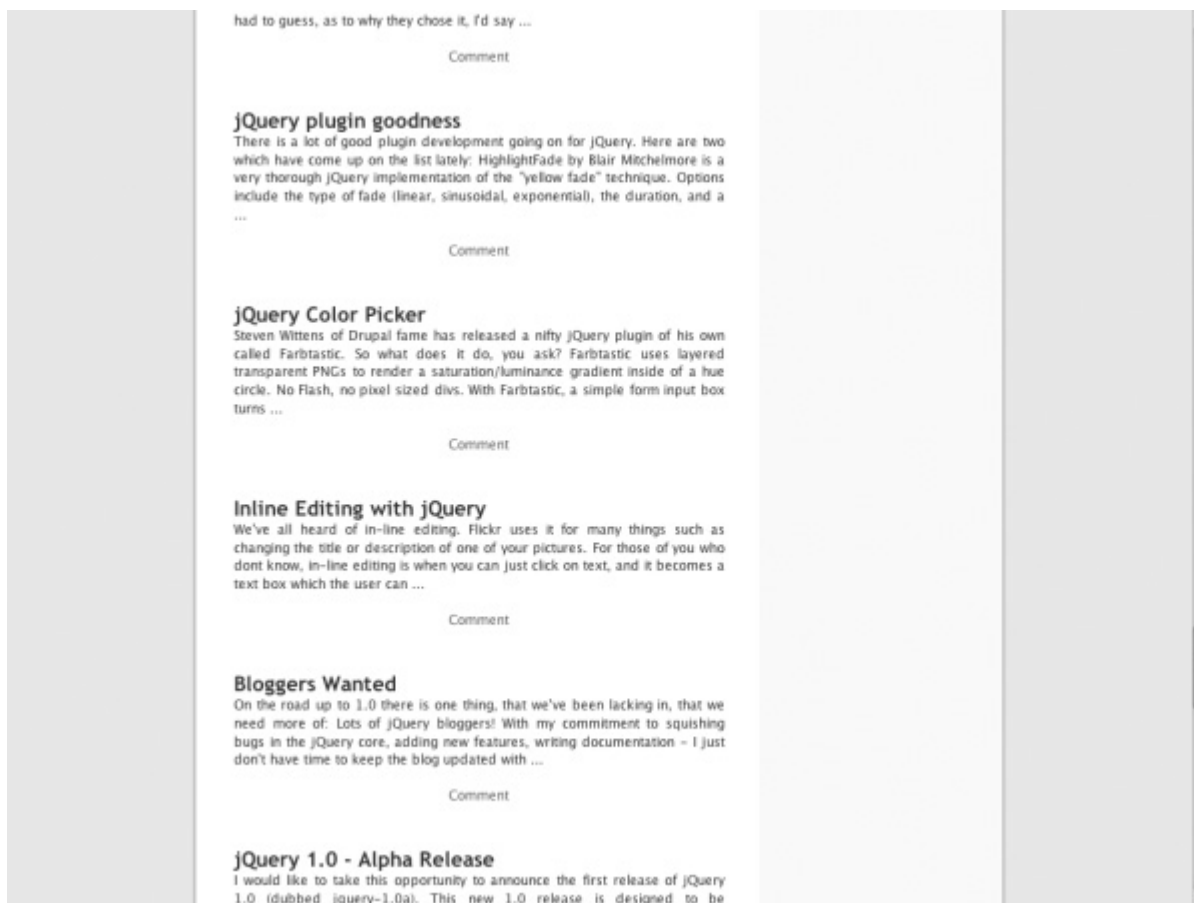


Рис. 11.2. Пример дополнительного содержимого, загруженного для просмотра в процессе дальнейшей прокрутки страницы вниз

Динамическая загрузка содержимого является обычным примером применения Ajax. Практически всегда это ведет к облегчению для пользователя процесса просмотра страницы, и облегчению нагрузки на сервер. В любом случае количество приложений, где будет востребована вставка дополнительного содержимого не ограничивается лишь одними блогами. Любые веб-приложения могут получить от этой технологии весьма существенную выгоду, и мы более подробно рассмотрим этот аспект в следующей главе.

А в следующем разделе мы рассмотрим еще один пример загрузки динамического содержимого, в котором используется та же Ajax-технология, но на этот раз мы создадим для просмотрщиков вашего блога возможность наблюдать за его ведением в режиме реального времени.

Наблюдение за ведением блога в режиме реального времени

Теперь, после того, как была проделана вся тяжелая работа по созданию динамического отображения публикаций, извлечению опубликованных данных и их разбору, мы можем рассмотреть другое приложение, польза от которого проявляется именно в блоге WordPress.

Обычно в блогах отсутствует возможность немедленного отображения новостей и обновления. Пользователям предоставляется статическая страница с перечнем самых последних публикаций блога, а если им потребуются дополнительные обновления, они будут вынуждены перезагрузить страницу в своем браузере. Но существует масса примеров, когда блоггер захочет немедленно сообщить пользователю какую-нибудь информацию, если тот все еще просматривает его страницу. Теоретически пользователь должен иметь возможность загрузить страницу, оставить ее открытой, чуть позже вернуться к ее просмотру, и увидеть новую публикацию.

И в этом случае мы можем найти технологии Ajax весьма хорошее применение. Для загрузки нового содержимого из вашего RSS-потока можно воспользоваться той же самой технологией, которая была использована в предыдущем разделе. Рассмотрим следующий порядок действий, необходимых для придания обычному блогу режима работы в реальном времени:

- Извлечение перечня самых свежих публикаций блога за определенный промежуток времени (к примеру, за одну минуту).
- Обнаружение еще не отображенных публикаций.
- Добавление этих публикаций в начало страницы.

Реализация этого замысла, показанная в листинге 11.8, также проста, как и вышеперечисленный порядок действий.

Листинг 11.8. Реализация блога, обновляемого на лету из RSS-потока, основанного на формате XML

```
// Мы собираемся совершать повторные попытки загрузить новое содержимое
// страницы через определенный интервал времени
setInterval(function(){

    // Загрузка публикации с использованием доступной нам функции ajax()
    ajax({

        // Мы запрашиваем простую веб-страницу, поэтому используем GET
        type: "GET",

        // Ожидается RSS-поток, представленный XML-файлом
        data: "xml",

        // Получение текущего RSS-потока (содержащего самые свежие
        // публикации)
        url: "./?feed=rss&paged=1",

        // Отслеживание успешного завершения извлечения RSS-потока
        onSuccess: function( rss ){

            // Загрузка новых публикаций в <div>,
            // у которого ID имеет значение "content"
            var content = document.getElementById("content");

            // Получение URL самой последней публикации (чтобы убедиться,
            // что мы не работаем с дубликатами публикаций)
            var recentURL =
                content.getElementsByTagName("h2")[0].firstChild.href;

            // Мы собираемся осуществить последовательный перебор
            // всех публикаций, имеющих в RSS-потоке
            var items = rss.getElementsByTagName("item");

            // Мы собираемся поместить все новые публикации
```

```

// в отдельный массив

var newItems = [];

// Проход по всем элементам
for ( var i = 0; i < items.length; i++ ) {

    // Принудительное прекращение цикла при
    // "старой" публикации
    if ( getData( items[i] ).link == recentURL )
        break;

    // Добавление нового элемента к временному массиву
    newItems.push( items[i] );
}

// Последовательный перебор всех новых публикаций
// в обратном порядке, чтобы обеспечить правильный порядок
// их размещения на веб-сайте
for ( var i = newItems.length -1; i >= 0; i-- ) {
    // Помещение в документ новой публикации
    content.insertBefore( makePost( newItems[i] ),
                          content.firstChild );
}
}
});
// Загрузка нового содержимого страницы раз в минуту
}, 60000 );

```

Когда этот сценарий будет добавлен в ваш шаблон WordPress (наряду с тем кодом, который был разработан в первой части главы при создании бесконечной страницы) вы получите результат, похожий на тот, что изображен на рис. 11.3.



Рис. 11.3. WordPress вставляет отдельную новую публикацию впереди других, более старых публикаций без обновления страницы

Воспользовавшись последним усовершенствованием, вы сможете превратить простой блог в платформу, работающую в режиме реального времени. Следующим в очереди на «оживление» может стать форум. Добавьте этот сценарий к вашему веб-сайту, и ваши читатели будут получать обновления по мере их публикации, без обновления страницы.

Вывод

Наиболее важным из рассмотренных понятий, которое можно вынести из этой главы, является то, что связанные с Ажас технологии позволяют вам придумывать новые режимы работы обычных статических приложений. Поскольку обработка XML-документов и их преобразование в полезные фрагменты HTML существенно упростились, вы имеете полную возможность их реализации в своих веб-приложениях.

В этой главе мы создали два дополнения к обычной блог-платформе на основе WordPress. Мы реально избавились от необходимости перехода к более ранним публикациям по старомодным ссылкам и с разбиением содержимого на разные страницы. Вместо этого мы загружаем блоки публикаций в динамическом режиме, по мере просмотра страницы пользователем. Кроме этого, при появлении новой публикации в период просмотра страницы пользователем, она будет добавлена к странице, позволяя пользователю продолжать чтение, не покидая страницу. Оба этих усовершенствования позволяют улучшить восприятие просмотра, делая его более динамичным и активным, и не разбивать публикации на несколько страниц.

В следующей главе мы создадим еще одно расширение за счет использования функциональных возможностей Ажас: поиск с автозаполнением.

Глава 12 Поиск автозаполнения

Наиболее важными функциональными преимуществами, предоставляемыми Ajax-технологией являются возможности создания пользовательских интерфейсов с высокой степенью взаимодействия, многие из которых ранее были недоступны. Примером одного из таких новых пользовательских элементов интерфейса может служить так называемое поле *поиска автозаполнения*. Это поле во многом похоже на обычное текстовое поле, но оно автоматически завершает условия поиска, как только вы начинаете их ввод. Это означает, что по мере набора поискового запроса, клиент посылает запрос на сервер (в фоновом режиме), и пытается быстро предоставить вам наиболее точные результаты.

В этой главе мы рассмотрим все компоненты, необходимые для построения полноценного поиска автозаполнения. Сначала мы рассмотрим, как построить и оформить страницу, потом узнаем, как отслеживать пользовательский ввод в текстовое поле, а затем свяжем все это с простым Ajax-запросом, обращающимся к простой базе данных на стороне сервера.

Примеры поиска автозаполнения

Поле поиска автозаполнения может проявлять себя несколькими разными способами. Например, в Google есть версия автозаполнения его поля поиска, называемая Google Suggest (<http://www.google.com/webhp?complete=1>). Когда вы начинаете набирать в поле запрос на поиск, эта система показывает вам другие запросы на поиски, наиболее часто проводившиеся другими пользователями, которые начинаются с тех же символов, что и только что набранные. Пример поиска автозаполнения показан на рис. 12.1.



Рис. 12.1. Пример обычных результатов автозаполнения

Другим весьма популярным примером является Instant Domain Search (<http://instantdomainsearch.com/>). Это специализированное приложение по мере набора предоставляет сведения о доступности покупаемого доменного имени. Этот вариант отличается от той реализации, которую представила компания Google тем, что он автоматически завершает поиск, а не заполняет строку самого запроса. Это означает, что по мере набора искомого доменного имени сервер автоматически в фоновом режиме завершает ваш запрос, выдавая соответствующие ему результаты. Пример работы этой системы показан на рис. 12.2.

javascr	.com is taken.	.net is available!	.org is available!
Suggestions		1 & 1: \$5.99	1 & 1: \$5.99
Make Offer (Sedo)		GoDaddy: \$8.95	GoDaddy: \$8.95
Backorder (GoDaddy)		Yahoo: \$9.95	Yahoo: \$9.95
WHOIS		Dreamhost: \$9.95	Dreamhost: \$9.95
Hacks			
URL Trends			
Visit Site			

Рис. 12.2. Пример поиска автозаполнения в системе Instant Domain Search


Последний приводимый пример, который наиболее близок к тому, что мы будем создавать — это механизм автозаполнения, предоставляемый Интернет-службой закладок del.icio.us (<http://del.icio.us/>). Эта служба предоставляет средства, с помощью которых вы можете связать ссылки с определенными словами, позволяя произвести автозаполнение нескольких слов в единственном поле ввода текста. Пример работы этого механизма показан на рис. 12.3.

url	http://beppu.lbox.org/articles/2006/08/22/the-decorator-pattern-for-javasc
description	The Decorator Pattern for JavaScript
notes	
tags	programming development j
suggestions	javascript java jobs jquery json jb js journalism japan

Рис. 12.3. Пример работы автозаполнения del.icio.us, завершающего новый тег





Теперь мы вплотную приблизились к созданию собственного механизма поиска автозаполнения. Это будет простая форма для отправки сообщений группе друзей, расположенная на веб-сайте. Автозаполняемое поле будет вести себя очень похоже на поле в del.icio.us, в том, что это будет поле для ввода имен пользователей, которое при помощи Ajax может быть автоматически заполнено в фоновом режиме. У вас будет возможность автозаполнения каждого из пользовательских имен ваших друзей, на основе тех имен, которые хранятся в центральной базе данных, и помещения их в список пользовательских имен, разделенных запятой, используемый для отправки им сообщений. Пример того эффекта, который мы собираемся получить, показан на рис. 12.4.

Send Message

From:  **John Resig**

To:

Suggestions:

-  **jason** (Jason S)
-  **john** (John R)
-  **josh** (Josh K)
-  **julia** (Julia W)

» Send

Рис. 12.4. Пример автозаполнения пользовательского имени после ввода одной буквы

Построение страницы

Первым шагом к созданию нашего поля поиска автозаполнения станет построение формы, в которой будет размещена полная установка. Эта страница по структуре будет напоминать простую форму для отправки сообщения группе пользователей веб-сайта. В дополнение к обычной форме отправки сообщений (состоящей из поля Кому и области сообщения) в эту страницу необходимо включить три важных аспекта:

Поле ввода текста: Нам нужно обеспечить, чтобы его свойство `autocomplete` было выключено. Тем самым мы отключим включенный по умолчанию механизм автозаполнения браузера (который использует для автозаполнения все, что было ранее набрано в области ввода):

```
<input type="text" id="to" name="to" autocomplete="off"/>
```

Загружаемое изображение: Это небольшое, вращающееся изображение, накладываемое поверх поля ввода текста, чтобы показать, когда с сервера загружаются новые данные:

```

```

Область результатов: Все результаты, возвращаемые с сервера, будут помещаться в область результатов и отображены по требованию. Физические результаты будут возвращаться в виде набора ``-элементов, в каждом из которых будет содержаться информация об отдельном пользователе:

```
<div id="results"><div class="suggest">Suggestions:</div><ul></ul></div>
```

И индикатор загрузки, и область результатов будут включены в страницу с помощью JavaScript. Полный код HTML для нашей страницы показан в листинге 12.1.

Листинг 12.1. Полный код HTML для формы отправки сообщений пользователям

```
<html>
<head>
  <script src="dom.js"></script>
  <script src="delay.js"></script>
  <script src="script.js"></script>
  <link rel="stylesheet" href="style.css"/>
</head>
<body>
  <form action="" method="POST" id="auto">
    <div id="top">
      <div id="mhead"><strong>Send Message</strong></div>
      <div class="light">
        <label>From:</label>
        <div class="rest from">
          
          <strong>John Resig</strong>
        </div>
      </div>
      <div class="query dark">
        <label>To:</label>
        <div class="rest">
```

```

        <input type="text" id="to" name="to" autocomplete="off"/>
    </div>
</div>
<div class="light"><textarea></textarea></div>
<div class="submit"><input type="submit" value="&raquo;
                                Send"/></div>
</div>
</form>
</body>
</html>

```

На рис. 12.5 представлена копия экрана с видом страницы, имеющей полное стилевое оформление.

Рис. 12.5. Вид макета формы после простого стилевого оформления. Результат нашего The result of your form mockup with some simple styling

Теперь, когда у нас есть установленная и готовая к пользовательскому вводу форма, настало время приступить к следующему шагу — отслеживанию вводимой пользователем информации в поле ввода имени пользователя, и соответствующего на нее реагирования.

Отслеживание ввода с клавиатуры

Одним из важных аспектов поиска автозаполнения является создание естественно воспринимаемого интерфейса, с помощью которого пользователь сможет вводить данные и получать его расширенный вариант. Для большинства поисковых вводов основная часть интерфейса поиска обращается к отдельному полю ввода текста. Вы будете и в дальнейшем использовать этот интерфейс, включая ваши собственные приложения.

При конструировании этого механизма ввода нужно учесть некоторые моменты, чтобы быть уверенным, что предоставленный уровень взаимодействия близок к идеалу:

- Нужно обеспечить включение поиска автозаполнения через подходящие интервалы времени, когда ответ дается достаточно быстро, чтобы пользователь сумел на него среагировать.
- Нужно обеспечить, чтобы поиск на сервере осуществлялся как можно медленнее. Чем быстрее осуществляется поиск, тем большая нагрузка ложится на сервер.
- Нужно обеспечить знание подходящего момента для осуществления нового поиска автозаполнения и открытия результатов; для отказа от поиска и открытия старых результатов; и для скрытия результатов.

Теперь, изложив эти пункты, мы можем более точно определить взаимодействие с пользователем, которое хотим реализовать:

- Результаты автозаполнения должны отображаться на основе того, что пользователь набрал в поле ввода текста. Кроме того, чтобы избежать слишком неоднозначных результатов поиска, пользователь должен предоставить для него минимальное количество символов.
- Результаты поиска должны вызываться через строго определенные интервалы времени (чтобы защитить сервер от перегрузки от действий слишком быстро печатающих пользователей), но только при изменениях введенного содержимого.
- Подборка результатов должна быть показана или скрыта в зависимости от того, имеет ли элемент ввода пользовательский фокус (к примеру, если пользователь убрал фокус с элемента ввода, результаты должны быть скрыты).

С учетом всех этих пунктов мы можем разработать отдельную функцию для привязки требуемого взаимодействия к отдельному полю ввода текста. В листинге 12.2 показана функция, которая может быть использована для достижения желаемых результатов. Эта функция обрабатывает только случаи, при которых должен проводиться поиск или при которых результаты должны быть показаны или скрыты, а не сам поиск или визуальные результаты (мы добавим все это чуть позже).

Листинг 12.2. Функция привязки поиска автозаполнения к полю ввода текста

```
function delayedInput(opt) {
  // Количество времени ожидания до отслеживания нового пользовательского
  // ввода
  opt.time = opt.time || 400;

  // Минимальное количество символов, ожидаемых до запуска запроса
  opt.chars = opt.chars != null ? opt.chars : 3;

  // Обратный вызов, запускаемый, когда должны быть показаны
  // всплывающие результаты, и, возможно, когда должен быть сделан
  // новый запрос
  opt.open = opt.open || function(){};

  // Обратный вызов, выполняемый, когда всплывающие результаты должны
  // быть закрыты
  opt.close = opt.close || function(){};

  // Фокус поля должен быть принят во внимание для открытия
  // (закрытия) всплывающих результатов
  opt.focus = opt.focus !== null ? opt.focus : false;

  // Запоминание исходного значения, с которым мы начинаем
  // работать
  var old = opt.elem.value;

  // и текущего состояния открыто-закрыто всплывающего результата
  var open = false;

  // Проверка изменений ввода в заданный интервал времени
```

```

setInterval(function(){
    // Новое вводимое значение
    var newValue = opt.elem.value;

    // Количество введенных символов
    var len = s.length;

    // Быстрая проверка на изменения значения со времени последней
    // проверки ввода
    if ( old != newValue ) {

        // Если введено недостаточно символов, и всплывающие результаты
        // в данный момент открыты
        if ( v < opt.chars && open ) {

            // Закрыть отображение
            opt.close();

            // И запомнить, что оно закрыто
            open = false;

            // В противном случае, если было введено минимальное
            // количество символов пока оно больше одного символа
        } else if ( v >= opt.chars && v > 0 ) {

            // открыть всплывающие результаты с текущим значением
            opt.open( newValue, open );

            // Запомнить, что всплывающие результаты на данный
            // момент открыты
            open = true;
        }

        // Сохранение на будущее текущего значения
        old = newValue;
    }
}, opt.time );

// Отслеживание нажатия клавиши
opt.elem.onkeyup = function(){
    // Если в результате нажатия клавиши символов больше не осталось,
    // закрыть всплывающие результаты
    if ( this.value.length == 0 ) {
        // Закрыть результаты
        opt.close();

        // Запомнить, что они закрыты
        open = false;
    }
}

```

```

    }
};

// Если мы также проверяем пользовательский фокус (для управления
// открытием-закрытием всплывающих результатов)
if ( opt.focus ) {
    // Отслеживание перемещения пользователя за пределы элемента ввода
    opt.elem.onblur = function(){
        // Если результаты открыты
        if ( open ) {
            // закрыть их
            opt.close();

            // и запомнить, что они закрыты
            open = false;
        }
    }

    // Отслеживание, когда пользовательский фокус вернется на элемент
    // ввода
    opt.elem.focus = function(){
        // если во всплывающих результатах есть какое-нибудь значение,
        // и они в данный момент закрыты
        if ( this.value.length != 0 && !open ) {
            // повторное открытие всплывающих результатов, но с пустым
            // значением
            // (это позволит 'открывающей' функции узнать, что
            // перезапрашивать новые результаты с сервера не нужно,
            // нужно их просто заново открыть).
            opt.open( '', open );

            // И запомнить, что они открыты
            open = true;
        }
    };
}
}
}

```

В листинге 12.3 показано, как в нашей реализации автозаполнения для отслеживания пользовательского ввода использовать простую функцию `delayedInput`.

Листинг 12.3. Использование универсальной функции `delayedInput()` в нашей реализации автозаполнения

```

// Инициализация задержки проверки ввода в поле
delayedInput({
    // Прикрепление к полю текстового ввода
    elem: id("to"),

    // Мы намереваемся приступить к поиску только после ввода

```

```

// одного символа
chars: 1,

// Когда текстовое поле теряет фокус, закрыть всплывающие результаты
focus: true,

// Обработка момента открытия всплывающего результата
open: function(q,open){
    // Извлечение последнего слова из списка слов, разделенных
    // запятыми
    var w = trim( q.substr( q.lastIndexOf(',')+1, q.length ) );

    // Обеспечение того, что мы работаем как минимум со словом
    if ( w ) {
        // Отображение анимации, свидетельствующей о загрузке
        show( id("qloading") );

        // Загрузка и обработка результатов с сервера
    }
},

// Когда нужно закрыть всплывающие результаты
close: function(){
    // Скрытие подборки результатов
    hide( id("results") );
}
});

```

Теперь, когда у нас есть универсальная функция для отслеживания пользовательского ввода, нужно решить задачу ее привязки к сценарию на стороне сервера, который будет предоставлять данные о пользователях, которые можно загрузить на ваш веб-сайт.

Извлечение результатов

Следующим основным аспектом построения поиска автозаполнения является загрузка данных, которые будут показаны пользователю. Для загрузки этих данных не требуется использование технологии Ajax (которую мы будем применять в этой конкретной реализации); вместо этого они могут быть написаны в виде структуры данных и загружены в страницу во время выполнения приложения.

Наша реализация автозаполнения требует в законченном виде только одного: пользователей. Их пользовательские имена будут отображены с сопутствующей информацией (включая полное имя пользователя и его значок). С расчетом на это намного легче будет просто вернуть с сервера HTML-фрагмент (в форме некоторого количества ``-элементов), содержащий всю необходимую информацию о соответствующих пользователях.

В листинге 12.4 показан простой Ajax-вызов, необходимый для загрузки с сервера фрагмента HTML во всплывающие результаты.

Листинг 12.4. AJAX-запрос для загрузки фрагмента HTML (содержащего информацию о пользователях) в подборку результатов для автозаполнения

```

// Создание запроса новых данных

```



```

ajax({
    // Создание простого GET-запроса к CGI-сценарию, возвращающему
    // HTML-блок, состоящий из элементов LI
    type: "GET",
    url: "auto.cgi?to=" + w,

    // Отслеживание возвращения HTML
    onSuccess: function(html) {
        // Вставка его в предназначенный для результатов UL-контейнер
        results.innerHTML = html;

        // и скрытие анимации загрузки
        hide( id("qloading") );

        // Обработка результатов...
    }
});

```

Вы должны обратить внимание на то, что в листинге 12.4 мы загружаем HTML-результаты из приложения серверной стороны под названием `auto.cgi`, которое воспринимает один аргумент — текущий текст, по которому ведется поиск (скорее всего это будет часть имени пользователя). Сценарий `auto.cgi`, написанный с использованием языка Perl, показан в листинге 12.5. Он ведет поиск соответствий в небольшом наборе данных, возвращая всех подходящих пользователей в длинный HTML-фрагмент.

Листинг 12.5. Простой Perl-сценарий, отыскивающий подходящих пользователей

```

#!/usr/bin/perl

use CGI;

# Извлечение из строки входящего запроса параметра 'q'
my $cgi = new CGI();
my $q = $cgi->param('to');

# Наша ограниченная "база данных" содержит пять пользователей,
# их пользовательские и полные имена.
my @data = (
    {
        user => "bradley",
        name => "Bradley S"
    },
    {
        user => "jason",
        name => "Jason S"
    },
    {
        user => "john",
        name => "John R"
    },

```

```

{
    user => "josh",
    name => "Josh K"
},
{
    user => "julia",
    name => "Julia W"
}
);

# Обеспечение выдачи правильного HTML-заголовка
print "Content-type: text/html\n\n";

# "Поиск" по данным
foreach my $row (@data) {

    # Поиск пользователей, соответствующих нашему аргументу
    # поиска автозаполнения
    if ( $row->{user} =~ /$q/i || $row->{name} =~ /$q/i ) {

        # Если пользователь соответствует условию, выдача
        # необходимого HTML
        print qq~<li id="$row->{user}">
            
            <div>
                <strong>$row->{user}</strong> ($row->{name})
            </div>
        </li>~;
    }
}

```

Результат, возвращаемый CGI-сценарием ничто иное, как HTML-фрагмент, содержащий -элементы, соответствующие каждому подходящему пользователю. В листинге 12.6 показан результат поиска для буквы j.

Листинг 12.6. Фрагмент HTML, возвращенный с сервера, представляющий несколько различных пользователей

```

<li id="jason">
    
    <div>
        <strong>jason</strong> (Jason S)
    </div>
</li><li id="john">
    
    <div>
        <strong>john</strong> (John R)
    </div>
</li><li id="josh">

```

```


<div>
  <strong>josh</strong> (Josh K)
</div>
</li><li id="julia">
  
  <div>
    <strong>julia</strong> (Julia W)
  </div>
</li>

```

Теперь, когда у нас есть небольшой набор данных, возвращенный HTML-фрагмент, и HTML, вводимый в веб-сайт, следующим логичным шагом будет добавление способов перехода пользователя по результатам.

Переход по списку результатов

Теперь, когда пользователь ввел какой-то текст в поле ввода, и с сервера были загружены некоторые результаты, настало время добавить способ перемещения пользователя по возвращенному набору результатов. В нашей реализации поиска автозаполнения мы собираемся предложить два различных способа перемещения по результатам: с помощью клавиатуры и с помощью мыши.

Перемещения с помощью клавиатуры

Перемещение по результатам с помощью клавиатуры по всей вероятности является наиболее важным аспектом реализации. Поскольку пользователь находится в процессе набора имени пользователя, он дает ему возможность если это необходимо завершить автозаполнение, оставляя руки на клавиатуре.

Нам нужна поддержка клавиши табуляции, чтобы завершить работу с текущим выбранным пользователем, и клавиш со стрелками вверх и вниз, для выбора различных пользователей из списка результатов. В листинге 12.7 показано, как этого можно будет достичь.

Листинг 12.7. Обработчик события нажатия клавиш перемещений

```

// Отслеживание ввода информации в поле
id("to").onkeypress = function(e) {
  // Получение всех пользователей из списка результатов
  var li = id("results").getElementsByTagName("li");

  // Когда нажата клавиша [TAB] или клавиша [Enter]
  if ( e.keyCode == 9 || e.keyCode == 13 ) {
    // Добавление пользователя к полю ввода текста

  // Если нажата клавиша вверх
  } else if ( e.keyCode == 38 )
    // Выбор предыдущего пользователя, или последнего пользователя
    // если мы находимся в начале списка)
    return updatePos( curPos.previousSibling || li[ li.length - 1 ] );

  // Если нажата клавиша вниз
  else if ( e.keyCode == 40 )
    // Выбор следующего пользователя, или первого пользователя (если мы

```

```

// находимся в конце списка)
return updatePos( curPos.nextSibling || li[0] );
};

```

Перемещение с помощью мыши

В отличие от перемещений с помощью клавиатуры, все перемещения, осуществляемые с помощью мыши должны иметь динамическую привязку при каждом возвращении с сервера нового набора результатов. Замысел перемещений с помощью мыши состоит в том, что при каждом прохождении указателя мыши над одним из ``-элементов, это элемент становится текущим «выбранным», и если на нем щелкнуть кнопкой, связанное с `` имя пользователя должно быть добавлено в поле ввода текста. В листинге 12.8 показан пример кода, с помощью которого можно реализовать этот замысел.

Листинг 12.8. Привязка события перемещения указателя мыши к ``-элементу

```

// При каждом прохождении указателя мыши над li,
// он становится текущим выбором пользователя
li[i].onmouseover = function(){
    updatePos( this );
};

// Если пользователь произвел щелчок,
li[i].onclick = function(){
    // добавить имя пользователя к полю ввода
    addUser( this );

    // и вернуть фокус на это поле
    id("to").focus();
};

```

Теперь, когда реализованы все перемещения, нужно завершить работу над основными компонентами поиска автозаполнения. Окончательный результат нашей работы показан в следующем разделе.

Окончательный результат

Мы завершили создание всех компонентов, необходимых для поиска автозаполнения: отслеживания пользовательского ввода, связь с сервером, и перемещение по результатам. Настало время связать все это воедино и поместить на страницу. В листинге 12.9 показан окончательный код JavaScript для полноценного поиска автозаполнения.

Листинг 12.9. Полный код JavaScript для поиска автозаполнения

```

domReady(function(){
    // Обеспечение скрытия всплывающих результатов в начале работы
    hide( id("results") );

    // Отслеживание уже введенных имен пользователей
    var doneUsers = {};

    // Отслеживание выбранного на данный момент пользователя
    var curPos;

```

```

// Создание изображения индикации загрузки
var img = document.createElement("img");
img.src = "indicator.gif";
img.id = "qloading";

// и добавление его к документу сразу за полем ввода
id("to").parentNode.insertBefore( img, id("to") );

// Создание области отображения результатов
var div = document.createElement("div");
div.id = "results";
div.innerHTML = "<div class='suggest'>Suggestions:</div><ul></ul>";

// и добавление ее за полем ввода
id("to").parentNode.appendChild( div );

// Отслеживание ввода в поле
id("to").onkeypress = function(e) {
    // Получение всех пользователей из набора результатов
    var li = id("results").getElementsByTagName("li");

    // Если нажата клавиша [TAB] или [Enter]
    if ( e.keyCode == 9 || e.keyCode == 13 ) {
        // Перезапуск списка текущих пользователей
        loadDone();

        // Если текущий выбранный пользователь отсутствует в списке
        // выбранных, добавление его к полю ввода
        if ( !doneUsers[ curPos.id ] )
            addUser( curPos );

        // Отключение обычной реакции на нажатие клавиши
        e.preventDefault();
        return false;
    }

    // Если нажата клавиша вверх
} else if ( e.keyCode == 38 )
    // Выбор предыдущего пользователя, или последнего пользователя
    // если мы находимся в начале списка)
    return updatePos( curPos.previousSibling ||
                      li[ li.length - 1 ] );

// Если нажата клавиша вниз
else if ( e.keyCode == 40 )
    // Выбор предыдущего пользователя, или последнего пользователя
    // если мы находимся в начале списка)
    return updatePos( curPos.nextSibling || li[0] );

```

```

};

// Инициализация задержки проверки ввода в поле
delayedInput({
  // Прикрепление к полю текстового ввода
  elem: id("to"),

  // Мы намереваемся приступить к поиску только после ввода
  // одного символа
  chars: 1,

  // Когда текстовое поле теряет фокус, закрыть всплывающие
  // результаты
  focus: true,

  // Обработка момента открытия всплывающего результата
  open: function(q, open) {
    // Извлечение последнего слова из списка слов, разделенных
    // запятыми
    var w = trim( q.substr( q.lastIndexOf(',')+1, q.length ) );

    // Обеспечение того, что мы работаем как минимум со словом
    if ( w ) {
      // Отображение анимации, свидетельствующей о загрузке
      show( id("qloading") );

      // Обеспечение, что ни один пользователь пока не
      // выбран
      curPos = null;

      // Получение UL, предназначенного для хранения всех
      // результатов
      var results = id("results").lastChild;

      // и его очистка
      results.innerHTML = "";

      // Создание запроса новых данных
      ajax({
        // Создание простого GET-запроса к
        // CGI-сценарию, возвращающему
        // HTML-блок с LI-элементами
        type: "GET",
        url: "auto.cgi?q=" + w,

        // Отслеживание поступления HTML
        onSuccess: function(html) {
          // Его вставка в UL результатов
          results.innerHTML = html;
        }
      });
    }
  }
});

```

```

// И скрытие загрузочной анимации
hide( id("qloading" ) );

// Повторная инициализация списка получаемых
// имен пользователей
loadDone();

// Последовательный перебор каждого
// возвращенного имени пользователя
var li = results.getElementsByTagName( "li" );
for ( var i = 0; i < li.length; i++ ) {

    // Если мы уже добавили пользователя,
    // удаление связанного с ним элемента LI
        if ( doneUsers [ li[i].id ] )
            results.removeChild( li[i--] );

    // Иначе привязка события к li с именем
    // пользователя
    else {
        // Как только указатель мыши
        // пользователя проходит над li,
        // установка связанного с ним имени
        // текущим именем пользователя
        li[i].onmouseover = function(){
            updatePos( this );
        };

        // При щелчке на имени пользователя
        li[i].onclick = function(){
            // Добавление имени в поле ввода
            addUser( this );

            // и возврат фокуса на поле ввода
            id("q").focus();
        };
    }
}

// Проход по списку имен пользователей
li = results.getElementsByTagName( "li" );

// Если имен не осталось (они все уже добавлены)
if ( li.length == 0 )
    // скрытие результатов
    hide( id("results" ) );

```

```

else {

    // Добавление к каждому оставшемуся
    // элементу имен пользователей классов
    // 'odd', чтобы придать списку
    // «полосатый» вид
    for ( var i = 1; i < li.length; i += 2 )
        addClass( li[i], "odd" );

    // Установка текущего выбранного имени
    // пользователя на первый элемент списка
    updatePos( li[0] );

    // и отображение результатов
    show( id("results") );
}
}
});
}
},

// Теперь всплывающие результаты должны быть скрыты
close: function(){
    // Скрытие набора результатов
    hide( id("results") );
}
});

function trim(s) {
    return s.replace(/^\s+/, "").replace(/\s+$/, "");
}

// Изменение подсветки текущего выбранного имени пользователя
function updatePos( elem ) {
    // Обновление позиции текущего выбранного элемента
    curPos = elem;

    // Получение всех li-элементов с именами пользователей
    var li = id("results").getElementsByTagName("li");

    // Удаления класса 'cur' из текущего выбранного элемента
    for ( var i = 0; i < li.length; i++ )
        removeClass( li[i], "cur" );

    // И добавление подсветки на текущий элемент имени
    // пользователя
    addClass( curPos, "cur" );
}

```



```

        return false;

    }

    // Повторная инициализация списка имен пользователей, который уже
    // добавлен пользователем в поле ввода

    function loadDone() {
        doneUsers = {};

        // Проход по списку имен пользователей (разделенных запятыми)
        var users = id("q").value.split(',');
        for ( var i = 0; i < users.length; i++ ) {

            // Сохранение имени пользователя (в качестве ключа) в
            // хэш-объекте
            doneUsers[ trim( users[i].toLowerCase() ) ] = true;
        }
    }

    // Добавление имени пользователя к полю ввода текста
    function addUser( elem ) {
        // Текстовое значение из поля ввода текста
        var v = id("to").value;

        // Добавление имени пользователя в конец содержимого поля ввода,
        // обеспечение его отделения знаком запятой
        id("to").value =
            ( v.indexOf(',') >= 0 ? v.substr(0, v.lastIndexOf(',') + 2 ) : '' )
            + elem.id + ", ";

        // Добавление имени пользователя к основному списку
        // (избавляющего от необходимости полной перезагрузки списка)
        doneUsers[ elem.id ] = true;

        // Удаление li-элемента с именем пользователя
        elem.parentNode.removeChild( elem );

        // и скрытие списка результатов
        hide( id("results") );
    }
});

```

На рис. 12.6 показано, как выглядит окончательный результат.

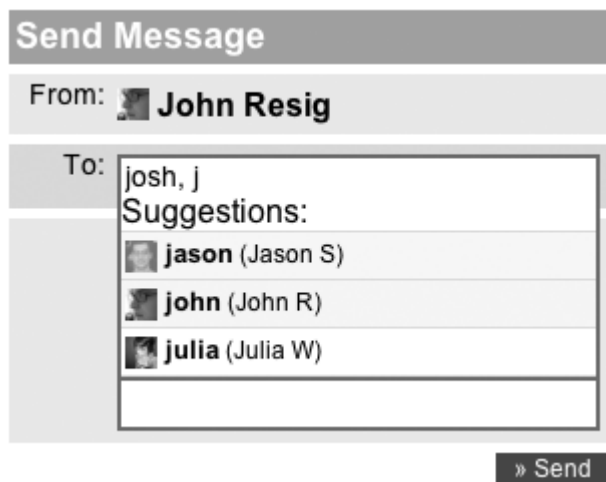


Рис. 12.6. Копия экрана поиска автозаполнения в действии при завершении выбора второго имени пользователя

Окончательный результат выглядит впечатляющим и весьма полезным в применении. Основные концепции поиска автозаполнения не отличаются особой сложностью или трудностью реализации, но собранные воедино они дают очень хороший интерактивный результат.

Вывод

Поиск автозаполнения создан в качестве хорошего дополнения практически к любому приложению. При этом появляется возможность помочь пользователю с вводом данных при работе почти с каждым текстовым полем.

В этой главе я охватил все аспекты создания поиска автозаполнения для ввода пользовательских имен в простую форму. Отдельно было рассмотрено, как захватить ввод текста пользователем, послать запрос сценарию на стороне сервера, забрать назад соответствующие данные и дать возможность пользователю осуществлять переход по результатам. Использование концепций, представленных в этой главе, и их адаптация к вашему конкретному случаю, принесет вашим пользователям весьма ощутимые выгоды.

Рабочая демонстрация этого примера, а также подробные инструкции по настройке оборудования на стороне сервера доступны на веб-сайте этой книги <http://www.jspro.org/>. Исходный код как всегда доступен в разделе Source Code/Download веб-сайта Apress <http://www.apress.com>.

Глава 13 Ajax Wiki

Появление на переднем крае веб-разработки различных исполнительных сред, основанных на концепции модель-представление-контроллер — MVC (к примеру Ruby on Rails и Django), навело меня на мысль, что настало время рассмотреть некоторые альтернативные языки программирования, используемые для разработки веб-приложений. В качестве примера, исследуемого в этой главе я выбрал простую браузерную wiki.

Что такое Wiki?

В соответствии с Wikipedia.org (бесспорно, самым популярным веб-сайтом, основанным на wiki-технологии), wiki — это разновидность веб-сайта, позволяющая любому посетителю быстро и просто добавлять, удалять, или же редактировать все содержимое, причем для этого иногда не требуется даже регистрации. Простота взаимодействия и работы превращает wiki в весьма эффективный инструмент для совместного создания письменного материала.

Кроме этого для пользовательской настройки wiki-записей предоставляется целая подборка средств форматирования. На рис. 13.1 показана главная страница wiki после нескольких правок, внесенных различными пользователями.

Уникальность этого конкретного примера заключается в том, что логика движка wiki целиком написана на JavaScript, именно код JavaScript посылает запросы к базе данных непосредственно на сервер. Этот учебный пример демонстрирует ряд основных концепций, необходимых для разработки современных веб-приложений, даже при том, что на сервере отсутствует обычная логика приложения.

Само приложение разбивается на три части: клиентскую, серверную и базу данных (как и в случаях с большинством Ajax-приложений), каждая из которых будет подробно рассмотрена. Клиентская часть отвечает за взаимодействие с пользователем и за управление интерфейсом. Серверная часть отвечает за управление связью между клиентской частью и источником данных.

Чтобы стало понятным, для чего конкретно предназначено это приложение, на что оно способно, и как работает, я собираюсь провести вас по каждой его особенности, и объяснить, как код этого приложения можно будет использовать в своих собственных разработках.

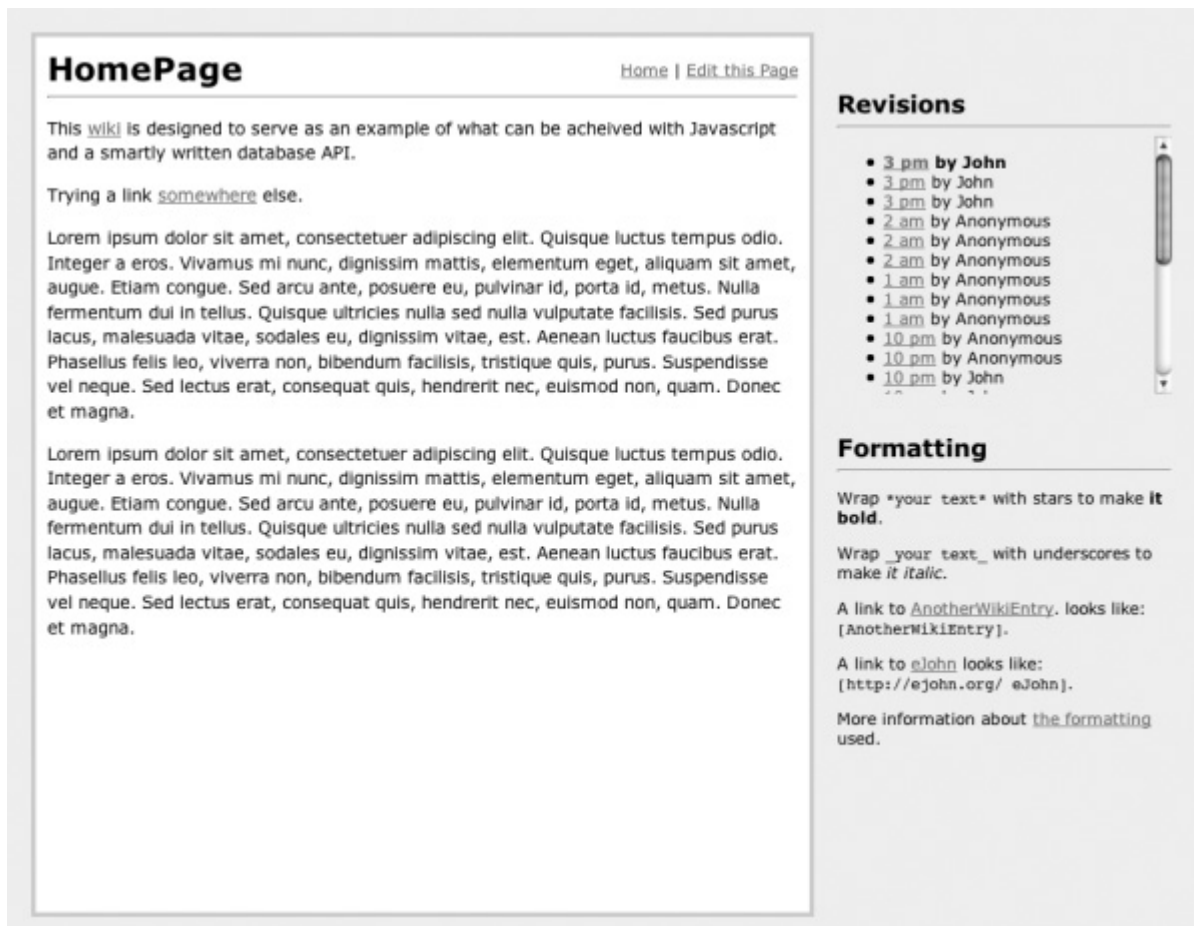


Рис. 13.1. Копия экрана, демонстрирующая wiki в действии

Обращение к базе данных

Каждая страница wiki должна быть кем-то создана и отредактирована. Это означает, что нам нужно где-то хранить все предоставленное пользователем содержимое, чтобы в дальнейшем получить к нему доступ. Наверное, наиболее правильным решением будет создание простой базы данных, в которой можно будет хранить все свои данные. Чтобы дать возможность клиентской стороне, написанной на JavaScript, обращаться к базе данных, нам нужно иметь на серверной стороне какой-нибудь код, занимающий место между клиентским кодом и базой данных wiki. На рис. 13.2 показано, как выглядит прохождение каждого запроса к базе данных. Этот процесс является общим для всех форм выполняемых операций (например, SELECT, INSERT и т.д.).

Связь клиентской стороны с базой данных нужна будет в двух случаях. Во-первых, для запроса к базе данных и извлечения всех правок конкретной страницы. Во-вторых, когда мы вставляем в базу данные новой правки. Порядок прохождения обоих этих запросов фактически один и тот же, что облегчает создание общего, упрощенного уровня связи, который может быть использован и в других JavaScript-приложениях. Еще одной приятной особенностью общего уровня связи является возможность легко пройти по всей технологической цепочке связи (показанной на рис. 13.2) и посмотреть, как JavaScript может связываться с базой данных.

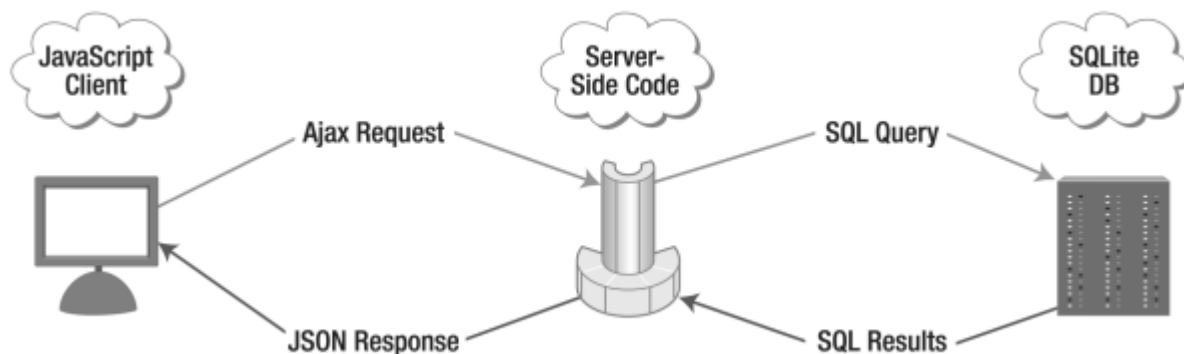


Рис. 13.2. Технологический процесс, используемый в приложении для выполнения запроса от клиента

ВНИМАНИЕ

Важно понимать, что отправлять обычные SQL-запросы от клиента к серверу не следует; подобные действия открывают путь для злоумышленников, атакующих сервер и базу данных. Именно из-за этого в нашем приложении все SQL-запросы отображены ключевыми словами, которые превращаются в действующие запросы на серверной стороне. Таким образом мы получаем большую гибкость на стороне клиента, уберегая базу данных от атак.

Чтобы понять, как работает эта технологическая цепочка, мы пройдем по всему процессу создания запроса от JavaScript клиента к серверу и базе данных.

Аjax-запрос

Аjax-запрос осуществляется при необходимости связи с базой данных (и, соответственно, с сервером). К примеру, Аjax-запрос осуществляется, если нужно сохранить правку, внесенную в wiki. Правка состоит из четырех информационных частей:

Заголовка wiki-страницы: Общее соглашение об наименовании wiki-страниц использует так называемую схему CamelCase (двугорбого верблюда), которая заключается в соединении воедино слов, начинающихся с большой буквы. Например, главная страница нашего wiki именуется HomePage.

Имени автора правки: Пользователи вносящие правки, могут указать свое имя. Но, если они захотят, то могут сохранить анонимность (столь популярную во многих wiki-проектах).

Самого текста правки: Это потенциально объемный текст, вводимый пользователем. Содержимое отформатировано с помощью распространенной системы форматирования текста Textile.

Точного времени внесения правки: Оно генерируется на стороне клиента и (будем надеяться) используется в качестве уникального идентификатора. Поскольку время дается с точностью до миллисекунд, для нашего приложения этого будет вполне достаточно.

Теперь, используя все эти части данных, мы можем построить запрос, посылаемый к базе данных. Упрощенная версия кода, необходимого для сохранения правки показана в листинге 13.1.

Листинг 13.1. Код, необходимый для сохранения данных, полученных от клиента, в базе данных на стороне сервера

```

// Вставка правки в базу данных
sqlExec (
    // Оператор запроса, действующий код которого хранится на сервере
  
```

```

"insert",
[
  document.title, // Заголовок записи
  $("#author").val(), // Имя автора, предоставленное пользователем
  $("#text").val(), // Текст правки
  (new Date()).getTime() // Точное время правки
],
// После выполнения запроса – перезагрузка списка правок
reload
);

```

Вслед за функцией `sqlExec` происходит подготовка SQL-запроса (который затем превращается в строку CGI-запроса), построение URL для запроса, и создание Ajax-запроса. Для объединения всего этого в единое целое мы воспользуемся возможностями Ajax-библиотеки `jQuery`, которая может обрабатывать HTTP POST-запросы (POST-запрос необходим для отправки длинного текста правки). Окончательный вариант Ajax-запроса мы увидим в разделе «Обработка JSON-ответа».

Отправив запрос на сервер, нужно посмотреть, как запрос к базе данных обрабатывается на стороне сервера.

Код на стороне сервера

Серверная часть приложения имеет очень простую и легко воспроизводимую логику. Общая задача состоит в получении от клиента SQL-запроса, выполнении его в базе данных SQL и возвращении результатов в виде JSON-строки. Чтобы воплотить это приложение в жизнь, я решил усовершенствовать его логику и разработать дубликаты рабочих версий на всех популярных языках сценариев: Perl, PHP, Python и Ruby. Я предполагаю, что вам приходилось иметь дело по крайней мере с одним из этих языков; а если нет, то у вас будет прекрасная возможность ознакомиться с ними на нашем примере.

Обработка запроса

Клиентская часть только что инициировала связь с серверной частью приложения, требуя выполнения SQL-запроса к базе данных. Доступ к этому запросу и к имени базы данных, на которой он должен быть выполнен, можно получить через обращение к CGI-параметрам, переданным приложению. В предыдущем разделе мы видели, что сценарию на серверной стороне передаются два параметра: имя базы данных и текст SQL-запроса. Чтобы понять, как извлекаются параметры, в листинге 13.2 показано, как это делается в версии кода на стороне сервера, написанном на языке Ruby.

Листинг 13.2. Извлечение CGI-параметров, переданных серверной части приложения, написанной на Ruby

```

# Импорт CGI-библиотеки
require 'cgi'

# Создание нового CGI-объекта, который проведет разбор
# входящих CGI-параметров
cgi = CGI.new

# Захват параметров запроса
sql = cgi['sql']

# Получение от пользователя имени базы данных,

```

```
# и обеспечение отсутствия в них злонамеренно
# вставленных символов
d = cgi['db'].gsub(/[^a-zA-Z0-9_]/, "")
```

Теперь, используя извлеченные имя базы данных и запрос, нужно подключиться к базе данных, и тут возникает вопрос, каким типом базы данных воспользоваться. Я уже решил использовать базу данных на основе SQL (поскольку это уже стало распространенным стандартом при разработке веб-приложений). Для нашего приложения я решил воспользоваться базой данных SQL под названием SQLite.

SQLite является весьма перспективной реализацией базы данных SQL, обладающей невероятной легковесностью и быстротой работы. Ради простоты и скорости работы в ней пришлось пожертвовать такими понятиями, как пользователи, роли и полномочия доступа. Для нашего приложения она подходит как нельзя лучше. SQLite работает, запускаясь из одного файла на вашей системе, поэтому у вас может быть столько баз данных, сколько имеется файлов. Кроме быстроты работы, SQLite служит быстрым и простым способом установки базы данных для простых приложений или проведения тестирования. SQLite даст все, что нужно, избавляя от необходимости устанавливать большие базы данных (такие как MySQL, PostgreSQL или Oracle).

Все исследованные мной языки (Perl, PHP, Python и Ruby) тем или иным способом поддерживали SQLite:

- В Perl имеется модуль DBD::SQLite. Этот модуль особенно примечателен, поскольку разработчики решили полностью реализовать спецификацию SQLite внутри самого модуля, а это значит, что для доступа к базе данных и ее запуска не требуется никаких дополнительных загрузок.
- PHP 5 имеет встроенную поддержку SQLite. К сожалению этот язык поддерживает только SQLite 2 (который вполне подойдет для некоторых приложений), но чтобы получить полную совместимость между различными кодовыми основами, вместо этого нужно установить библиотеку PHP SQLite 3.
- И Python и Ruby имеют библиотеки SQLite, которые становятся доступными при официальной установке SQLite. У Python 2.5 есть поддержка SQLite, встроенная прямо в язык (но я не решился ее использовать из-за ее относительной новизны).

Я настоятельно рекомендую, чтобы в каком-нибудь из своих небольших проектов вы исследовали применение SQLite. Это будет отличным способом его подготовки и запуска, если вам не нужны издержки установки большой базы данных.

Способ подключения к базе данных SQLite во всех языках практически одинаков. Все они требуют для этого выполнить два шага: во-первых, включить в код библиотеку SQLite (предоставляющую универсальные функции подключения), и во-вторых, подключиться к базе данных и запомнить это подключение для дальнейшего использования.

В листинге 13.3 показано, как это делается в Ruby.

Листинг 13.3. Импорт внешней библиотеки SQLite и подключение к базе данных в Ruby-версии серверного кода

```
# Импорт внешней SQLite-библиотеки
require 'rubygems'
require_gem 'sqlite3-ruby'

# Далее в программе ...;

# 'd' нужно очистить, чтобы гарантировать отсутствие зловредных символов,
# предоставленных вместе с именем файла базы данных
```

```
d = cgi['db'].gsub(/[^a-zA-Z0-9_-]/, "")

# Подключение к базе данных SQLite, которая является простым файлом
# 'd' содержит имя базы данных — 'wiki'
db = SQLite3::Database.new('../..data/' + d + '.db')
```

Теперь, имея открытое подключение к базе данных SQLite, мы можем выполнить запрос, отправленный со стороны клиента, и получить результаты.

Выполнение и форматирование SQL

После того, как подключение к базе данных открыто, мы можем выполнить SQL-запрос. Конечная цель заключается в возможности поместить результаты запроса в форму, которая может быть легко преобразована в JSON-строку и возвращена клиенту. Наипростейшей удобной формой для SQL-результатов является массив хэшей, который выглядит подобно коду, представленному в листинге 13.4. Каждый хэш представляет соответствующую строку базы данных. Каждая пара хеша ключ-значение представляет имя столбца и его значение в строке.

Листинг 13.4. Пример JSON-структуры, возвращенной с сервера

```
[
  {
    title: "HomePage",
    author: "John",
    content: "Welcome to my wonderful wiki!",
    date: "20060324122514"
  },
  {
    title: "Test",
    author: "Anonymous",
    content: "Lorem ipsum dolem...;",
    date: "20060321101345"
  },
  . . .
]
```

Сложность помещения SQL-результатов в желаемую структуру варьируется в зависимости от выбранного вами языка. Но в большинстве случаев SQL-библиотека возвращает две вещи: массив имен столбцов, и массив массивов, содержащий все данные строки (см. листинг 13.5).

Листинг 13.5. Структура данных, возвращенная в результате выполнения SQL-запроса информации о правке, внесенной в Wiki, выраженная на языке Ruby

```
rows.columns = ["title", "author", "content", "date"]

rows = [
  ["HomePage", "John", "Welcome to my wonderful wiki!", "20060324122514"],
  ["Test", "Anonymous", "Lorem ipsum dolem...;", "20060321101345"],
  . . .
]
```


Процесс преобразования SQL-результатов, представленный в листинге 13.5, чтобы они больше походили на структуру данных, показанную в листинге 13.4, может быть более интересным. По существу, нам нужно осуществить последовательный перебор всех соответствующих строк, создать временный хэш, заполнить его всеми данными столбцов, а затем добавить новый хэш в глобальный массив. В листинге 13.6 показано, как это делается в Ruby.

Листинг 13.6. Как в Ruby выполняется SQL-оператор, и результаты его работы помещаются в окончательную структуру данных (названную `r`)

```
# Если в sql есть возвращаемые строки (например, выполнялся оператор SELECT)
db.query( sql ) do |rows|
  # Проход по всем возвращенным строкам
  rows.each do |row|
    # Создание временного хэша
    tmp = {}

    # Вывод столбцов массива в пары хэша ключ-значение
    for i in 0 .. rows.columns.length-1
      tmp[rows.columns[i]] = row[i]
    end

    # Добавление строки хэша к массиву найденных строк
    r.push tmp
  end
end
```

Теперь, после того как у нас есть удобная конечная структура данных, мы можем приступить к ее преобразованию в строку JSON. По своей сути, JSON является способом представления значений (строк и чисел), массивов значений и хэшей (пар ключ-значение), использующим JavaScript-совместимую объектную запись. Поскольку мы позаботились о гарантии отсутствия в структуре данных всего, кроме массивов, хэшей и строк, все это можно легко преобразовать в строку JSON-формата.

Все языки сценариев, используемые нами для этого приложения, имеют реализацию JSON-сериализации (которая воспринимает исходную структуру данных и конвертирует ее в JSON-строку), что нам, собственно, и нужно. Кроме того, поскольку вывод очень легок, как и большинстве случаев, связанных с данными формата JSON, его очень просто вывести на браузер. Кстати, реализации на разных языках в конечном счете очень похожи друг на друга, что значительно упрощает процесс перехода с одного языка на другой:

- Каждая реализация доступна в форме библиотеки или модуля.
- Каждая реализация способна транслировать исходные объекты языка (например, строки, массивы и хэши).
- Каждая реализация облегчает получение строки объекта формата JSON.

Но все же в Ruby реализация JSON-сериализации сделана особенно элегантно. Вот как выглядит пример преобразования объекта в JSON-строку (после загрузки библиотеки JSON) и возвращения ее клиенту:

```
# Преобразование объекта (r) в JSON-строку, и ее вывод
print r.to_json
```

Я настоятельно рекомендую посмотреть код серверной реализации на вашем любимом языке, и разобраться, как он обрабатывает SQL-запросы и проводит JSON-сериализацию. Я думаю, что вы будете приятно удивлены тому, насколько все это просто.

Обработка JSON-ответа

Теперь у нас есть ответ, полученный с сервера, который содержит отформатированную JSON-строку. Возвращаясь к листингу 13.2 теперь можно написать код, необходимый для обработки и выполнения этого JSON-кода. Чем хорош JSON, так это простотой выполнения и перемещения по его элементам. Здесь не нужен полнофункциональный парсер (как при работе с XML); все, что нужно вместо этого — функция `eval()` (которая выполняет код JavaScript, чем, собственно, и является JSON-строка). В листинге 13.7 показано, что библиотека `jQuery`, полностью берет все это на себя. Все, что для этого нужно сделать — это определить тип данных (названный `dataType`) как «`json`», и вы немедленно получите JSON-данные с сервера.

Листинг 13.7. Получение результатов с сервера и отправка структуры данных JavaScript функции обратного вызова

```
// Отправка запроса на сервер
$.ajax({
    // POST к API URL
    type: "POST",
    url: apiURL,

    // Сериализация массива данных
    data: $.param(p),

    // Ожидается возвращение данных в формате JSON
    dataType: "json",

    // Ожидание успешного завершения запроса
    // Если пользователь определил функцию обратного вызова,
    // отправить ей данные
    success: callback
});
```

Если вы заметили, последним совершаемым действием в листинге 13.7 было выполнение функции под названием `callback` (которая указывает на ту функцию обратного вызова, которая предоставлена функции `sqlExec`). Единственным аргументом, передаваемым этой функции является полная структура данных, так тщательно выстроенная на сервере и выставленная клиенту. Чтобы понять, как работает полный технологический процесс `sqlExec-Ajax-ответ`, посмотрите на показанную в листинге 13.8 упрощенную версию всего, что реально происходит в `wiki`. Логика состоит в том, что если для страницы есть правка, то должна быть показан ее самый последний вариант. А если правки не существует, форма должна быть отображена так, чтобы пользователь мог создать новую правку.

Листинг 13.8. Упрощенная форма кода, используемого на стороне клиента для извлечения правки с сервера и отображения ее на веб-сайте

```
// Запрос всех правок для текущей wiki-страницы
// После загрузки возвращение данных функции 'loaded'
sqlExec("select", [$s], loaded);

// Обработка SQL-результатов, возвращенных с сервера
function loaded(sql) {
    // Если для этой wiki правка существует
    if ( sql.length > 0 ) {
```

```

// Показать wiki-страницу
showContent();
// Представление правки с использованием textile
$("#content").html(textile(sql[0].content));
// Включение возможности редактирования содержимого правки
$("#textarea").val( sql[0].content );
// А если правки не существует, показать форму 'create me'
} else {
    // Отображение исходной формы для редактирования
    showForm();
}
}

```

Важной частью кода, показанного в листинге 13.8, является краткий фрагмент `textile(sql[0].content)`, который выхватывает wiki-правку из структуры данных и запускает для его обработки систему форматирования Textile. Много в HTML зависит от оформления содержимого, чем как раз и занимается Textile. Это очень простое средство, предоставляющее базовое, вполне понятное форматирование для любых случаев. Пример Textile-форматирования показан на рис. 13.3.

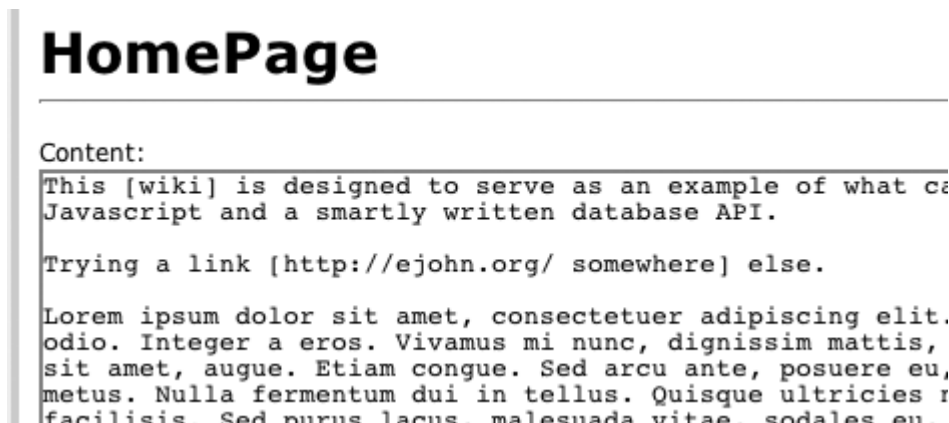


Рис. 13.3. Wiki-правка, использующая Textile-форматирование

Но одной из привлекательных сторон Textile является то, что кто-то уже выполнил всю трудную работу, и создал библиотеку Textile JavaScript, которая способна превращать отформатированный Textile текст в итоговый, презентабельный HTML. Более подробная информация о Textile может быть найдена на следующих веб-сайтах:

- Обзор Textile: <http://www.textism.com/tools/textile/>
- JavaScript-реализация Textile (использованная в этом проекте): <http://jrm.cc/extras/live-textile-preview.php>

Теперь, когда данные вернулись с сервера отформатированными и вставленными в документ, у вас есть все необходимое для полноценной wiki-системы. С этого момента я рекомендую вам установить свою собственную копию wiki-кода и посмотреть его в работе. Он подробно откомментирован, поэтому с ним можно будет легко справиться, не испытывая каких-либо стрессов.

Дополнительный учебный пример: JavaScript блог

Если вы интересуетесь тем, что еще может быть сделано с помощью этой связки клиент-сервер-SQLite, то я создал еще одну демонстрационную программу, которую вы можете совершенно свободно изучить. Это приложение представляет собой простой, персональный браузерный блог. Тем не менее он очень удобен; при посещении страницы вы получаете свой собственный персональный блог, который вы можете просматривать и вести. Кроме этого, в нем есть настоящий предварительный просмотр публикации и небольшая SQL-консоль, с

которой будет интересно познакомиться. Код вместе с демонстрационным блог-приложением находится на веб-сайте книги (<http://jspro.org/>). Копия экрана работающего блога показана на рис. 13.4.



Рис. 13.4. Копия экрана браузерного блога и SQL-консоли

Код приложения

Обучающий пример, представленный в этой главе намного сложнее, чем только что продемонстрированный. В этом разделе приводится подробный исходный код, необходимый для запуска нашего приложения (который имеет непосредственное отношение ко всему, что рассматривается в этой главе).

Теперь посмотрим на список файлов, необходимых для приведения в действие нашего wiki-приложения. Он представляет собой сборник кода клиентской и серверной сторон, рассмотренного в этой главе, дополненный различными библиотеками и файлами стилей, которые в ней напрямую не упоминались:

- *index.html*: Главная страница приложения, в которой собран воедино весь код клиентской стороны.
- *install.html*: Главный установочный файл, который запускается перед первым использованием приложения.
- *css/style.css*: Стилиевое CSS-оформление для клиентской стороны приложения.
- *js/wiki.js* (см. листинг 13.9): Основной код JavaScript, отвечающий за привязку событий и запуск SQL-запросов.
- *js/sql.js* (см. листинг 13.10): Код, отвечающий за связь с сервером, извлекающий JSON-данные, возвращаемые SQL-запросами.
- *js/textile.js*: Копия библиотеки JavaScript Textile (для преобразования текста в HTML): <http://jrm.cc/extras/live-textile-preview.php>.

- *js/jquery.js*: Копия текущей версии jQuery: <http://jquery.com/>.
- *api/*: Основной серверный код, отвечающий за преобразование результатов SQL-запроса в JSON и возвращения их клиенту. Этот каталог содержит все версии когда: на Perl, PHP, Python и Ruby. Я уже включил копию Ruby-версии кода в листинг 13.11.
- *data/wiki.db*: База данных SQLite, в которой хранится wiki.

Полный код файлов этой главы приведен в следующих листингах.

Основной код JavaScript

В листинге 13.9 показан *wiki.js*, основной код JavaScript, отвечающий за привязку событий и взаимодействие с пользователем.

Листинг 13.9. *js/wiki.js*

```
// Получение имени текущей страницы
var $s = window.location.search;
$s = $s.substr(1,$s.length);

// Определение, предоставлен ли номер правки, и если
// предоставлен, то запомнить его ID
var $r = false;

// Правки предоставляются в формате
// ?Title&RevisionID
var tmp = $s.split("&");
if ( tmp.length > 1 ) {
    $s = tmp[0];
    $r = tmp[1];
}

// Если страница не предоставлена, переход на начальную страницу
if (!$s) window.location = "?HomePage";

// Установка имени базы данных
var db = "wiki";

// Нужно дождаться окончания загрузки DOM
$(document).ready(function() {

    // Установка заголовка страницы
    document.title = $s;
    $("h1").html($s);

    // Загрузка всех wiki-правок
    reload();

    // Если произведен щелчок на ссылке
    // 'edit page' (редактировать страницу)
    $("#edit").click(showForm);
```

```

// Когда пользователь отправляет новую правку
$("#post form").submit(function() {
    // Получение имени автора
    var author = $("#author").val();

    // Получение текста
    var text = $("#text").val();

    // Отображение содержимого
    $("#content").html(textile(text));

    // Предоставление времени текущей правки
    // (которое поможет ее выделению)
    $r = (new Date()).getTime();

    // Вставка правки в базу данных
    sqlExec("insert", [$s,author,text,$r], reload);

    return false;
});

// Если пользователь щелкнул на ссылке 'cancel' (Отмена)
// внутри области редактирования
$("#cancel").click(showContent);
});

// Отображение текущей версии правки
function showContent() {
    // Отображение ссылки на редактирование
    $("#edit,#cancel").css("display","inline");

    // Скрытие область редактирования
    $("#post").hide();

    // Отображение содержимого
    $("#content").show();
    return false;
}

// Отображение формы редактирования текущего варианта правки
function showForm() {
    // Скрытие ссылки на редактирование
    $("#edit").hide();

    // Отображение области редактирования
    $("#post").show();
}

```

```

// Скрытие содержимого
$("#content").hide();
return false;
}

// Загрузка всех правок из базы данных
function reload(t) {
    // Запрос всех правок
    sqlExec("select", [$s], function(sql) {
        // Если для этой wiki-страницы существуют правки
        if ( sql.length > 0 ) {
            if ( !$r ) $r = sql[0].date;

            // Отображение wiki-страницы
            showContent();

            // Отображение всех правок
            $("#side ul").html('');

            // Проход по всем правкам
            for ( var i = 0; i < sql.length; i++ ) {

                // Если эта правка является текущей отображаемой правкой
                if ( sql[i].date == $r ) {

                    // Отображение правки
                    $("#content").html(textile(sql[i].content));

                    // Включение возможности редактирования
                    // содержимого правки
                    $("#textarea").val( sql[i].content );

                }

                // Получение реального объекта данных
                var d = new Date( parseInt(sql[i].date) );

                // Определение, была ли правка сделана в течение
                // последних суток, или нет
                if ( d.getTime() > (new Date()).getTime() - (3600 * 24000) )

                    // Если да, формирование приемлемого отображения
                    // времени создания в формате am/pm
                    d = d.getHours() >= 12 ?
                        (d.getHours() != 12 ? d.getHours() - 12 : 12 )
                        + " pm" :
                        d.getHours() + " am";
            }
        }
    });
}

```

```

// В противном случае отображение месяца и дня
// правки
else {
    var a = d.toUTCString().split(" ");
    d = a[2] + " " + d.getDate();
}

// Добавление правки в список правок
$("#side ul").append("<li class='" +
    ( $r == sql[i].date ? "cur" : "" )
    + "'><a href='?' + $s + ( i > 0 ? "&"
    + sql[i].date : "" )
    + "'>" + d + "</a> by " + sql[i].author + "</li>");
}

// В противном случае эта страница правке никогда не подвергалась
} else {
    // Отображение этого обстоятельства на панели правки
    $("#rev").html("<li>No Revisions.</li>");

    // Скрытие элементов управления редактированием
    $("#edit,#cancel").hide();

    // Отображение исходной формы редактирования
    showForm();
}
});
}

```

JavaScript SQL-библиотека

В листинге 13.10 показан файл `sql.js`, в котором содержится код, отвечающий за связь с сервером и извлечение JSON-данных, полученных на основе SQL-запросов.

Листинг 13.10. `js/sql.js`

```

// ЭТА ПЕРЕМЕННАЯ ТРЕБУЕТ ОБНОВЛЕНИЯ
// URL, по которому находится сценарий на стороне сервера
var apiURL = "api/ruby/";

// Некоторые заданные по умолчанию глобальные переменные
var sqlLoaded = function(){}

// Обработка больших SQL-отправлений
// Эта функция способна отправлять большие объемы данных
// (например, большие вставки — INSERT), но она может
// отправлять данные только на тот же сервер, с которым работает клиент
function sqlExec(q, p, callback) {

    // Загрузка всех параметров в структурированный массив

```



```

for ( var i = 0; i < p.length; i++ ) {
    p[i] = { name: "arg", value: p[i] };
}

// Включение имени базы данных
p.push({ name: "db", value: db });

// и названия SQL-запроса, который нужно выполнить
p.push({ name: "sql", value: q });

// Отправка запроса на сервер
$.ajax({
    // POST к API URL
    type: "POST",
    url: apiURL,

    // Сериализация массива данных
    data: $.param(p),

    // Ожидается возвращение данных в формате JSON
    dataType: "json",

    // Ожидание успешного завершения запроса
    // Если пользователь определил функцию обратного вызова,
    // отправить ей данные
    success: callback
});
}

```

Ruby-код на стороне сервера

Следующий код, показанный в листинге 13.11, представляет серверную часть Ajax-кода wiki-приложения. Этот код целиком написан на языке программирования Ruby. Примеры такого же кода, но написанного на PHP, Perl или Python, можно найти на веб-сайте <http://jspro.org/>.

Листинг 13.11. Серверная часть кода wiki-приложения, написанная на языке Ruby

```

#!/usr/bin/env ruby

# Импорт всех внешних библиотек
require 'cgi'
require 'rubygems'
require_gem 'sqlite3-ruby'
require 'json/objects'

# Отображение заголовка Javascript
print "Content-type: text/javascript\n\n"

# Инициализация переменных приложения
err = ""

```

```

r = []
cgi = CGI.new

# Перехват параметров, переданных пользователем
call = cgi['callback']
sql = cgi['sql']

# Получение от пользователя имени базы данных,
# и обеспечение отсутствия в них злонамеренно
# вставленных символов
d = cgi['db'].gsub(/[^a-zA-Z0-9_-]/, "")

# Если имя базы данных не предоставлено, использовать имя 'test'
if d == '' then
  d = "test"
end

# Получение списка параметров, помещаемых в
# SQL-запрос
args = cgi.params['arg']

# Воспринимаются только два различных SQL-запроса

# Вставка новой wiki-правки в базу данных
if sql == "insert" then
  sql = "INSERT INTO wiki VALUES (?, ?, ?, ?);"

# Получение всех правок wiki-записи
elsif sql == "select" then
  sql = "SELECT * FROM wiki WHERE title=? ORDER BY date DESC;"

# В противном случае, отказать в запросе
else
  sql = ""
end

# Если запрос был предоставлен
if sql != '' then

  # Проход по всем предоставленным параметрам
  for i in 0 .. args.length-1
    # Замена всех одиночных кавычек на '' (что эквивалентно их
    # деактивации в SQLite), и деактивация всех знаков ?
    args[i] = args[i].gsub(/'/, "''").gsub(/\?/, "\\?")

  # Затем проход по SQL-запросу и замена первого соответствующего
  # знака вопроса данными параметрами
  sql = sql.sub(/([\^\])\?/, "\\1'" + args[i] + "'")

```

```

end

# После того, как все сделано, снятие деактивации со знака вопроса
sql = sql.gsub(/\\?\?/, "?")

# Обеспечение захвата всех выдаваемых ошибок базы данных
begin
  # Подключение к базе данных SQLite, представляющей
  # собой простой файл
  db = SQLite3::Database.new('.././data/' + d + '.db')

  # Если в sql есть возвращаемые строки
  # (например, выполнялся оператор SELECT)
  db.query( sql ) do |rows|
    # Проход по всем возвращенным строкам
    rows.each do |row|
      # Создание временного хэша
      tmp = {}

      # Вывод столбцов массива
      # в пары хэша ключ-значение
      for i in 0 .. rows.columns.length-1
        tmp[rows.columns[i]] = row[i]
      end

      # Добавление строки хэша к массиву найденных строк
      r.push tmp
    end
  end

rescue Exception => e
  # Если произошла ошибка, запоминание сообщения
  # для использования в будущем
  err = e
end

else
  # Если SQL-запрос предоставлен не был, отображение ошибки
  err = "Запрос не предоставлен."
end

# Если произошла ошибка, возвращение хэша, содержащего
# ключ error и значение, содержащее сообщение об ошибке
if err != '' then
  r = { "error" => err }
end

# Преобразование возвращенного объекта в JSON-строку
jout = r.to_json

```

```
# Если предоставлена функция обратного вызова
if call != '' then
    # Помещение возвращаемого объекта в строку обратного вызова
    print call + "(" + jout + ")"
else
    # В противном случае простой вывод JSON-строки
    print jout
end
```

Вывод

Я надеюсь, что при изучении нашего приложения вам удалось усвоить ряд полезных вещей. Во-первых, что JSON является мощной и жизнеспособной альтернативой использованию XML в веб-приложениях. Во-вторых, сохраняя насколько это возможно простоту кода на серверной стороне, вы можете дать программе пользовательского интерфейса больше возможностей управления пользовательскими данными (и это надо делать умеренно, сохраняя в скрытом состоянии наиболее важную часть логики действия). И наконец, что все современные языки сценариев, используемые на стороне сервера могут вести себя весьма похожим образом (поскольку обладают многими сходными чертами) и с их помощью сравнительно легко осуществлять загрузку и извлечение информации (поэтому я и написал идентичные версии кода серверной стороны на четырех наиболее популярных языках сценариев).

Весь код основного приложения может быть найден в разделе «Код приложения» этой главы. Полная установочная версия приложения может быть найдена на веб-сайте этой книги <http://jspro.org/>, или на веб-сайте <http://www.apress.com/>, дополненном подробной инструкцией по установке. Работоспособная демонстрационная версия может быть найдена по адресу: <http://jspro.org/demo/wiki/>.

Этот веб-сайт включает полноценные инструкции по использованию кода. Кроме этого, там же организован форум для обсуждения любых проблем, с которыми вы можете столкнуться при попытке создать свою собственную установку.

Глава 14 В каком направлении движется JavaScript?

За последнюю пару лет с многих направлений появилось огромное количество разработок на языке JavaScript. Mozilla Foundation значительно продвинулась в повышении качества языка JavaScript, согласовав его с ECMAScript 4 (языком, на котором основан JavaScript). С другого направления действовала рабочая группа WHAT-WG, объединившая усилия производителей веб-браузеров, желающих разрабатывать новые технологии, позволяющие создавать и развертывать приложения всемирной паутины, которая разработала спецификацию для следующего поколения приложений, основанных на использовании браузеров. И наконец, авторы библиотек и корпорации работали над укреплением технологий потоковых браузерных приложений в средство под названием *Comet*. Все эти новинки представляют будущее языка JavaScript и браузерных приложений.

В этой главе мы собираемся рассмотреть усовершенствования, появившиеся в JavaScript 1.6 и 1.7, которые приближают выпуск JavaScript 2.0. Затем мы рассмотрим одно из усовершенствований, созданных в рамках спецификации Web Applications 1.0: возможность рисования с использованием JavaScript. В заключение мы рассмотрим накоротке замысел, положенный в основу Comet и потоковых веб-приложений.

JavaScript 1.6 и 1.7

С начала текущего десятилетия язык JavaScript медленно продвигался вперед, добавляя в свой арсенал функциональные усовершенствования. Хотя многие современные браузеры поддерживают JavaScript 1.5 (или его эквивалент), они крайне слабо продвигают этот язык вперед.

Брендан Эйч (Brendan Eich) и другие специалисты Mozilla Foundation, усердно работали над продвижением языка во взаимодействии с ECMAScript 4. Более подробная информация о работе Mozilla могут быть найдены в следующих источниках:

- *Работа Mozilla над JavaScript*: <http://www.mozilla.org/js/language/>
- *Предложения Mozilla по JavaScript 2.0*: <http://www.mozilla.org/js/language/js20/>
- *Предложения Mozilla по ECMAScript 4*: <http://www.mozilla.org/js/language/es4/>

Не дожидаясь завершения JavaScript 2.0, Mozilla посягнула на выпуск JavaScript версий 1.6 и 1.7, в которые включен ряд возможностей, намеченных для конечной, пересмотренной версии языка. Многие добавленные возможности носят весьма существенный характер, и в этом разделе мы проведем их краткий обзор.

JavaScript 1.6

Первый выпуск обновленного языка JavaScript появился в виде JavaScript 1.6. Он был выпущен вместе с браузером Firefox 1.5, созданным Mozilla Foundation. Краткий обзор изменений, внесенных в JavaScript 1.6 можно найти на веб-сайте Mozilla: http://developer.mozilla.org/en/docs/New_in_JavaScript_1.6.

В этом выпуске появились две важные свойства: E4X (ECMAScript для XML) и набор дополнительных функций для работы с массивами. На данный момент ни одна из этих функций не реализована ни в одном другом браузере, но вполне возможно, что Opera и Safari будут следующими, кто окажется запрыгнет на борт этого судна. Я покажу вам те преимущества которые имеются у каждого из этих свойств.

ECMAScript для XML (E4X)

E4X добавила к языку JavaScript набор новых синтаксических элементов, дающих возможности записывать встроенный XML прямо внутри кода JavaScript. Результат получился довольно интересный, но все же весьма сложный. Более подробную информацию о E4X можно найти в его спецификации на веб-сайте Mozilla:

- *Спецификация ECMAScript для XML*: <http://www.ecma-international.org/publications/standards/Ecma-357.htm>

- *Краткий обзор E4X*: <http://developer.mozilla.org/presentations/xttech2005/e4x/>

В общем, спецификация позволяет вам использовать при написании кода JavaScript-подобный синтаксис, результаты которого выражаются в XML DOM. Например, если написать `writing var img = + <hr/>`, то после элемента изображения появится горизонтальная линия, а получившийся DOM-элемент будет сохранен в переменной, которой позже можно будет воспользоваться. Более сложный пример приведен в листинге 14.1. Получившийся XML-документ показан в листинге 14.2.

Листинг 14.1. Построение HTML-документа с использованием E4X, взятое из презентации Брендана Эйча

```
<script type="text/javascript;e4x=1">
  // Создание HTML-элемента и сохранение его в переменной
  var html = <html/>;

  // Присвоение содержимому элемента title текстовой строки
  // E4X автоматически создает все пропущенные элементы и
  // берет на себя создание соответствующих текстовых узлов
  html.head.title = "Заголовок моей страницы";

  // Установка свойства цвета фона для элемента body,
  // который создается автоматически
  html.body.@bgcolor = "#e4e4e4";

  // И некоторых свойств к элементу form внутри элемента body
  html.body.form.@name = "myform";
  html.body.form.@action = "someurl.cgi";
  html.body.form.@method = "post";
  html.body.form.@onclick = "return somejs()";

  // Создание пустого элемента input с определенным именем
  html.body.form.input[0] = "";
  html.body.form.input[0].@name = "test";
</script>
```

Листинг 14.2. HTML-документ, сгенерированный вызовом E4X-кода из листинга 14.1

```
<html>
  <head>
    <title>Заголовок моей страницы</title>
  </head>
  <body bgcolor="#e4e4e4">
    <form name="myform" action="someurl.jss"
      method="post" onclick="return somejs();">
      <input name="test"></input>
    </form>
  </body>
</html>
```

Хотя синтаксис для E4X довольно сильно отстывает от нормального стиля JavaScript — что может быть достаточным, чтобы отпугнуть новичков — результат мог бы принести существенную пользу, позволяя сократить количество повторяющихся DOM-операций.

Дополнительные возможности по работе с массивами

Самые важные новые свойства, добавленные в JavaScript 1.6, относятся к работе с массивами. Теперь в версии 1.6 массивы приобрели несколько дополнительных методов, которые могут использоваться в обычных действиях:

- Два действия, `indexOf()` и `lastIndexOf()`, похожи на одноименные методы, существующие для строковых объектов. Они дают возможность найти позицию объекта внутри массива, возвращая соответствующий индекс, или `-1`, если объект в массиве отсутствует.
- Три новых метода, `forEach()`, `some()` и `map()`, предназначены для упрощения общепринятых итераций, и позволяют выполнять функцию над содержимым массива, над каждым его объектом.
- Новые функции `filter()` и `map()` позволяют проводить встроенные преобразования массива, аналогичные операциям `map` и `grep`, существующим в других языках (например, в Perl).

Примеры использования всех новых функций JavaScript 1.6 по работе с массивами показаны в листинге 14.3.

Листинг 14.3. Примеры использования новых функций JavaScript 1.6, работающих с массивами

```
// Простой массив чисел
var tmp = [ 1, 2, 3, 4, 5, 3 ];

// indexOf( Объект )
// Определяет индекс объекта внутри массива объектов
tmp.indexOf( 3 ) == 2
tmp.indexOf( 8 ) == -1

// lastIndexOf( Объект )
// Определяет последний объект внутри массива объектов
tmp.lastIndexOf( 3 ) == 5

// forEach( Функция )
// Вызывает функцию для каждого имеющегося в массиве объекта.
// Функции передаются три аргумента: объект, его индекс и ссылка на массив
tmp.forEach( alert );

// every( Функция )
// Вызывает функцию для каждого имеющегося в массиве объекта, если для
// каждого из них будет возвращено значение true, возвращает true
tmp.every(function(num) {
    return num < 6;
}) // true

// some( Функция )
```

```

// Вызывает функцию для каждого имеющегося в массиве объекта.
// Если для какого-нибудь объекта будет возвращено значение true,
// возвращает true
tmp.some(function(num) {
    return num > 6;
}) // false

// filter( Функция )
// Урезает массив до тех элементов, которые соответствуют определенным
// критериям. Объект сохраняется, если функция возвращает 'true'.
tmp.filter(function(num) {
    return num > 3;
}) // [ 4, 5 ]

// map( Функция )
// Преобразует массив объектов в другой набор объектов. Результат,
// возвращенный определенной функцией, преобразует объект к его новому
// значению
tmp.map(function(num) {
    return num + 2;
}) // [ 3, 4, 5, 6, 7, 5 ]

```

Кроме этих простых примеров, эти новые методы придают работе с массивами в JavaScript столь необходимую скорость и функциональность. Я конечно с нетерпением жду того дня, когда эти методы получат поддержку на всех браузерах.

JavaScript 1.7

Этот новый выпуск языка JavaScript привнес в него много новых функциональных возможностей, добавив ряд свойств, которые приблизили его к другим полнофункциональным языкам. Кроме всего прочего, новый выпуск JavaScript 1.7, обладает еще большими усовершенствованиями, чем выпущенное ранее обновление JavaScript 1.6, в него добавлен ряд новых особенностей, изменяющих сам способ его работы. Подробности некоторых новых свойств, доступных в JavaScript 1.7, приведены на веб-сайте http://developer.mozilla.org/en/docs/New_in_JavaScript_1.7.

Это обновление языка JavaScript было выпущено вместе с новой версией браузера Mozilla Firefox 2.0. В этот браузер включена полная реализация всего, что будет рассмотрено в этом разделе, и только он обладает столь значительными, современными обновлениями языка JavaScript.

Включения в массив

Одно очень интересное добавление позволяет весьма стильно и ловко писать код, относящийся к генерации массива. Раньше, чтобы заполнить массив списком элементов, нужно было последовательно перебрать набор и поместить его элементы в конечный массив. Теперь вместо этого можно воспользоваться включением в массив, и сделать все одним простым приемом, который поясняется на примере, показанном, в листинге 14.4.

Листинг 14.4. Включения в массив в JavaScript 1.7

```

<script type="application/javascript;version=1.7">
    // Старый способ помещения ряда номеров в массив
    var array = [];

```



```

for ( var i = 0; i < 10; i++ ) {
    array.push( i );
}

// Новый способ
var array = [ i for ( i = 0; i < 10; i++ ) ];

// старый способ помещения ключей объектов в массив
var array = []
for ( var key in obj ) {
    array.push( key );
}

// Новый способ
var array = [ key for ( key in obj ) ];
</script>

```

Эта специфическая особенность языка некоторое время присутствовала в других языках (таких как Python), и приятно видеть, что она добралась и до JavaScript.

Управление областью видимости переменных (Let Scoping)

Let Scoping — фантастическое нововведение, которое, скорее всего, получит наиболее широкое применение и признание. До сих пор в JavaScript не было никакой области видимости на уровне блоков (что уже рассматривалось в главе 2). С добавлением новых let-оператора, выражения и определения теперь появилась возможность определения различных областей видимости на нескольких различных уровнях. В листинге 14.5 показано несколько примеров того, что можно сделать с помощью let scoping.

Листинг 14.5. Примеры Let Scoping в JavaScript 1.7

```

<script type="application/javascript;version=1.7">
// Оператор let
var test = 10;
let( test = 20 ) {
    alert( test ); // выводится 20
}
alert( test ); // выводится 10

// Выражение let
var test = 10;
alert( let( test = 20 ) test ); // выводится 20
alert( test ); // выводится 10

// Определение let
var test = 10;
if ( test == 10 ) {
    let newTest = 20;
    test += newTest;
}
alert( test ); // выводится 30

```

```

alert( newText ); // сбой, переменная newText за пределами
                  // оператора не определена

// Использование let в блоке for
for ( let i = 0; i < 10; i++ ) {
  alert( i );
}
alert( i ); // сбой, переменная i за пределами оператора for
           // не определена
</script>

```

Используя это простое дополнение вы можете сделать свой код понятнее, работать более эффективно и избежать ряда общих конфликтных ситуаций, связанных с пространством имен (большая часть всего этого будет работать до введения классов и пространств имен в JavaScript 2.0).

Деструктуризация

Заключительной серьезным понятием, введенным в JavaScript 1.7, является деструктуризация (destructuring). Это понятие позаимствовано из функциональных языков программирования (в частности, в Lisp), и позволяет иметь сложные структуры данных слева от операнда, заполненные определенными значениями.

Более подробная информация о деструктуризации в ECMAScript 4 размещена на веб-сайте Mozilla: http://developer.mozilla.org/es4/proposals/destructuring_assignment.html.

Хотя понятие деструктуризации не отличается простотой, но на то чтобы с ним разобраться безусловно стоит потратить некоторое время. В листинге 14.6 показано несколько примеров работы деструктуризации в JavaScript 1.7.

Листинг 14.6. Примеры деструктуризации в 1.7

```

<script type="application/javascript;version=1.7">
  // Пример использования деструктуризации для обмена значений
  // двух переменных
  [ b, a ] = [ a, b ]

  // Простая функция, возвращающая массив строк
  function test() {
    return [ "Джон", "октябрь" ];
  }

  // Мы можем деструктурировать возвращенные данные в две новые
  // переменные - name и month
  var [ name, month ] = test();

  // Пример деструктуризации с использованием объекта
  var { name: myName } = { name: "Джон" };
  // Теперь myName == "Джон"

  // Простая структура данных
  var users = [
    { name: "Джон", month: "октябрь" },

```

```

    { name: "Боб", month: "декабрь" },
    { name: "Джейн", month: "май" }
  ];

  // Деструктуризация внутри цикла
  for ( let { name: name, month: month } in users ) {
    // Вывод всех уведомлений для Джона, Боба и Джейн
    alert( name + " месяц рождения " + month );
  }
</script>

```

В целом язык JavaScript развивается в нескольких весьма позитивных направлениях, как правило впитывая в себя полезные качества других языков (среди которых Python и Lisp). Но большинство появляющихся в нем полезных усовершенствований зависят от тех усилий по их реализации, которые различные поставщики браузеров вкладывают в язык. Наряду с тем, что Mozilla Foundation проявляет настойчивость в реализации новых возможностей, в других браузерах они развиваются довольно слабо. Хотя до начала использования JavaScript 1.6 или 1.7 в кроссбраузерах веб-приложениях должно еще пройти некоторое время, у вас есть возможность приступить к их использованию уже сейчас, разрабатывая расширения для браузеров на движке Mozilla (по крайней мере до тех пор, пока не появится более распространенная реализация языка).

Web Applications 1.0

Вторая спецификация, продвигающая вперед JavaScript-разработку — это Web Applications 1.0, которая принадлежит рабочей группе WHAT-WG (Web Hypertext Application Technology Working Group). Эта спецификация распространяется на несколько различных направлений, добавляя ряд нововведений к HTML, DOM и JavaScript в целом. Многие считают, что работа над этой спецификацией должна превратиться в HTML 5. К счастью, в отличие от новых версий JavaScript, реализации этой спецификации (или их части) заполучить намного проще.

Хотя вся спецификация достаточно обширна, я все же настоятельно рекомендую вам ее прочитать и посмотреть на новые технологии, которые уже на подходе. В этом разделе я обращаю ваше внимание только на одну конкретную особенность этой новой спецификации: на элемент <canvas>. Этот новый элемент дает возможность программным путем создавать двумерные изображения, используя JavaScript. Внедрение этой технологии идет очень интенсивно, облегчая ее изучение и тестирование. Дополнительная информация по Web Applications 1.0 и элементу <canvas> может быть найдена в следующих источниках:

- Полная спецификация Web Applications 1.0: <http://whatwg.org/specs/web-apps/current-work/>
- Подраздел спецификации, относящийся конкретно к элементу <canvas>: <http://whatwg.org/specs/web-apps/current-work/#the-2d>
- Ряд примеров использования нового элемента <canvas>: http://developer.mozilla.org/en/docs/Canvas_tutorial:Basic_animations

Элемент <canvas> удовлетворяет запросы многих разработчиков веб-приложений, позволяя им вращать изображения, рисовать линии и создавать графические формы. Только одно это нововведение может добавить веб-приложению совершенно новый уровень интерактивности.

Полезный эффект от возможности рисовать произвольные формы подстегнула команды, поддерживающие браузеры, к быстрому включению Canvas API в последние выпуски их продуктов. На данный момент уже все современные браузеры, за исключением Internet Explorer, поддерживают Canvas API, но Google нашел подход и к этому браузеру, и реализовал полноценный Canvas API в IE, используя исконно присущую ему поддержку VML. Более подробная информация о ExplorerCanvas, разработанной Google реализации Canvas в Internet Explorer находится по адресу: <http://code.google.com/p/explorercanvas/>.

А теперь я собираюсь провести углубленное рассмотрение двух примеров, предоставленных в основном руководстве Mozilla по <canvas>.

Создание часов

Первый пример заключается в создании простых часов. Мы собираемся использовать Canvas 2D API для рисования всех элементов часов; в этот процесс не будут использоваться какие-нибудь изображения или посторонние HTML-элементы. На рис. 14.1 показан пример часов, которые мы собираемся нарисовать.



Рис. 14.1. Анимированные часы, нарисованные с помощью Canvas API

Элемент <canvas> работает подобно настоящему холсту художника. Вы можете нарисовать единственный статический кадр анимированных часов, но чтобы нарисовать новый кадр, нужно полностью очистить свой холст и нарисовать все заново. Если у вас есть опыт работы с OpenGL API, то вы почувствуете себя как дома и в среде Canvas API.

При работе с Canvas API, вам понадобится часто вращать изображение, но это может привести к большой путанице, поскольку вы можете не знать, на какой угол вы в настоящий момент указываете, или где в данный момент находится ваша «кисть». По этой причине работа Canvas API во многом похожа на стек: сначала вы сохраняете старую позицию и вращаете холст, вносите какие-то изменения во внешний облик текущего изображения, а затем возвращаетесь назад к первоначальному изображению и продолжаете рисовать.

В листинге 14.7 приводится код, необходимый для создания анимированных часов с использованием Canvas; обратите внимание, что для облегчения рисования в этом коде широко используется стек-система Canvas.

Листинг 14.7. Рисование анимированных часов с использованием Canvas API

```
<html>
<head>
  <title>Демонстрация часов Canvas </title>
</script>
// Ожидание загрузки в браузер
window.onload = function() {
  // Рисование часов
  clock();

  // и перерисовка часов по прошествии каждой секунды
  setInterval(clock, 1000);
};

function clock() {
  // Получение текущей даты и времени
  var now = new Date();
  var sec = now.getSeconds();
```

```

var min = now.getMinutes();
var hr = now.getHours();
hr = hr >= 12 ? hr - 12 : hr;

// Получение контекста элемента <canvas>
var ctx = document.getElementById('canvas').getContext('2d');

ctx.save();
    // Инициализация холста для рисования
    ctx.clearRect(0,0,150,150);

    // Когда мы рисуем в 0,0, мы фактически рисуем в 75,75
    ctx.translate(75,75);

    // При рисовании линии 100px, фактически рисуется линия в 40px
    ctx.scale(0.4,0.4);

    // Начало вращения курсора с 12:00
    ctx.rotate(-Math.PI/2);

    // Инициализация свойств рисунка
    ctx.strokeStyle = "black";
    ctx.fillStyle = "black";
    ctx.lineWidth = 8;
    ctx.lineCap = "round";

    // Часовые метки
    ctx.save();
        ctx.beginPath();
            // Для каждого часа
            for ( var i = 0; i < 12; i++ ) {
                // Вращение холста на 1/12 пути
                // (помните: длина окружности = 2 * PI)
                ctx.rotate(Math.PI/6);

                // Перемещение курсора почти к краю холста
                ctx.moveTo(100,0);

                // и рисование короткой черточки (20px)
                ctx.lineTo(120,0);
            }
        ctx.stroke();
    ctx.restore();

    // Минутные метки
    ctx.save();
        // Эти черточки будут меньше часовых
        ctx.lineWidth = 5;

```

```

ctx.beginPath();
    // Для каждой минуты
    for ( var i = 0; i < 60; i++ ) {
        // кроме тех, что совпадают с часами
        if ( i % 5 != 0 ) {
            // Перемещение курсора чуть дальше
            ctx.moveTo(117,0);

            // И рисование короткой линии (3px)
            ctx.lineTo(120,0);
        }

        // Вращение холста 1/60 пути по кругу
        ctx.rotate(Math.PI/30);
    }
    ctx.stroke();
ctx.restore();

// Рисование часовой стрелки
ctx.save();
    // Вращение холста на текущую позицию
    ctx.rotate( (Math.PI/6) * hr + (Math.PI/360) * min
                + (Math.PI/21600) * sec )

    // Эта линия должна быть шире
    ctx.lineWidth = 14;

    ctx.beginPath();
        // Начало рисования с выходом за пределы центра (чтобы
        // было похоже на часовую стрелку)
        ctx.moveTo(-20,0);

        // и рисование почти до часовых черточек
        ctx.lineTo(80,0);
    ctx.stroke();
ctx.restore();

// Рисование минутной стрелки
ctx.save();
    // Вращение холста на текущую минутную позицию
    ctx.rotate( (Math.PI/30) * min + (Math.PI/1800) * sec )

    // Эта линия будет тоньше, чем часовая стрелка
    ctx.lineWidth = 10;

    ctx.beginPath();
        // Но она также и длиннее, поэтому ее нужно установить чуть

```

```

        // назад
        ctx.moveTo(-28,0);

        // и нарисовать чуть длиннее
        ctx.lineTo(112,0);
        ctx.stroke();
    ctx.restore();

    // Рисование секундной стрелки
    ctx.save();
        // Вращение холста на текущую секундную позицию
        ctx.rotate(sec * Math.PI/30);

        // Эта линия должна быть красноватой
        ctx.strokeStyle = "#D40000";
        ctx.fillStyle = "#D40000";

        // и более тонкой, чем другие стрелки
        ctx.lineWidth = 6;

        ctx.beginPath();
            // А также больше выступать назад
            ctx.moveTo(-30,0);

            // но быть короче
            ctx.lineTo(83,0);
            ctx.stroke();
        ctx.restore();

    // Внешняя синяя окружность
    ctx.save();
        // Обрамление будет широким
        ctx.lineWidth = 14;

        // и синеватым
        ctx.strokeStyle = '#325FA2';

        ctx.beginPath();
            // Рисование полной окружности, отступающей от центра
            // на 142px
            ctx.arc(0,0,142,0,Math.PI*2,true);
            ctx.stroke();
        ctx.restore();

    ctx.restore();
}
</script>
</head>

```

```
<body>
  <canvas id="canvas" height="150" width="150"></canvas>
</body>
</html>
```

Как только вы проработаете все подробности и математические особенности (которые различаются в зависимости от сложности объекта, который вы пытаетесь нарисовать), Canvas 2D API станет очень полезным инструментом.

Простая модель планет

В нашем втором примере мы собираемся рассмотреть вращение изображений в противоположность обычному рисованию форм. Начнем с трех базовых изображений: одного для Солнца, второго для Земли и третьего для Луны. Затем создадим простую модель (которая выглядит хорошо, но точностью не отличается), показывающую вращение Земли вокруг Солнца, и вращение Луны вокруг Земли. Кроме этого будет примерно показано, какая сторона Земли будет темной во время ее вращения вокруг Солнца. На рис. 14.2 показан пример, как будет выглядеть конечный результат действующей модели.

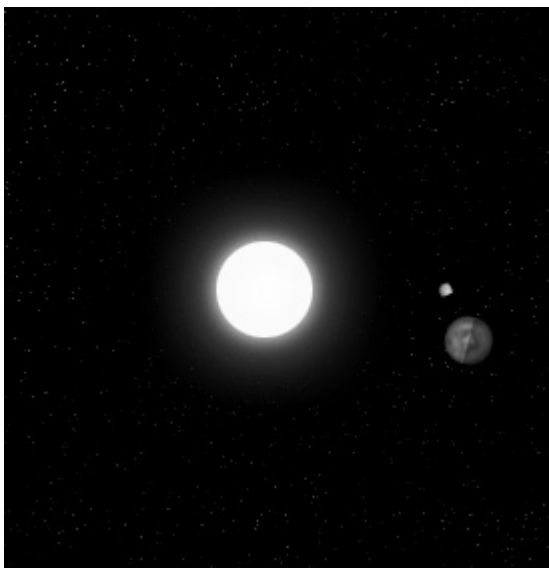


Рис. 14.2. Вращение Земли вокруг Солнца, и Луны вокруг Земли в простой планетной модели, выполненной с помощью Canvas

В коде этого примера, конечно, найдется много похожего на код предыдущего примера (а именно, сохранение и восстановление позиций холста), но важно отметить, как именно происходит обработка рисунка и вращение отдельных изображений. В листинге 14.8 показан полный код для планетной модели, выполненной с помощью Canvas.

Листинг 14.8. Модель вращения Земли вокруг Солнца, созданная с помощью Canvas 2D API

```
<html>
<head>
<title>Демонстрация части солнечной системы с помощью Canvas</title>
<script>
// Инициализация списка используемых изображений
var imgs = { sun: null, moon: null, earth: null };

// Ожидание полной загрузки окна
```



```

window.onload = function() {
    // Загрузка всех изображений из документа
    for ( var i in imgs )
        imgs[i] = document.getElementById(i);

    // Запуск рисования 10 раз в секунду
    setInterval( draw, 100 );
};

function draw() {
    // Получение необходимых интервалов времени
    var time = new Date();
    var s = ( (2 * Math.PI) / 6) * time.getSeconds();
    var m = ( (2 * Math.PI) / 6000 ) * time.getMilliseconds();

    // Получение контекста элемента <canvas>
    var ctx = document.getElementById('canvas').getContext('2d');

    // Очистка холста
    ctx.clearRect(0,0,300,300);

    // Новые элементы всегда рисуются под старыми (используется для тени)
    // Дополнительная информация:
    // http://developer.mozilla.org/en/docs/Canvas_tutorial:Compositing
    ctx.globalCompositeOperation = 'destination-over';

    ctx.save();
        // Рисование в 0,0 = рисованию в 150,150
        ctx.translate(150,150);

        // Вращение холста к позиции Земли
        ctx.rotate( (s + m) / 10 );

        // Перемещение на 105 пикселей
        ctx.translate(105,0);

        // Заполнение для тени (которая будет наплывать,
        // и мы сможем видеть сквозь нее)
        ctx.fillStyle = 'rgba(0,0,0,0.4)';
        ctx.strokeStyle = 'rgba(0,153,255,0.4)';

        // Рисование прямоугольника тени (не совсем
        // безупречного, но близкого к нужному)
        ctx.fillRect(0,-12,50,24);

        // Рисование Земли
        ctx.drawImage(imgs.earth,-12,-12);

    ctx.save();

```

```

    // Вращение холста, относительно вращения Земли
    ctx.rotate( s + m );

    // Вращение Луны 'по орбите'
    ctx.translate(0,28.5);

    // Рисование изображения Луны
    ctx.drawImage(imgs.moon,-3.5,-3.5);
ctx.restore();

ctx.restore();

// Рисование орбиты Земли
ctx.beginPath();
    ctx.arc(150,150,105,0,Math.PI*2,false);
ctx.stroke();

// Рисование неподвижного Солнца
ctx.drawImage(imgs.sun,0,0);
}
</script>
</head>
<body style="background:#000;">
    <canvas id="canvas" height="300" width="300"></canvas>
    <!-- Предварительная загрузка исходных изображений -->
    <div style="display:none;">
        
        
        
    </div>
</body>
</html>

```

Элемент `<canvas>` и соответствующий API в последнее время нашли широкое применение; они используются как в инструментальной панели Apple, так и в области графических элементов управления Opera. Это одна из частей «будущего» JavaScript которая доступна во всем своем практическом воплощении уже сейчас, и вы должны всерьез рассматривать ее использование в ваших приложениях.

Comet

Последняя новая концепция, которую мы собираемся рассмотреть, недавно подверглась усовершенствованию и находится в процессе превращения в новый стандарт. Хотя концепция Ajax является достаточно простой и понятной (наличие единственного асинхронного подключения), она не рассчитана на формирование пакета с каким-нибудь потоковым содержимым. Концепцию, способную обеспечить поток обновлений, приходящих в ваше веб-приложение, сейчас часто называют Comet (название выдумал Алекс Рассел (Alex Russell) из Dojo). Возможность потокового обновления внутри веб-приложения предоставляет вам совершенно новую степень свободы, позволяя создавать свертотзывчивые приложения (например, чаты).

Аналогично Ajax, Comet, по существу, тоже не содержит новой технологии. Но многие разработчики взялись за усовершенствование общих потоковых концепций, ориентированных на веб-приложения в конечный

стандарт, называемый Cometd. Дополнительная информация о планируемом стандарте Cometd и сама концепция Comet могут быть найдены в следующих источниках:

- Первоначальная статья Алекса Рассела (Alex Russell), уточняющая концепцию, лежащую в основе Comet: <http://alex.dojotoolkit.org/?p=545>
- Определение спецификации Comet: <http://cometd.com/>

Comet улучшает текущие Ajax-приложения за счет наличия долговременных подключений, которые обеспечивают непрерывный поток новой информации с центрального сервера. Центральному серверу отводится роль равномерного распределения подключений на соответствующие каналы. На рис. 14.3 показан пример, как работает Comet по сравнению с традиционным Ajax-приложением.

Организация Dojo Foundation несет ответственность за основной объем работы, ведущей к стандартизации Comet и превращению этой концепции в готовый продукт, которым мог бы воспользоваться обычный веб-разработчик. В листинге 14.9 показан пример использования библиотеки Dojo для запуска Comet-подключения к ресурсам на стороне сервера (который работает с использованием сервера Cometd).

Листинг 14.9. Использование Dojo и ее библиотеки Cometd для подключения к потоковому серверу и прослеживания различных каналов передачи

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
  "http://www.w3.org/TR/html4/strict.dtd">
<html>
  <head>
    <title>Тестовая страница клиент-серверной системы Cometd</title>
    <script type="text/javascript">
      // Мы собираемся регистрировать все взаимодействие на
      // панели отладки
      djConfig = { isDebug: true };
    </script>
    <script type="text/javascript" src="../dojo/dojo.js"></script>
    <script type="text/javascript">
      dojo.require("dojo.io.cometd");
      dojo.addOnLoad(function() {
        // Установка базового URL для всего взаимодействия с cometd
        cometd.init({}, "/cometd");

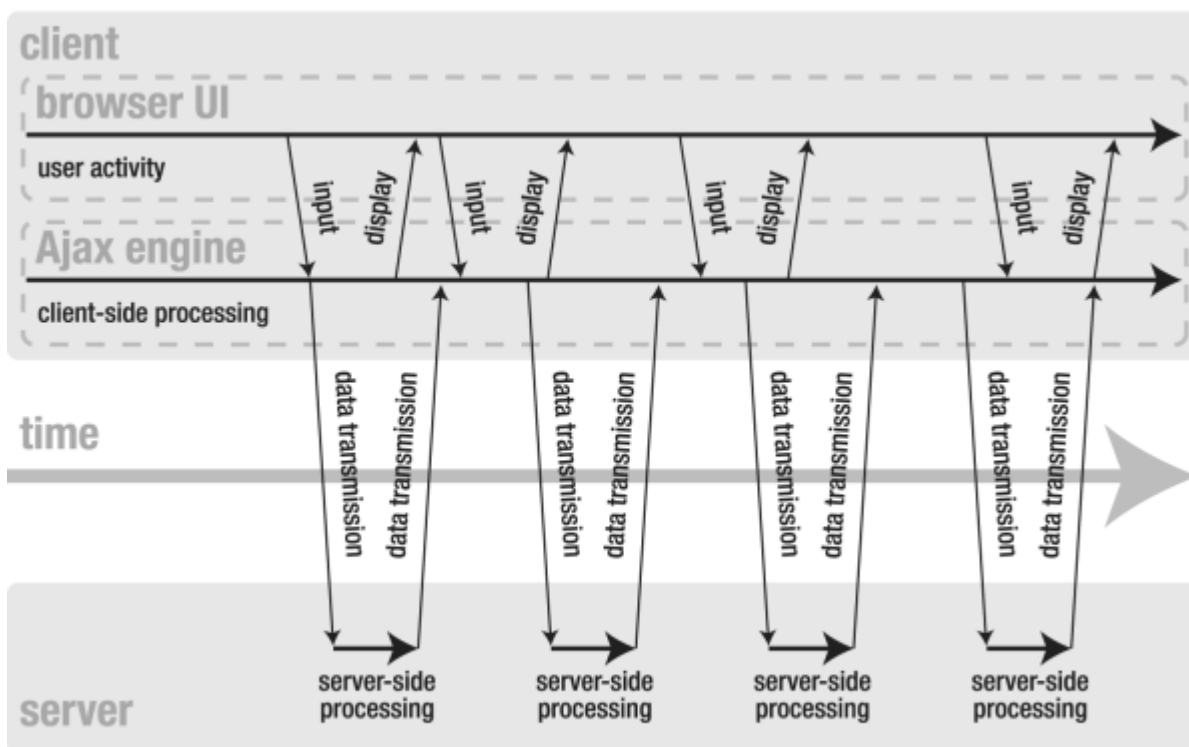
        // Подписка на отдельный пункт передачи
        // Будет отслеживаться весь выходящий поток, поступающий именно
        // из этой службы и регистрироваться на панели отладки
        cometd.subscribe("/foo/bar/baz", false, dojo, "debugShallow");

        // Распространение двух сообщений двум различным службам
        cometd.publish("/foo/bar/baz", { thud: "thonk!" });
        cometd.publish("/foo/bar/baz/xyzy",
          { foo: "A simple message" });
      });
    </script>
```

```
</head>  
<body></body>  
</html>
```

Хотя количество приложений, использующих Comet (или подобную Comet технологию) пока еще не очень велико, это количество обязательно увеличится по мере того, как все большее количество людей начнет понимать, насколько полезна именно эта часть технологии для создания высокопроизводительных веб-приложений.

Ajax web application model (asynchronous)



Comet web application model

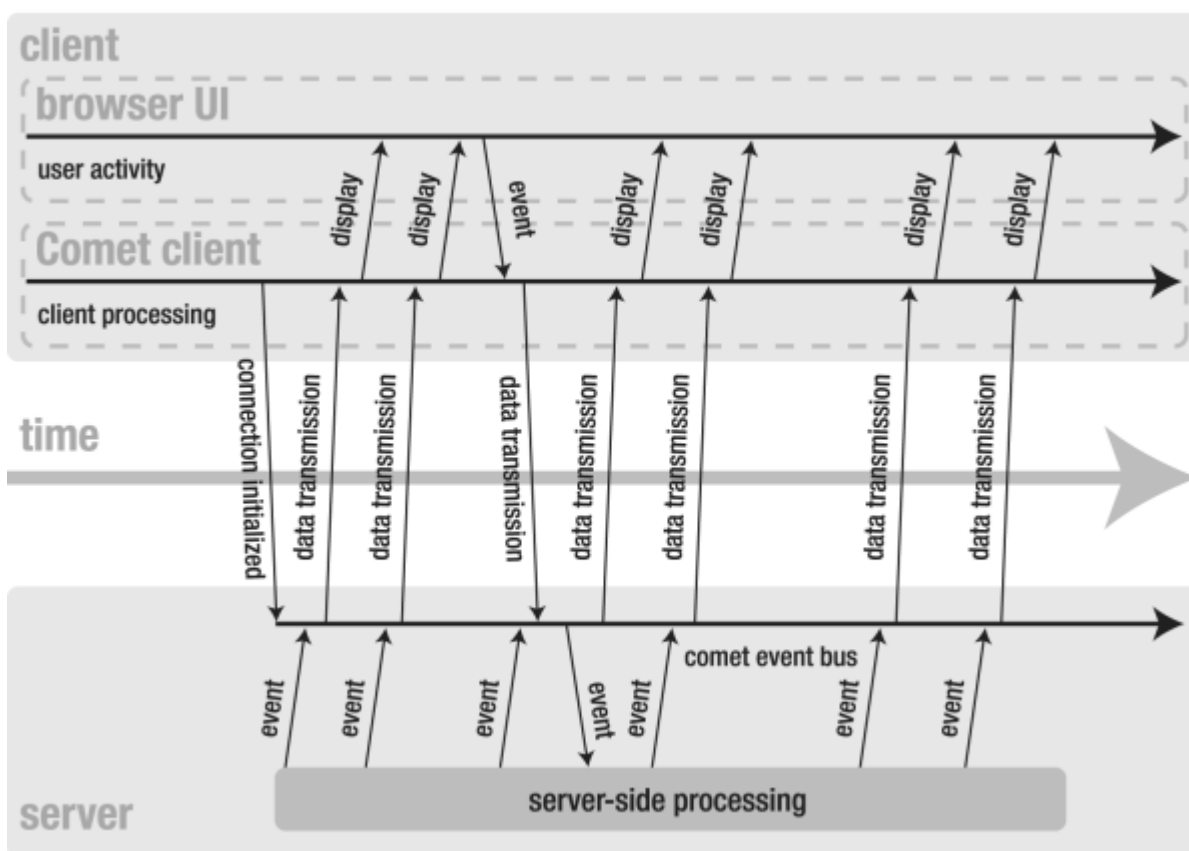


Рис. 14.3. Сравнение традиционной модели Ajax и новой модели веб-приложения, использующего методику Comet

Вывод

Технология, представленная в этой главе, относится к широкому диапазону, в нее входит все, начиная со сложной и отдаленной перспективы (как деструктуризация в JavaScript 1.7), и до текущих и широко используемых вещей (например, элемента `<canvas>`). Надеюсь, мне удалось дать вам хорошее представление о том направлении, в котором в ближайшем будущем будет двигаться веб-разработка, основанная на использовании браузеров.

Приложение А Справочник по DOM

Это приложение служит справочником по функциональным возможностям, предоставляемым объектной моделью документа, рассмотренной в главе 5.

Resources

Функциональные возможности DOM сформировались из множества разновидностей, начиная с исходной предварительной спецификации DOM Level 0, и развиваясь до вполне определенных DOM Levels 1 и 2. Поскольку все современные браузеры практически полностью поддерживают разработанные W3C спецификации DOM Levels 1 и 2, веб-сайт W3C служит превосходным справочным пособием для изучения порядка работы DOM:

- *DOM Level 1*: <http://www.w3.org/TR/REC-DOM-Level-1/level-one-core.html>
- *HTML DOM Level 1*: <http://www.w3.org/TR/REC-DOM-Level-1/level-one-html.html>
- *DOM Level 2*: <http://www.w3.org/TR/DOM-Level-2-Core/>
- *HTML DOM Level 2*: <http://www.w3.org/TR/DOM-Level-2-HTML/>

Кроме этого существует ряд отличных справочных руководств для изучения функционирования DOM, но лучшими все же являются источники, находящиеся на Quirksmode.org, веб-сайте, запущенном Питером-Паулем Кохом (Peter-Paul Koch). Он провел всесторонние исследования каждого доступного DOM-метода и сравнил свои результаты на всех современных браузерах (и не только на них). Это бесценный источник для определения того, что возможно, а что невозможно на тех браузерах, для которых вы ведете разработку:

- *W3C DOM Core Levels 1 and 2 reference*: http://www.quirksmode.org/dom/w3c_core.html
- *W3C DOM HTML Levels 1 and 2 reference*: http://www.quirksmode.org/dom/w3c_html.html

Терминология

В главе 5, посвященной объектной модели документа, и в этом приложении, я использую общепринятую XML и DOM терминологию для описания различных аспектов DOM-представления XML-документа. Следующие слова и фразы представляют терминологию, которая относится к объектной модели документа и, в общем, к документу XML. Все терминологические примеры будут относиться к образцовому HTML-документу, показанному в листинге А.1.

Листинг А.1. Пример, на который мы будем ссылаться при рассмотрении DOM и XML терминологии

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
  <title>Introduction to the DOM</title>
</head>
<body>
  <h1>Introduction to the DOM</h1>
  <p class="test">There are a number of reasons why the DOM is awesome,
  here are some:</p>
  <ul>
    <li id="everywhere">It can be found everywhere.</li>
    <li class="test">It's easy to use.</li>
```

```

    <li class="test">It can help you to find what you want, really
                                                quickly.</li>
</ul>
</body>
</html>

```

Предок

Термин, очень похожий на генеалогический, ссылается на родителя текущего элемента, а также на родителя этого родителя, и на родителя этого родителя и т.д. В листинге A.1 родительскими для элемента `` являются элемент `<body>` и элемент `<html>`.

Атрибут

Атрибуты являются свойствами элементов, в которых содержится принадлежащая им дополнительная информация. В листинге A.1 у элемента `<p>` есть атрибут `class`, в котором содержится значение `test`.

Дочерний элемент

Любой элемент может содержать любое количество узлов (каждый из которых рассматривается в качестве дочернего по отношению к родительскому элементу). В листинге A.1 элемент `` содержит семь дочерних узлов; три дочерних узла являются элементами ``, а остальные четыре — конечными строками, находящимися внутри каждого элемента (содержащихся в пределах текстовых узлов).

Элемент Document

Каждый XML-документ состоит из одного элемента (так называемого корневого узла или элемента `document`), который содержит все остальные аспекты документа. В листинге A.1 элементом `document` является `<html>`, в котором содержится весь остальной документ.

Потомки

Потомки элементов содержат его дочерние узлы, детей его детей, детей их детей и т.д. В листинге A.1 потомки элемента `<body>` включают `<h1>`, `<p>`, `` и ``-элементы, а также все текстовые узлы, которые содержатся во всех этих элементах.

Элемент

Элемент является контейнером, в котором содержатся атрибуты и другие узлы. Первичными и наиболее примечательными компонентами любого HTML-документа являются его элементы. В листинге A.1 присутствует множество элементов; все теги — `<html>`, `<head>`, `<title>`, `<body>`, `<h1>`, `<p>`, `` и `` — являются элементами.

Узел

Узел является основной составной частью DOM-представления. Элементы, атрибуты, комментарии, документы и текстовые узлы — все являются узлами, и поэтому обладают стандартными свойствами узлов (к примеру, `nodeType`, `nodeName` и `nodeValue` имеются в каждом узле).

Родитель

Термин «родитель» используется для ссылки на элемент, в котором содержится текущий узел. Родитель есть у всех узлов, кроме корневого. В листинге A.1 родителем для элемента `<p>` является элемент `<body>`.

Сестры

Сестринский узел является дочерним узлом того же самого родительского узла. Обычно этот термин используется в контексте `previousSibling` и `nextSibling`, двух свойств, имеющих у всех DOM-узлов. В листинге A.1 сестринскими для элемента `<p>` являются элементы `<h1>` и `` (вместе с несколькими пустыми текстовыми узлами).

Текстовые узлы

Текстовым называется особый узел, который содержит только текст; сюда включается видимый текст и все виды пустых пространств. Если вы видите текст внутри элемента (например, `hello world!`), значит, внутри элемента `` фактически находится отдельный текстовый узел, содержащий текст «hello world!». Текст «It's easy to use» в листинге A.1, который находится внутри второго элемента ``, содержится внутри текстового узла.

Глобальные переменные

Глобальные переменные существуют в пределах глобальной области видимости вашего кода, и существуют для того чтобы помочь вам работать с общими DOM-операциями.

document

Эта переменная содержит активный HTML DOM-документ, который просматривается в браузере. Но сам факт существования этой переменной и наличия в ней какого-нибудь значения еще не означает, что ее содержимое было полностью загружено и прошло синтаксический анализ. Более подробные сведения об ожидании загрузки DOM приведены в главе 5. В листинге A.2 показан ряд примеров использования переменной `document`, которая содержит представление HTML DOM для обращения к элементам документа.

Листинг A.2. Использование переменной `document` для доступа к элементам документа

```
// Обнаружение элемента, ID которого имеет значение 'body'
document.getElementById("body")

// Обнаружение всех элементов, имеющих имена тегов <div>.
document.getElementsByTagName("div")
```

HTMLElement

Эта переменная является надклассовым объектом для всех HTML DOM-элементов. Продолжение прототипа этого элемента распространяется на все HTML DOM-элементы. Этот суперкласс доступен по умолчанию во всех браузерах на движке Mozilla и в браузере Opera. Используя методы, описанные в главе 5, его также можно добавить к Internet Explorer и Safari. В листинге A.3 показан пример привязки новых функций к глобальному суперклассу `HTMLElement`. Присоединение функции `hasClass` дает возможность увидеть, имеет ли элемент определенный класс.

Листинг A.3. Привязка новых функций к глобальному суперклассу `HTMLElement`

```
// Добавление ко всем HTML DOM-элементам нового метода,
// который может быть использован, чтобы увидеть
// имеется ли у элемента определенный класс или нет.
HTMLElement.prototype.hasClass = function( class ) {
```

```
return new RegExp("(^|\\s)" + class + "(\\s|$)").test( this.className );
};
```

Перемещение по DOM

Следующие свойства являются составной частью всех DOM-элементов и могут быть использованы для перемещения по DOM-документам.

body

Это свойство глобального HTML DOM-документа (переменной `document`) непосредственно указывает на HTML-элемент `<body>` (который должен быть только один). Это свойство относится к тем, которые были перенесены еще из времен DOM 0 JavaScript. В листинге A.4 показано несколько примеров обращения к элементу `<body>` из HTML DOM-документа.

Листинг A.4. Обращение к элементу `<body>` внутри HTML DOM-документа

```
// Изменение полей <body>
document.body.style.margin = "0px";

// document.body эквивалентно следующему выражению:
document.getElementsByTagName("body")[0]
```

childNodes

Это свойство имеется у всех DOM-элементов и содержит массив всех дочерних узлов (включая элементы, текстовые узлы, комментарии и т.д.). Оно предназначено только для чтения. В листинге A.5 показано, как нужно использовать свойство `childNodes` для добавления стилевой установки ко все дочерним элементам данного родителя.

Листинг A.5. Добавление красного обрамления вокруг дочерних элементов, принадлежащих элементу `<body>` с использованием свойства `childNodes`

```
// Добавление обрамления ко всем, имеющимся у <body> дочерним элементам
var c = document.body.childNodes;
for ( var i = 0; i < c.length; i++ ) {
    // Нужно убедиться, что этот узел является элементом
    if ( c[i].nodeType == 1 )
        c[i].style.border = "1px solid red";
}
```

documentElement

Это свойство, которое имеется у всех DOM-узлов, действует в качестве ссылки на корневой элемент документа (в случае с HTML-документами, он всегда будет указывать на элемент `<html>`). В листинге A.6 показан пример использования `documentElement` для поиска DOM-элемента.

Листинг A.6. Пример обнаружения корневого элемента из любого DOM-узла

```
// Обнаружение documentElement для поиска элемента по его ID
```

```
некийПроизвольныйУзел.documentElement.getElementById("body")
```

firstChild

Это свойство, имеющееся у всех DOM-элементов, указывает на первый дочерний узел данного элемента. Если у элемента нет дочерних узлов, `firstChild` будет равен нулю. В листинге А.7 показан пример использования свойства `firstChild` для удаления всех дочерних узлов данного элемента.

Листинг А.7. Удаление из элемента всех дочерних узлов

```
// Удаление из элемента всех дочерних узлов
var e = document.getElementById("body");
while ( e.firstChild )
    e.removeChild( e.firstChild );
```

getElementById(элемID)

Это весьма эффективная функция, которая обнаруживает в документе один из элементов, имеющий определенный ID. Функция доступна только элементу `document`. Кроме этого, функция может не работать, если будет применена к DOM-документам, не имеющим отношения к HTML; как правило, при работе XML DOM-документами нужно точно указать атрибут ID в DTD (определении типа документа — Document Type Definition) или в схеме.

Как показано в листинге А.8, эта функция принимает единственный аргумент: имя искомого ID.

Листинг А.8. Два примера обнаружения HTML-элементов по именам их ID-атрибутов

```
// Обнаружение элемента по значению ID, равному body
document.getElementById("body")

// Скрытие элемента, у которого значение ID равно notice
document.getElementById("notice").style.display = 'none';
```

getElementsByTagName(имяТега)

Это свойство отыскивает все элементы-потомки, которые имеют определенное имя тега, начиная с текущего элемента. Эта функция одинаково хорошо работает как XML DOM, так и в HTML DOM-документах.

Во всех современных браузерах можно задать в качестве имени тега знак звездочки (*) и найти все элементы-потомки, и это получится намного быстрее, чем при использовании рекурсивной функции, написанной на чистом JavaScript.

Эта функция принимает единственный аргумент: имя тега искомым элементов. В листинге А.9 показан пример использования `getElementsByTagName`. Первый блок добавляет класс `highlight` всем `<div>`-элементам документа. Второй блок находит все элементы внутри элемента, у которого значение ID равно `body`, и прячет те из них, которые имеют класс `highlight`.

Листинг А.9. Два блока кода, в которых демонстрируется использование `getElementsByTagName`

```
// Обнаружение всех элементов <div> в текущем HTML-документе
// и установка значения их атрибута class в 'highlight'
var d = document.getElementsByTagName("div");
for ( var i = 0; i < d.length; i++ ) {
    d[i].className = 'hilite';
}

// Проход по всем элементам-потомкам элемента, у которого значение
// ID равно body. Затем обнаружение всех элементов, у которых значение
// аргумента class равно 'hilite'. После чего скрывание всех элементов,
// соответствующих этому условию.
var all = document.getElementById("body").getElementsByTagName("*");
for ( var i = 0; i < all.length; i++ ) {
    if ( all[i].className == 'hilite' )
        all[i].style.display = 'none';
}
```

lastChild

Эта ссылка, доступная во всех DOM-элементах, указывает на последний дочерний узел этого элемента. Если дочерних узлов нет, значение lastChild будет равно нулю (null). В листинге A.10 показан пример использования свойства lastChild для вставки элемента в документ.

Листинг A.10. Создание нового <div>-элемента и вставка его перед последним элементом в <body>

```
// Вставка нового элемента непосредственно перед последним элементом в
// <body>
var n = document.createElement("div");
n.innerHTML = "Спасибо за визит!";

document.body.insertBefore( n, document.body.lastChild );
```

nextSibling

Эта ссылка, доступная во всех DOM-узлах, указывает на последний сестринский узел. Если узел является последним сестринским, значение nextSibling будет равно нулю (null). Важно помнить о том, что свойство nextSibling может указывать на DOM-элемент, комментарий или даже на текстовый узел; поэтому оно не служит исключительным способом перемещения по DOM-элементам. В листинге A.11 показан пример использования свойства nextSibling для создания интерактивного списка определений.

Листинг A.11. Принуждение всех элементов <dt> по щелчку раскрывать свои сестринские элементы <dd>

```
// Обнаружение всех элементов <dt> (Defintion Term — определяемый термин)
var dt = document.getElementsByTagName("dt");
for ( var i = 0; i < dt.length; i++ ) {
    // Отслеживание щелчка на термине
    dt[i].onclick = function() {
```

```

// Поскольку каждый термин имеет прикрепленный к нему
// элемент <dd> (Definition — определение),
// мы можем его отобразить по щелчку
// ПРИМЕЧАНИЕ: Работает при условии, что между элементами <dd>
// отсутствуют пустые пространства
this.nextSibling.style.display = 'block';
};
}

```

parentNode

Это свойство есть у всех DOM-узлов. Свойство parentNode каждого DOM-узла указывает на элемент, которые его содержит, за исключением элемента document, который указывает на null (поскольку ничто не содержит корневой элемент). В листинге A.12 показан пример использования свойства parentNode для создания собственного сценария взаимодействия. Щелчок на кнопке Cancel (Отмена) приводит к скрытию родительского элемента.

Листинг A.12. Использование свойства parentNode для создания собственного сценария взаимодействия

```

// Отслеживание щелчка на кнопке (например, Cancel)
// и скрытие родительского элемента
document.getElementById("cancel").onclick = function(){
    this.parentNode.style.display = 'none';
};

```

previousSibling

Эта ссылка, доступная во всех DOM-узлах, указывает на предыдущий сестринский узел. Если узел является первым сестринским, значение previousSibling будет равно нулю (null). Важно помнить о том, что свойство previousSibling может указывать на DOM-элемент, комментарий или даже на текстовый узел; поэтому оно не служит исключительным способом перемещения по DOM-элементам. В листинге A.13 показан пример использования свойства previousSibling для скрытия элементов.

Листинг A.13. Скрытие всех элементов перед текущим элементом

```

// Обнаружение всех элементов перед текущим, и их скрытие
var cur = this.previousSibling;
while ( cur != null ) {
    cur.style.display = 'none';
    cur = this.previousSibling;
}

```

Информация об узле

Эти свойства имеются у большинства DOM-элементов с целью предоставления простого доступа к общей информации об элементе.

innerText

Это свойство есть у всех DOM-элементов (которые существуют только в тех браузерах, которые не работают на движках Mozilla, поскольку оно не является частью стандарта W3C). Оно возвращает строку, содержащую весь текст, находящийся внутри текущего элемента. Поскольку это свойство не поддерживается браузерами на движках Mozilla, вы можете воспользоваться обходным путем, подобным одному из описанных в главе 5 (где мы использовали функцию для сбора значений текстовых узлов-потомков). В листинге A.14 показан пример использования свойства `innerText` и функции `text()` из главы 5.

Листинг A.14. Использование свойства `innerText` для извлечения из элемента текстовой информации

```
// Предположим, что у нас есть элемент <li> похожий на этот,
// сохраненный в переменной 'li':
// <li>Пожалуйста, посетите <a href="http://mysite.com/">
//                               мой веб-сайт </a>.</li>

// Использование свойства innerText
li.innerText

// или функции text(), рассмотренной в главе 5
text( li )

// Результат использования свойства или функции будет следующим:
"Пожалуйста, посетите мой веб-сайт."
```

nodeName

Это свойство доступно для всех DOM-элементов, содержащих версию имени элемента в верхнем регистре. Например, если у вас имеется элемент ``, и вы обращаетесь к его свойству `nodeName`, он вернет `LI`. В листинге A.15 показан пример использования свойства `nodeName` для модификации имен классов родительских элементов.

Листинг A.15. Обнаружение всех родительских элементов `` и установка значение их класса в `current`

```
// Обнаружение всех родителей данного узла, являющихся элементом <li>
var cur = this.parentNode;
while ( cur != null ) {
    // Как только элемент найден и его имя проверено, добавить класс
    if ( cur.nodeName == 'LI' )
        cur.className += " current";
    cur = this.parentNode;
}
```

nodeType

Это общее свойство всех DOM-узлов, содержащее номер, соответствующий типу данного узла. Существует три наиболее широко распространенных типа узлов, используемых в HTML-документах:

- Узел элемента (имеет значение 1 или `document.ELEMENT_NODE`)
- Текстовый узел (имеет значение 3 или `document.TEXT_NODE`)
- Узел документа (имеет значение 9 или `document.DOCUMENT_NODE`)

Использование свойства `nodeType` является надежным способом убедиться в том, что узел, к которому вы пытаетесь получить доступ, обладает всеми ожидаемыми свойствами (к примеру, свойство `nodeName` может использоваться только для DOM-элементов; поэтому перед обращением к нему можно воспользоваться `nodeType`, чтобы убедиться, что это свойство равно 1). В листинге A.16 показан пример использования свойства `nodeType` для добавления класса многим элементам.

Листинг A.16. Обнаружение первого элемента в HTML `<body>` и использование для него класса `header`

```
// Обнаружение в <body> первого элемента
var cur = document.body.firstChild;
while ( cur != null ) {
    // Если элемент был найден, добавить к нему класс header
    if ( cur.nodeType == 1 ) {
        cur.className += " header";
        cur = null;

        // В противном случае продолжить проход по дочерним узлам
    } else {
        cur = cur.nextSibling;
    }
}
```

nodeValue

Это полезное свойство текстовых узлов, оно может быть использовано для доступа и работы с текстом, который в них содержится. Хорошим примером его использования может послужить функция `text`, представленная в главе 5, которая используется для извлечения текстового содержимого элемента. В листинге A.17 показан пример использования свойства `nodeValue` для создания простой функции извлечения текстового значения.

Листинг A.17. Функция, принимающая элемент и возвращающая текстовое содержимое этого элемента и его элементов-потомков

```
function text(e) {
    var t = "";

    // Если элемент передан, обнаружение его дочерних элементов,
    // если нет, предположение, что это массив
    e = e.childNodes || e;

    // просмотр всех дочерних узлов
    for ( var j = 0; j < e.length; j++ ) {
        // Если это не элемент, добавление его текстового значения,
        // в противном случае рекурсивное обращение ко всем его дочерним
        // элементам
        t += e[j].nodeType != 1 ?
            e[j].nodeValue : text(e[j].childNodes);
    }
}
```

```
// Возвращение соответствующего текста
return t;
}
```

Атрибуты

Большинство атрибутов доступно в виде свойств своих элементов. Например, атрибут ID может быть получен с использованием простого выражения `element.id`. Эта особенность осталась со времен DOM 0, и скорее всего оно так и останется, благодаря его простоте и популярности.

className

Это свойство позволяет добавлять и удалять классы в DOM-элементе. Оно существует во всех элементах DOM. Причиной его отдельного упоминания стало название — `className`, в котором нарушен ожидаемый порядок слов. Это странное название связано с тем, что слово `class` зарезервировано в большинстве объектно-ориентированных языков программирования; поэтому его использования избегают, чтобы не создавать трудности в программировании веб-браузера. В листинге A.18 показан пример использования свойства `className` для скрытия некоторых элементов.

Листинг A.18. Обнаружение всех `<div>`-элементов, имеющих класс `special` и их скрытие

```
// Обнаружение всех имеющихся в документе элементов <div>
var div = document.getElementsByTagName("div");
for ( var i = 0; i < div.length; i++ ) {
    // Обнаружение всех элементов <div>, имеющих общий класс 'special'
    if ( div[i].className == "special" ) {
        // И их скрытие
        div[i].style.display = 'none';
    }
}
```

getAttribute(имяАтрибута)

Эта функция служит хорошим способом доступа к значению атрибута, содержащегося внутри DOM-элемента. Атрибуты инициализируются значением, которое пользователь предоставляет в обычном HTML-документе.

Функция принимает единственный аргумент: имя атрибута, значение которого нужно извлечь. В листинге A.19 показан пример использования функции `getAttribute()` для поиска элементов `input` определенного типа.

Листинг A.19. Обнаружение элемента `<input>`, у которого атрибут `name` имеет значение `text` и копирование его значения в элемент, у которого атрибут ID имеет значение `preview`

```
// Обнаружение всех элементов ввода данных формы — input
var input = document.getElementsByTagName("input");
for ( var i = 0; i < input.length; i++ ) {

    // Обнаружение элемента, у которого атрибут name имеет значение "text"
    if ( input[i].getAttribute("name") == "text" ) {
```



```

// Копирование значения в другой элемент
document.getElementById("preview").innerHTML =
    input[i].getAttribute("value");
}
}

```

removeAttribute(имяАтрибута)

Эту функцию можно использовать для полного удаления атрибута из элемента. Обычно результат использования этой функции можно сравнить с результатом применения функции `setAttribute` со значением " " (пустой строки) или `null`; но на практике лучше все же всегда полностью удалять лишние атрибуты, чтобы избежать любых неожиданных последствий.

Эта функция принимает единственный аргумент: имя атрибута, требующего удаления. В листинге А.20 показан пример снятия в форме некоторых флажков.

Листинг А.20. Обнаружение всех флажков, имеющихся в документе, и их снятие

```

// Обнаружение всех элементов ввода данных формы
var input = document.getElementsByTagName("input");
for ( var i = 0; i < input.length; i++ ) {

    // Обнаружение всех флажков
    if ( input[i].getAttribute("type") == "checkbox" ) {

        // Снятие флажков
        input[i].removeAttribute("checked");
    }
}
}

```

setAttribute(attrName, attrValue)

Эта функция служит хорошим способом установки значений атрибута, содержащегося внутри DOM-элемента. Кроме этого она может добавить ваши собственные атрибуты, к которым позже можно будет обратиться, не оказывая влияния на появление DOM-элементов. Функция `setAttribute` склонна вести себя в Internet Explorer несколько странным образом, не позволяя устанавливать определенные атрибуты (например, `class` или `maxlength`). Все это подробнее объяснено в главе 5.

Функция принимает два аргумента. Первый — имя атрибута, а второй — присваиваемое ему значение. В листинге А.21 показан пример установки значения атрибута DOM-элемента.

Листинг А.21. Использование функции `setAttribute` для создания <a>-ссылки на Google

```

// Создание нового элемента <a>
var a = document.createElement("a").

// Установка URL для посещения веб-сайта Google
a.setAttribute("href", "http://google.com/");

```

```
// Добавление внутреннего текста, на котором пользователь может сделать
// щелчок
a.appendChild( document.createTextNode( "Посетите Google!" ) );

// Добавление ссылки в самый конец документа
document.body.appendChild( a );
```

Модификация DOM

Здесь представлены все свойства и функции, которые можно использовать для работы с DOM.

appendChild(добавляемыйУзел)

Эту функцию можно использовать для добавления дочернего узла к элементу-контейнеру. Если дополняемый узел уже существует в документе, он перемещается со своего текущего места и добавляется к текущему элементу. Функция `appendChild` должна быть вызвана для элемента, который требуется дополнить.

Функция принимает единственный аргумент: ссылку на DOM-узел (он может быть только что созданным, или ссылка может быть на узел, который уже существует в каком-нибудь месте документа). В листинге A.22 показан пример создания нового элемента `` и перемещения в него всех ``-элементов с их первоначального места в DOM, а затем добавления нового `` к телу документа.

Листинг A.22. Добавление серии ``-элементов к отдельному ``

```
// Создание нового элемента <ul>
var ul = document.createElement("ul");

// Обнаружение всех первых элементов <li>
var li = document.getElementsByTagName("li");
for ( var i = 0; i < li.length; i++ ) {

    // добавление каждого соответствующего <li> к новому
    // элементу <ul>
    ul.appendChild( li[i] );
}

// Добавление нашего нового элемента <ul> к концу тела документа — body
document.body.appendChild( ul );
```

cloneNode(true|false)

Эта функция является для разработчиков способом упрощения кода за счет создания дубликатов существующих узлов, которые затем могут быть вставлены в DOM. Если обычный вызов `insertBefore` или `appendChild` приведет к физическому перемещению DOM-узла в документе, то функция `cloneNode` может быть использована для создания его дубликата.

Функция принимает один из аргументов — `true` или `false`. Если передан аргумент `true`, то копируется узел со всем его содержимым; а если `false`, то копируется только сам узел. В листинге A.23 показан пример использования этой функции для клонирования элемента и его добавления к нему самому.

Листинг А.23. Обнаружение в документе первого элемента ``, создание его полной копии и добавление ее к нему самому

```
// Обнаружение первого элемента <ul>
var ul = document.getElementsByTagName("ul")[0];

// клонирование узла и добавление его сразу после старого узла
ul.parentNode.appendChild( ul.cloneNode( true ) );
```

createElement(имяТега)

Это основная функция, используемая для создания новых элементов внутри DOM-структуры. Функция существует в виде свойства документа, внутри которого вы хотите создать элемент.

ПРИМЕЧАНИЕ

Если вы используете XHTML, для которого указан тип контекста (content-type) `application/xhtml+xml`, вместо обычного HTML, для которого указан тип контекста `text/html`, то вместо функции `createElement` нужно использовать функцию `createElementNS`.

Эта функция принимает один аргумент: имя тега создаваемого элемента. В листинге А.24 показан пример использования этой функции для создания элемента и помещения в него нескольких других элементов.

Листинг А.24. Помещение содержимого элемента `<p>` в элемент ``

```
// Создание нового элемента <strong>
var s = document.createElement("strong");

// Обнаружение первого абзаца
var p = document.getElementsByTagName("p")[0];

// Помещение содержимого <p> в элемент <strong>
while ( p.firstChild ) {
    s.appendChild( p.firstChild );
}

// Помещение элемента <strong> (содержащего прежнее содержимое элемента <p>)
// обратно в элемент <p>
p.appendChild( s );
```

createElementNS(пространство_имен, имяТега)

Эта функция очень похожа на функцию `createElement`, она тоже создает новый элемент, но она также предоставляет возможность указать для элемента пространство имен (если, к примеру, элемент добавляется к документу XML или XHTML).

Эта функция принимает два аргумента: пространство имен добавляемого элемента, и имя тега элемента. В листинге А.25 показан пример использования этой функции для создания DOM-элемента в допустимом XHTML-документе.

Листинг А.25. Создание нового XHTML <p>-элемента, заполнение его некоторым текстом и добавление его к телу документа

```
// Создание нового XHTML-совместимого <p>
var p = document.createElementNS("http://www.w3.org/1999/xhtml", "p");

// Добавление в элемент <p> некоторого текста
p.appendChild( document.createTextNode( "Welcome to my site." ) );

// Добавление элемента <p> в документ
document.body.insertBefore( p, document.body.firstChild );
```

createTextNode(текстоваяСтрока)

Это подходящий способ создания новой текстовой строки для вставки ее в DOM-документ. Поскольку текстовые узлы это всего лишь существующая только в DOM оболочка текста, важно помнить, что они не могут быть стилизованы или дополнены. Эта функция существует только как свойство DOM-документа.

Функция принимает один аргумент: строку, которая станет содержимым текстового узла. В листинге А.26 показан пример использования этой функции для создания нового текстового узла и добавления его к телу HTML-страницы.

Листинг А.26. Создание элемента <h1> и добавление нового текстового узла

```
// Создание нового элемента <h1>
var h = document.createElement("h1");

// Создание текста заголовка и добавление его к элементу <h1>
h.appendChild( document.createTextNode("Главная страница") );

// Добавление заголовка в начале <body>
document.body.insertBefore( h, document.body.firstChild );
```

innerHTML

Это свойство, характерное для HTML DOM, предназначено для обращения к текстовой версии HTML-содержимого DOM-элемента и работе с ним. Если вы работаете только с HTML-документом (который не имеет отношения к XML), этот метод может быть очень полезным, поскольку код, который нужен для генерации нового DOM-элемента, может быть существенно сокращен (если не считать, что это более быстрая альтернатива традиционным DOM-методам). Хотя это свойство не входит ни в один W3C-стандарт, оно все же существует во всех современных браузерах.

В листинге А.27 показан пример использования свойства innerHTML для изменения содержимого элемента при изменении содержимого <textarea>.

Листинг А.27. Отслеживание изменения в <textarea> и обновление предварительного просмотра его значением в реальном времени

```
// Получение textarea для отслеживания обновлений
```

```
var t = document.getElementsByTagName("textarea")[0];

// Захват текущего значения <textarea> и обновление предварительного
// просмотра в реальном времени при каждом изменении
t.onkeypress = function() {
    document.getElementById("preview").innerHTML = this.value;
};
```

insertBefore(узелДляВставки, узелПередКоторымВставляем)

Эта функция используется для вставки DOM-узла в любое место документа. Она должна быть вызвана для родительского элемента узла, перед которым нужно сделать вставку. Так сделано для того, чтобы вы могли указать null в качестве узла, перед которым нужно сделать вставку, чтобы ваш узел был вставлен в качестве последнего дочернего узла.

Функция принимает два аргумента. Первый аргумент является узлом, который нужно вставить в DOM, а второй — это DOM-узел, перед которым будет осуществлена вставка. Он должен ссылаться на существующий узел. В листинге A.28 показан пример использования этой функции для вставки значка сайта (который предшествует URL в адресной строке браузера) рядом с набором URL веб-сайтов на странице.

Листинг A.28. Проход по всем элементам <a> и добавление изображения значка веб-сайта

```
// Обнаружение в документе всех ссылок <a>
var a = document.getElementsByTagName("a");
for ( var i = 0; i < a.length; i++ ) {

    // Создание изображения значка веб-сайта, на который указывает
    // ссылка
    var img = document.createElement("img");
    img.src = a[i].href.split('/').splice(0,3).join('/') + '/favicon.ico';

    // Вставка изображения перед ссылкой
    a[i].parentNode.insertBefore( img, a[i] );
}
```

removeChild(удаляемыйУзел)

Эта функция используется для удаления узла из DOM-документа. Функция removeChild должна быть вызвана для родительского элемента того узла, который нужно удалить.

Функция принимает один аргумент: ссылку на DOM-узел, удаляемый из документа. В листинге A.29 показан пример ее запуска в отношении всех <div>-элементов документа и удаления каждого из них, который имеет особый класс warning.

Листинг A.29. Удаление всех элементов, которые имеют особое имя класса

```
// Обнаружение всех элементов <div>
var div = document.getElementsByTagName("div");
for ( var i = 0; i < div.length; i++ ) {
```

```
// Если значение класса <div> равно 'warning'
if ( div[i].className == "warning" ) {

    // Удалить <div> из документа
    div[i].parentNode.removeChild( div[i] );

}
}
```

replaceChild(вставляемыйУзел, заменяемыйУзел)

Эта функция служит альтернативой процессу удаления узла и вставляет на его место другой узел. Она может быть вызвана родительским элементом заменяемого узла.

Эта функция принимает два аргумента: узел, который нужно вставить в DOM, и узел, который вы собираетесь заменить. В листинге A.30 показан пример замены всех элементов <a> элементом , содержащим URL, на который первоначально была сделана ссылка.

Листинг A.30. Преобразование набора ссылок в обычные URL

```
// Преобразование всех ссылок в видимые URL (что хорошо подойдет для
// распечатки)
// Обнаружение в документе всех ссылок <a>
var a = document.getElementsByTagName("a");
while ( a.length ) {

    // Создание элемента <strong>
    var s = document.createElement("strong");

    // Создание содержимого, эквивалентного URL, на который ссылается <a>
    s.appendChild( document.createTextNode( a[i].href ) );

    // Замена исходного <a> новым элементом <strong>
    a[i].replaceChild( s, a[i] );

}
}
```

Приложение Б Справочник по событиям

Это приложение служит справочником для главы 6, посвященной событиям. Оно предоставляет полное описание всех возможных DOM-событий и дополняет общую теорию, представленную в главе 6. Здесь вы найдете источники дополнительной информации, определения общепринятой терминологии, связанной с событиями, и объяснения всех общих объектов событий и взаимодействия с ними.

Источники информации

Если выбирать только один источник информации, к которому нужно обратиться за дополнительными сведениями о событиях, то это Quirksmode.org. Этот веб-сайт предоставляет сравнительную характеристику для каждого события во всех современных браузерах. Я настоятельно рекомендую на него зайти и составить представление о том, какие события поддерживает каждый из браузеров (http://www.quirksmode.org/js/events_compinfo.html).

Кроме этого были использованы две наиболее популярные спецификации: W3C DOM-события и Internet Explorer HTML-события. На каждом веб-сайте имеется подробный список всех возможных событий и изложена каждая особенность их поведения:

- W3C DOM Level 2 events: <http://www.w3.org/TR/DOM-Level-2-Events/events.html>
- Internet Explorer HTML events: <http://msdn.microsoft.com/workshop/author/dhtml/reference/events.asp>

Терминология

В этом разделе определяется ряд новых терминов, введенных по теме обработки событий в JavaScript, которые могут быть незнакомы для тех, кто не работал с событиями JavaScript, или с асинхронной обработкой событий.

Асинхронный

Асинхронные события имеют общую структуру обратного вызова, в отличие от потоковой прикладной структуры. Это означает, что какой-то единственный фрагмент кода (обратный вызов) зарегистрирован в качестве обработчика события. При наступлении события выполняется обратный вызов.

Прикрепление / Привязка / Регистрация обратного вызова

Прикрепление (иногда называемое привязкой) обратного вызова к обработчику события является регистрацией кода в асинхронной модели события. Как только событие произойдет, вызывается обработчик события, в котором содержится ссылка на зарегистрированный обратный вызов. Этот обратный вызов является частью кода в форме функции, которая вызывается по ссылке, как только завершится наступление события.

Всплытие

Фаза всплытия события происходит после фазы захвата, относящейся к этому событию. Эта фаза начинается с источника события (такого как ссылка, на которой щелкнул пользователь) и перемещается вверх по дереву DOM к корневому элементу документа.

Захват

Фаза захвата (возникающая только в модели события W3C) это первая имеющая место фаза события, которая состоит из перемещения события вниз по дереву DOM к местоположению элемента, которому приписывается событие.

Исходное действие (или действие по умолчанию)

Исходное действие предопределяется браузером и совершается независимо от того, имеет пользователь привязанный обработчик события, или нет. Примером исходного действия, предоставляемого браузером может послужить переход на другую веб-страницу по щелчку пользователя на ссылке.

Событие

Событие — действие, которое запущено (инициировано) в пределах веб-страницы. Как правило, события инициируются пользователем (например, перемещение указателя мыши, нажатие клавиши и т.д.), но могут возникать и без участия пользователя (например, загрузка страницы, или возникновение ошибки).

Обработчик события

Обработчик события (в частности, ссылка на функцию) является кодом, который вызывается при наступлении события. Если в качестве обработчика события не был зарегистрирован обратный вызов, то ничего не произойдет (кроме исходного действия).

Потоковый

В потоковом приложении обычно существует несколько отдельно текущих процессов, выполняющих постоянную задачу (к примеру, отслеживание доступности ресурса). JavaScript не занимается какой-либо обработкой потоков, и является чисто асинхронным языком.

Объект события

Объект события является объектом, предоставляемым или доступным внутри каждой функции обработчика события. Их работа в Internet Explorer и в других браузерах происходит по-разному.

Браузеры, совместимые со спецификацией W3C, предоставляют функции обработчика события единственный аргумент, в котором содержится ссылка на объект события. В Internet Explorer объект события всегда доступен в свойстве `window.event`, доступ к которому может быть открыт только в обработчике события.

Общие свойства

Для любого типа захваченных событий в объекте события имеется ряд свойств. Все эти свойства объекта события относятся непосредственно к самому событию и не содержат никакой специфики, относящейся к типу события. Далее приводится список всех свойств объекта события с пояснениями и примерами кода.

type

Это свойство содержит имя текущего инициированного события (например, `click`, `mouseover` и т.д.). Оно может быть использовано для предоставления универсальной функции обработки события, которая затем предопределенно выполнить соответствующие функции (например, функции `addEventListener/removeEvent`, рассмотренные в главе 6). В листинге Б.1 показан пример использования этого свойства для создания обработчика, производящего различный эффект в зависимости от типа обрабатываемого события.

Листинг Б.1. Использование свойства `type` для придания элементу возможности изменения при прохождении над ним указателя мыши

```
// Обнаружение <div>, для которого создается эффект прохождения
var div = document.getElementsByTagName('div')[0];
```



```
// Привязка одной и той же функции к обоим событиям mouseover и mouseout
div.onmouseover = div.onmouseout = function(e) {
    // Нормализация объекта события
    e = e || window.event;

    // Переключение цвета фона <div>, в зависимости от типа
    // возникшего события мыши
    this.style.background = (e.type == 'mouseover') ? '#EEE' : '#FFF';
};
```

target / srcElement

Это свойство содержит ссылку на элемент, который инициировал событие. Например, привязка обработчика щелчка (click), к элементу <a> приведет к тому, что свойство target будет равно самому элементу <a>. Свойство srcElement является эквивалентом свойства target, но работает в Internet Explorer. В листинге Б.2 показан пример использования этого свойства для работы с глобальным обработчиком событий.

Листинг Б.2. Двойной щелчок на узле, принадлежащем HTML DOM, приводит к его удалению

```
// Привязка к документу отслеживателя двойного щелчка
document.ondblclick = function(e) {
    // Нормализация объекта события
    e = e || window.event;

    // Обнаружение правильного целевого узла
    var t = e.target || e.srcElement;

    // удаление узла из DOM
    t.parentNode.removeChild( t );
};
```

stopPropagation() / cancelBubble

Метод stopPropagation() останавливает процесс всплытия (или захвата) события, делая текущий элемент последним получателем данного события. Фазы события подробно рассмотрены в главе 6. Свойство cancelBubble доступно в Internet Explorer; если установить его значение в true, то это будет эквивалентно вызову метода stopPropagation() для W3C-совместимых браузеров. В листинге Б.3 показан пример использования этой технологии для прекращения распространения события.

Листинг Б.3. Динамическая подсветка всех элементов , имеющих в документе

```
// Обнаружение в документе всех элементов <li>
var li = document.getElementsByTagName('li');
for ( var i = 0; i < li.length; i++ ) {

    // Отслеживание прохождения над <li> указателя мыши
    li[i].onmouseover = function(e) {
        // Если браузер W3C-совместимый
        if ( e )
            // Использование stopPropagation для остановки всплытия
            e.stopPropagation();
    };
};
```

```

// Если нет, то это Internet Explorer
else
    // поэтому для остановки всплытия cancelBubble
    // устанавливается в true
    e.cancelBubble = true;

// В заключение, подсветка фона элемента <li>
this.style.background = '#EEE';
};

// Когда указатель мыши уходит с <li>
li[i].onmouseout = function(){
    // Переключение фонового цвета обратно на белый
    this.style.background = '#FFF';
};
}

```

preventDefault() / returnValue = false

Вызов метода `preventDefault()` останавливает выполнение исходного действия браузера во всех современных W3C-совместимых браузерах. В Internet Explorer, чтобы остановить выполнение исходного действия браузера, нужно установить значение свойства `returnValue`, принадлежащее объекту события, в `false`.

Объяснение процесса выполнения исходного действия приведено в главе 6. Код в листинге Б.4 делает так, что при каждом щелчке на расположенной на странице ссылке, вместо перехода на другую страницу (как это обычно происходит), заголовок документа становится URL ссылки.

Листинг Б.4. Предотвращение исходного действия браузера

```

// Обнаружение на странице всех элементов <a>
var a = document.getElementsByTagName('a');
for ( var i = 0; i < a.length; i++ ) {

    // Привязка к <a> обработчика щелчка
    a[i].onclick = function(e) {
        // Установка заголовка страницы в качестве URL этой ссылки, вместо
        // перехода по прежнему URL
        document.title = this.href;

        // Предотвращение визитов браузера на веб-сайт, указанный в
        // <a> (что является его исходным действием)
        if ( e ) {
            e.preventDefault();

            // Предотвращение действия по умолчанию в IE
        } else {
            window.event.returnValue = false;
        }
    };
}

```

}

Свойства мыши

Свойства мыши существуют в объекте события только при инициации событий, связанных с мышью (среди которых `click`, `mousedown`, `mouseup`, `mouseover`, `mousemove` и `mouseout`). Во всех остальных случаях можно считать, что возвращенные значения не существуют или наверняка отсутствуют. В этом разделе приводится перечень всех свойств, которые присутствуют в объекте события в то время, когда имеет место событие, связанное с мышью.

`clientX` / `clientY`

Это свойство содержит координаты `x` и `y` указателя мыши относительно окна браузера. Пример использования этого свойства показан в листинге Б.5.

Листинг Б.5. Определение текущей позиции указателя мыши на веб-странице

```
// Определение горизонтальной позиции указателя
function getX(e) {
    // Сначала проверка позиции в не-IE браузере, потом в IE,
    // и только потом возвращение нуля
    return e.pageX || (e.clientX +
        (document.documentElement.scrollLeft || document.body.scrollLeft));
}

// Определение вертикальной позиции указателя
function getY(e) {
    // Сначала проверка позиции в не-IE браузере, потом в IE,
    // и только потом возвращение нуля
    return e.pageY || (e.clientY +
        (document.documentElement.scrollTop || document.body.scrollTop));
}
```

`pageX` / `pageY`

Эти свойства содержат координаты `x` и `y` указателя мыши относительно отображаемого документа (к примеру, если вы прокрутили документ вниз, значения уже не будут соответствовать тем, что содержатся в свойствах `clientX/clientY`). В Internet Explorer эти свойства не работают. Чтобы получить позицию курсора в IE, нужно воспользоваться свойствами `clientX/clientY`, и добавить к ним текущее смещение прокрутки.

`layerX` / `layerY` и `offsetX` / `offsetY`

Эти свойства содержат координаты `x` и `y` указателя мыши относительно элемента назначения события. Свойства `layerX/layerY` доступны только в браузерах, работающих на движке Mozilla и Safari, а свойства `offsetX/offsetY` доступны в Opera и Internet Explorer. (Примеры их использования можно посмотреть в листинге Б.17.)

`button`

Это свойство содержит число, представляющее кнопку мыши, нажатую в данный момент (доступно только в событиях `click`, `mousedown` и `mouseup`). К сожалению, в представлении соответствия чисел нажатой кнопке тоже присутствует некоторая неразбериха. Хорошо, что хоть число 2 используется во всех браузерах для представления щелчка правой кнопкой, и вы можете чувствовать себя вполне свободно хотя бы при проверке

щелчка этой кнопкой. В таблице Б.1 показаны все возможные значения свойства `button` как в Internet Explorer, так и в W3C-совместимых браузерах.

Таблица Б.1. Возможные значения для свойства `button` объекта события

Щелчок	Internet Explorer	W3C
Левой кнопкой	1	0
Правой кнопкой	1	0
Средней кнопкой	4	1

В листинге Б.6 показан фрагмент кода, предотвращающий пользовательский щелчок правой кнопкой мыши (и вызов контекстного меню) в любом месте веб-страницы.

Листинг Б.6. Использование свойства `button` объекта события

```
// Привязка обработчика щелчка ко всему документу
document.onclick = function(e) {
    // Нормализация объекта события
    e = e || window.event;

    // Если произведен щелчок правой кнопкой мыши
    if ( e.button== 2 ) {
        // Предотвращение выполнения исходного действия
        e.preventDefault();
        return false;
    }
};
```

relatedTarget

Это свойство события содержит ссылку на элемент, пространство которого указатель мыши только что покинул. Почти всегда оно используется в ситуациях, когда нужно воспользоваться событиями `mouseover/mouseout`, но вам нужно знать, где указатель только что был, или по какому элементу он проходит. В листинге Б.7 показан вариант древовидного меню (элементы `` содержат другие элементы ``) в котором поддережья отображаются в первый раз только тогда, когда пользователь перемещает элемент над подэлементом ``.

Листинг Б.7. Использование свойства `relatedTarget` для создания дерева, по которому можно перемещаться

```
// Обнаружение всех элементов <li>, содержащихся в документе
var li = document.getElementsByTagName('li');
for ( var i = 0; i < li.length; i++ ) {

    // и прикрепление к ним обработчика события mouseover
    li[i].onmouseover = function(e) {

        // Если указатель мыши входит на элемент в первый раз (от родителя)
        if ( e.relatedTarget == this.parentNode ) {
            // отображение последнего дочернего элемента (который
            // является другим <ol>)
```

```

        this.lastChild.style.display = 'block';
    }
};
}

```

// Пример HTML:

```

<ol>
  <li>Hello <ol>
    <li>Another</li>
    <li>Item</li>
  </ol></li>
  <li>Test <ol>
    <li>More</li>
    <li>Items</li>
  </ol></li>
</ol>

```

Свойства клавиатуры

Обычно свойства клавиатуры существуют в объекте события только при инициации событий, связанных с клавиатурой (среди которых `keydown`, `keyup` и `keypress`). Исключение из этого правила относится к свойствам `ctrlKey` и `shiftKey`, которые доступны во время наступления событий, связанных с мышью (позволяя вам отслеживать `CtrlClick` на элементе). Во всех остальных случаях можно считать, что значения, содержащиеся в свойстве, не существуют или наверняка отсутствуют.

`ctrlKey`

Это свойство возвращает булево значение, отображающее, нажата ли клавиша `Ctrl`. Оно доступно для событий, связанных как с клавиатурой, так и с мышью. Код в листинге Б.8 отслеживает пользовательский щелчок мышью и удержание клавиши `control`; когда это происходит, элемент, на котором произошел щелчок, удаляется из документа.

Листинг Б.8. Применение свойства `ctrlKey` для создания разновидности взаимодействия, использующего щелчок мыши

```

// Привязка обработчика щелчка ко всему документу
document.onclick = function(e){
  // Нормализация объекта события
  e = e || window.event;
  var t = e.target || e.srcElement;

  // Если клавиша control удерживается нажатой во время щелчка,
  if ( e.ctrlKey )
    // удаление узла, на котором произведен щелчок
    t.parentNode.removeChild( t );
};

```

`keyCode`

Это свойство содержит число, соответствующее различным клавишам клавиатуры. Возможность использования конкретных клавиш (таких как `Page Up` и `Home`) может варьироваться, но вообще-то все другие

клавиши работают вполне надежно. В табл. Б.2 дается справка по всем часто используемым клавишам и связанных с ними кодам.

Таблица Б.2. Наиболее часто используемые коды клавиатуры

Клавиша	Код клавиши
Забой (Backspace)	8
Табуляция (Tab)	9
Ввод (Enter)	13
Пробел	32
Стрелка влево	37
Стрелка вверх	38
Стрелка вправо	39
Стрелка вниз	40
0–9	48–57
A–Z	65–90

В листинге Б.9 показан код, необходимый для запуска простого показа слайдов. В этом коде предполагается наличие ряда -элементов внутри единственного элемента или . Каждый элемент может содержать все что угодно (например, изображение). Когда нажимаются клавиши стрелок вправо и влево, пользователю показывается следующий или предыдущий элемент .

Листинг Б.9. Использование свойства keyCode для создания простого показа слайдов

```
// Обнаружение первого элемента <li> на странице
var cur = document.getElementsByTagName('li')[0];

// и обеспечение его видимости
cur.style.display = 'block';

// отслеживание любого нажатия клавиши на странице
document.onkeyup = function(e) {
    // Нормализация объекта события
    e = e || window.event;

    // Если нажаты клавиши левой или правой стрелок
    if ( e.keyCode == 37 || e.keyCode == 39 ) {

        // скрытие текущего отображаемого <li>-элемента
        cur.style.display = 'none';

        // Если нажата клавиша левой стрелки, обнаружение
        // предыдущего <li>-элемента
        // (или циклический переход на самый последний элемент)
        if ( e.keyCode == 37 )
            cur = cur.previousSibling || cur.parentNode.lastChild;
```

```

// Если нажата клавиша правой стрелки, обнаружение следующего
// <li>-элемента, или, если текущий элемент был последним,
// возвращение к первому <li>-элементу
else if ( e.keyCode == 39 )
    cur = cur.nextSibling || cur.parentNode.firstChild;

// показ очередного <li>-элемента последовательности
cur.style.display = 'block';
}
};

```

shiftKey

Это свойство возвращает булево значение, отображающее, нажата ли клавиша Shift. Оно доступно для событий, связанных как с клавиатурой, так и с мышью. Код в листинге Б.8 отслеживает пользовательский щелчок мышью и удержание клавиши Shift; когда это происходит, отображается контекстное меню.

Листинг Б.10. Использование свойства shiftKey для отображения специального меню

```

// Привязка обработчика щелчка ко всему документу
document.onclick = function(e) {
    // Нормализация объекта события
    e = e || window.event;

    // Если клавиша Shift удерживается нажатой при совершении щелчка,
    if ( e.shiftKey )
        // Отображение щелчка
        document.getElementById('menu').style.display = 'block';
};

```

События страницы

Все события страницы работают конкретно с функционированием и состоянием всей страницы. Большинство типов событий имеют дело с загрузкой и выгрузкой страницы (когда пользователь посещает страницу, а затем снова ее покидает).

load

Событие load возникает, когда страница полностью завершает свою загрузку; это событие включает загрузку всех изображений, внешних файлов JavaScript, и внешних файлов CSS. Оно может быть использовано в качестве способа запуска вашего DOM-зависимого кода, но, если нужно ускорить время реакции, можно обратиться к функции domReady(), рассмотренной в главе 6.

В листинге Б.11 показан код, ожидающий загрузки страницы, после чего он привязывает обработчик щелчка к элементу, значение ID которого равен cancel. Как только будет запущен обработчик события, он скроет элемент, ID которого равен main.

Листинг Б.11. Использование события load для ожидания окончания загрузки всей страницы

```

// Ожидание завершения загрузки страницы
window.onload = function() {

```

```
// Обнаружение элемента, значение ID которого равно 'cancel' и привязка
// к нему обработчика щелчка
document.getElementById('cancel').onclick = function(){

    // Затем при щелчке скрыть элемент 'main'
    document.getElementById('main').style.display = 'none';
};
};
```

beforeunload

Это несколько неординарное событие, поскольку оно совершенно не стандартное, но широко поддерживаемое. Оно ведет себя очень похоже на событие unload, но с одной существенной разницей. Если из обработчика события beforeunload будет возвращена строка, она появится в подтверждающем сообщении, спрашивающем пользователей, хотят ли они покинуть текущую страницу. Если ответ будет отрицательным, пользователь останется на текущей странице. Динамические веб-приложения, вроде Gmail, используют его, чтобы уберечь пользователей от потенциальных потерь любых несохраненных данных.

В листинге Б.12 к событию прикрепляется простой обработчик (который всего лишь возвращает строку), объясняющий, почему пользователь не должен покидать текущую страницу, на которой он находится. Браузер отобразит окно подтверждения с полным объяснением, включая составленное вами сообщение.

Листинг Б.12. Использование события beforeunload для удержания пользователей от ухода со страницы

```
// Прикрепление обработчика к beforeunload
window.onbeforeunload = function(){

    // Возвращение объяснения, почему пользователю не нужно покидать
    // страницу.
    return 'Ваши данные не было сохранены.';
};
```

error

Событие error возникает при обнаружении ошибки в коде JavaScript. Оно может послужить для перехвата сообщений об ошибках и их отображения, или для того чтобы успешно справиться с возникшей ошибкой. Обработчик этого события ведет себя не так как все остальные обработчики, вместо того, чтобы передать объект события, он передает сообщение, поясняющее суть возникшей ошибки.

В листинге Б.13 показан самостоятельно разработанный способ обработки и отображения сообщений об ошибках в маркированном списке в отличие от вывода их в традиционную консоль ошибок.

Листинг Б.13. Использование события error для ведения просматриваемого журнала регистрации ошибок

```
// Прикрепление обработчика события error
window.onerror = function( message ){

    // Создание элемента <li>, чтобы сохранить сообщение об ошибке
    var li = document.createElement('li');
```



```

li.innerHTML = message;

// Обнаружение нашего списка ошибок (ID элементов которого имеет
// значение 'errors')
var errors = document.getElementById('errors');

// и добавление нашего сообщения об ошибке к верхней части списка
errors.insertBefore( li, errors.firstChild );
};

```

resize

Событие `resize` возникает, как только пользователь изменяет размер окна браузера. Когда пользователь корректирует размер окна браузера, событие `resize` возникнет только когда процесс завершится, но не в ходе его выполнения.

В листинге Б.14 показан код, отслеживающий случаи уменьшения пользователем размеров окна браузера до определенных пределов, применяя дополнительный класс к элементу `document` (чтобы предоставить лучшее стилевое оформление документа для меньшего по размерам окна).

Листинг Б.14. Использование события `resize` для динамического изменения размера элемента

```

// Отслеживание изменения размеров окна пользователем
window.onresize = function() {
    // Обнаружение элемента document
    // (используемого для определения ширины окна)
    var de = document.documentElement;

    // Определение ширины окна браузера
    // (К сожалению, каждый браузер предпочитает делать это
    // по-своему)
    var w = window.innerWidth || (de && de.clientWidth)
        || document.body.clientWidth;

    // Если окно уменьшилось до определенных пределов
    // добавить класс к элементу document
    de.className = w < 990 ? 'small' : '';
};

```

scroll

Событие `scroll` возникает, когда пользователь перемещает позицию документа в окне браузера. Оно может возникнуть в результате нажатия клавиши (использования клавишей стрелок, Page Up/Down или пробела) или путем использования полосы прокрутки.

unload

Это событие возникает, когда пользователь покидает текущую страницу (возможно, он щелкнул на ссылке, на кнопке возврата или даже закрыл окно браузера). Предотвращение исходного действия для этого события не работает (другим событием, представляющим интерес в этом плане является `beforeunload`).

В листинге Б.15 показан код, привязывающий обработчик к событию unload, показывающий пользователю сообщение, когда тот покидает текущую страницу.

Листинг Б.15. Событие unload

```
// Отслеживание момента, когда пользователь покидает веб-сайт
window.onunload = function(){

    // Отображение сообщения для пользователя с благодарностью за визит
    alert( 'Спасибо, что Вы нас посетили!' );

};
```

События пользовательского интерфейса (UI)

События пользовательского интерфейса относятся к тому, как пользователь взаимодействует непосредственно с браузером или с элементами страницы. Они помогают определить, с какими элементами страница пользователь взаимодействует в настоящий момент, и предоставить этим элементам дополнительное информационное наполнение (например, подсветку или вспомогательные меню).

focus

Событие focus является способом определения, где в настоящий момент находится курсор страницы. По умолчанию он находится внутри всего документа; но по щелчку на ссылке или на элементе ввода формы, или по переходу на них с использованием клавиши табуляции, фокус перемещается на эти элементы. (Пример использования этого события показан в листинге Б.18.)

blur

Событие blur возникает когда пользователь переводит фокус с одного элемента на другой (в пределах содержимого ссылок, элементов ввода или самой страницы). (Пример использования этого события показан в листинге Б.18.)

События мыши

События мыши возникают либо когда пользователь перемещает указатель мыши, либо когда он щелкает одной из ее кнопок.

click

Событие click возникает, когда пользователь нажимает левую кнопку мыши на элементе (см. событие mousedown) и отпускает ее (см. событие mouseup) на том же самом элементе. В листинге Б.16 показан пример использования события click для предотвращения перехода по ссылке, которая ссылается на текущую страницу.

Листинг Б.16. Отключение всех попыток щелчков по ссылкам, указывающим на текущую страницу

```
// Обнаружение всех имеющихся в документе элементов <a>
var a = document.getElementsByTagName('a');
for ( var i = 0; i < a.length; i++ ) {

    // Если ссылка указывает на ту же самую страницу, на которой мы
    // находимся
    if ( a[i].href == window.location.href ) {
```

```

// превращение ее в 'недействующую' по щелчку
a[i].onclick = function(e) {
    return false;
};
}
}

```

dblclick

Событие `dblclick` возникает после того, как пользователь достаточно быстро совершил два щелчка. Временной диапазон двойного щелчка зависит от настроек операционной системы.

mousedown

Событие `mousedown` возникает, когда пользователь нажимает кнопку мыши. В отличие от события `keydown`, это событие возникает при нажатии кнопки лишь один раз. Пример использования этого события показан в листинге Б.17.

mouseup

Событие `mouseup` возникает, когда пользователь освобождает ранее нажатую кнопку мыши. Если кнопка освобождается на том же самом элементе, на котором она была нажата, то в дополнение к этому событию возникает также и событие `click`. Пример использования этого события показан в листинге Б.17.

mousemove

Событие `mousemove` возникает, когда пользователь перемещает указатель мыши по странице хотя бы на один пиксел. Сколько событий `mousemove` произойдет (за полное перемещение указателя мыши) зависит от того, насколько быстро пользователь перемещает мышь, и насколько быстро браузер успевает обновлять положение курсора. В листинге Б.17 показан пример простого перетаскивания.

Листинг Б.17. Элементы со значением класса `draggable` могут быть перетасканы пользователем

```

// Инициализация всех переменных, которые будут использованы
var curDrag, origX, origY;

// Отслеживание каждого нажатия кнопки мыши на элементе
document.onmousedown = function(e) {
    // Нормализация объекта события
    e = fixEvent( e );

    // Перетаскиваются только элементы, имеющий класс 'draggable'
    if ( e.target.className == 'draggable' ) {
        // Текущий перетаскиваемый элемент
        curDrag = e.target;

        // Запоминание начальной позиции указателя мыши и местоположения
        // элемента
        origX = getX( e ) + (parseInt( curDrag.style.left ) || 0);
        origY = getY( e ) + (parseInt( curDrag.style.top ) || 0);
    }
}

```

```

    // отслеживание перемещения мыши или освобождения ее кнопки
    document.onmousemove = dragMove;
    document.onmouseup = dragStop;
}
};
// Отслеживание перемещения мыши
function dragMove(e) {
    // Нормализация объекта события
    e = fixEvent( e );

    // Обеспечение отслеживания нужного элемента
    if ( !curDrag || e.target == curDrag ) return;

    // Установка новой позиции указателя
    curDrag.style.left = (getX(e)) + 'px';
    curDrag.style.top = (getY(e)) + 'px';
}

// Ожидание окончания перетаскивания
function dragStop(e) {
    // Нормализация объекта события
    e = fixEvent( e );

    // Перезапуск всех наших методов отслеживания
    curDrag = document.onmousemove = document.onmouseup = null;
}

// Настройка объекта события для его нормализации
function fixEvent(e) {
    // Превращение всех IE-ориентированных параметров в
    // W3C-подобные
    if (!e) {
        e = window.event;
        e.target = e.srcElement;
        e.layerX = e.offsetX;
        e.layerY = e.offsetY;
    }
    return e;
}

```

mouseover

Событие `mouseover` возникает, когда пользователь перемещает указатель мыши на текущий элемент с другого элемента. Если нужно узнать, с какого элемента пришел пользователь, используется свойство `relatedTarget`. Пример использования этого события показан в листинге Б.18.

mouseout

Событие `mouseout` возникает, когда пользователь перемещает указатель мыши за пределы элемента. Сюда включается перемещение указателя мыши с родительского на дочерний элемент (которое может поначалу

показаться не вполне очевидным). Если нужно узнать, к какому элементу пользователь перемещает указатель, используется свойство `relatedTarget`.

В листинге Б.18 показан пример прикрепления пар событий к элементам, чтобы позволить веб-странице использоваться в режиме управления с помощью клавиатуры (и мыши). Когда пользователь перемещает указатель мыши над ссылкой, или когда он для перехода на нее пользуется клавиатурой, ссылка получает дополнительное цветовое выделение.

Листинг Б.18. Создание эффекта прохождения с использованием событий `mouseover` и `mouseout`

```
// Обнаружение всех <a>-элементов для прикрепления к ним обработчиков
// событий
var a = document.getElementsByTagName('a');
for ( var i = 0; i < a.length; i++ ) {

    // Прикрепление к <a>-элементу обработчиков событий mouseover
    // и focus, которые изменяют фон на синий, когда пользователь
    // либо перемещает над ним указатель мыши, либо переводит на него
    // фокус (используя клавиатуру)
    a[i].onmouseover = a[i].onfocus = function() {
        this.style.backgroundColor = 'blue';
    };

    // Прикрепление к <a>-элементу обработчиков событий mouseout
    // и blur, которые возвращают белый фоновый цвет элемента,
    // когда пользователь уходит со ссылки

    a[i].onmouseout = a[i].onblur = function() {
        this.style.backgroundColor = 'white';
    };
}
```

События клавиатуры

События клавиатуры оперируют с со всеми случаями нажатия клавиш, как внутри, так и снаружи области ввода текста.

keydown / keypress

Событие `keydown` является первым событием, которое возникает при нажатии клавиши. Если пользователь продолжает удерживать клавишу нажатой, событие `keydown` будет возникать снова и снова. Событие `keypress` является общепринятым синонимом события `keydown`, их поведение практически идентично, за одним исключением: если нужно предотвратить исходное действие по нажатию клавиши, это следует делать в отношении события `keypress`. В листинге Б.19 показан пример использования обработчика `keypress`, чтобы предотвратить нажатие определенных клавиш в пределах элемента `<input>`.

Листинг Б.19. Предотвращение отправки формы по нечаянному нажатию клавиши `Enter` из элемента `<input>`

```
// Обнаружение всех имеющихся в документе элементов <input>
var input = document.getElementsByTagName('input');
```

```
for ( var i = 0; i < input.length; i++ ) {

    // Привязка обработчика keypress к элементу <input>
    input[i].onkeypress = function(e) {
        // Предотвращение исходного действия, если нажата клавиша ввода
        return e.keyCode != 13;
    };
}
```

keyup

Событие `keyup` является заключительным возникающим событием клавиатуры (после события `keydown`). В отличие от события `keydown`, это событие возникает при освобождении лишь однажды (поскольку клавишу невозможно освобождать в течение довольно длительного периода времени).

События форма

События формы имеют отношение непосредственно к элементам `<form>`, `<input>`, `<select>`, `<button>` и `<textarea>`, основных элементов HTML-форм.

select

Событие `select` возникает, когда пользователь выделяет различные блоки текста в пределах области ввода, используя мышь. С помощью этого события можно переопределить порядок взаимодействия пользователя с формой. В листинге Б.20 показан пример использования события `select` для предотвращения выделения текста в поле формы.

Листинг Б.20. Предотвращение выделения текста пользователем внутри элемента `<textarea>`

```
// Обнаружение первого элемента <textarea> на странице
var textarea = document.getElementsByTagName('textarea')[0];

// Привязка отслеживателя события select
textarea.onselect = function(){
    // Когда осуществляется новое выделение, предотвращение действия
    return false;
};
```

change

Событие `change` возникает, когда значение элемента ввода (сюда включаются элементы `<select>` и `<textarea>`) изменяется пользователем. Это событие возникает, когда пользователь уже покинул элемент, и он утратил фокус.

Код, показанный в листинге Б.21 способен отслеживать изменения и обновления в элементе, ID которого имеет значение `entryArea` (элемент должен быть типа `<textarea>`), в связанной с ним области предварительного просмотра его содержимого в реальном времени.

Листинг Б.21. Отслеживание события `change` для обновления связанного элемента

```
// Отслеживание любых изменений 'entryArea' (типа <textarea>)
document.getElementById('entryArea').onchange = function(){
```

```

// Как только область будет изменена, обновление
// предварительного просмотра
document.getElementById('preview').innerHTML = this.value;
};

```

submit

Событие `submit` возникает только в формах, и только когда пользователь щелкнет на кнопке отправки — `Submit` (находящейся в пределах формы) или нажмет клавишу `Enter/Return` на одном из элементов ввода. Привязав обработчик `submit` к форме, и не привязывая обработчик `click` к кнопке `Submit`, вы обеспечите перехват всех попыток со стороны пользователя отправить данные формы.

В листинге Б.22 показан код, отслеживающий первую форму на странице, перехватывающий отправку, и отображающий сообщение для пользователя, вместо отправки данных на сервер.

Листинг Б.22. Использование `submit` для осуществления альтернативного действия

```

// Привязка обработчика submit к первой форме документа
document.getElementsByTagName('form')[0].onsubmit = function(e) {

    // Получение имени, введенного пользователем
    var name = document.getElementById('name').value;

    // Установка в элемент <h1> содержимого Привет, Имя! (где Имя — это
    // значение name, введенного пользователем в форму)
    document.getElementsByTagName('h1')[0].innerHTML =
        'Привет, ' + name + '!';

    // Предотвращение передачи данных формы на сервер
    return false;
};

```

reset

Событие `reset` возникает только если пользователь щелкнул на кнопке `Reset` (Сброс) внутри формы (в отличие от кнопки `Submit`, действие которой может быть продублировано нажатием клавиши `Enter`). В листинге Б.23 показан пример отслеживания сброса данных формы (события `reset`), чтобы предоставить пользователю альтернативное действие.

Листинг Б.23. Созданный наспех способ управления сбросом данных формы

```

// Обнаружение первой формы на странице
var form = document.getElementsByTagName('form')[0];

// Отслеживание щелчка на кнопке reset
form.onreset = function(){

    // Обнаружение всех элементов <input> внутри формы
    var input = form.getElementsByTagName('input');

```

```
// и сброс их значений за счет присвоения пустой строки
for ( var i = 0; i < input.length; i++ )
    input[i].value = '';
};
```


Приложение В Браузеры

Программирование на JavaScript всегда обуславливалось степенью разработки веб-браузеров. Поскольку использование JavaScript за пределами браузерной среды (например, на стороне сервера) до сих пор не вышло за рамки экспериментов, набор функциональных возможностей этого языка имеет ярко выраженную браузерную направленность. Поэтому возможности, доступные в JavaScript, очень тесно связаны с процессами развития браузеров, и с теми свойствами, которые их разработчики (или пользователи) считают наиболее важными.

Современные браузеры

Из всех браузеров, доступных в Интернете, есть лишь несколько, идущих в ногу со временем и поддерживающих самые последние технологии, и только немногие из них расширяют границы своих возможностей. В конечном счете тот круг браузеров, который будет выбран вами для поддержки, скорее всего будет зависеть от состава вашей аудитории и от того, сколько времени вы готовы потратить на то, чтобы создаваемое приложение нормально работало на каждом из этих браузеров.

К слову сказать, теперь, по сравнению с прежними временами, создавать работоспособные JavaScript-приложения стало намного легче. Благодаря процессу стандартизации многих методов (DOM, XMLHttpRequest и т.д.) пройдет еще немного времени, и для поддержки всех современных браузеров уже не придется прикладывать каких-нибудь дополнительных усилий. Но пока это время еще не настало, поэтому сегодня существует ряд наиболее популярных браузеров, и технологий, которые ими поддерживаются.

Internet Explorer

Созданный компанией Microsoft и входящий в состав ее операционных систем (начиная с Windows 95), Internet Explorer на данный момент является наиболее популярным веб-браузером. С годами, как только Microsoft заняла доминирующие позиции на рынке браузеров, его развитие замедлилось.

Недавно, в связи с выпуском новой операционной системы Microsoft Windows Vista (в состав которой вошел Internet Explorer 7), команда разработчиков снова взялась за работу.

Версии 5.5 и 6.0

Версия 5.5 является обновлением версии IE, принадлежащего Windows 98, а версия 6.0 является исходной для Windows XP. Хотя обе эти версии так и не избавились от ошибок и от непоследовательности в своей реализации CSS и DOM, они обладают хорошей функциональностью, которая может быть поддержана любым веб-приложением.

Версия 7

Это самая последняя версия браузера Internet Explorer, доступная для использования в Windows XP и в Windows Vista. Пока ее применение ограничено, но ожидается, что с ростом популярности Vista круг ее пользователей также возрастет.

Хотя в движке JavaScript мало что изменилось, в механизме отображения была исправлена масса ошибок, связанных с CSS. Для JavaScript произошло лишь одно приметное событие, теперь для объектов XMLHttpRequest не требуется использование XMLHttpRequest, и к ним (по умолчанию) имеется прямой доступ.

Mozilla

Родившийся на базе того, что осталось от Netscape, Mozilla представляет собой плоды усилий по разработке популярного браузера с открытым кодом. Благодаря последним выпускам Firefox и ростом его популярности, Mozilla завоевал передовые позиции в современной разработке браузеров.

Firefox 1.0, Netscape 8 и Mozilla 1.7

Все три этих браузера основаны на версии 1.7 движка отображения Gecko. Этот движок обладает полной совместимостью со всеми современными технологиями создания сценариев. Все эти браузеры отличаются хорошей надежностью, стабильностью и удобством в работе.

Firefox 1.5 и 2.0

В самой последней версии движка отображения Gecko (1.8), использованной в последних версиях браузера Firefox, поддерживается ряд новых усовершенствований, которые в ближайшие годы будут играть весьма существенную роль. Сюда включается частичная поддержка SVG 1.1, поддержка элементов `<canvas>`, и поддержка JavaScript 1.6. Все эти свойства рассмотрены в главе 14.

Safari

Safari является результатом попытки компании Apple создать лучший (по сравнению с неприглядным Internet Explorer 5) браузер для OS X. Сначала (в версии 1.0), его движок отображения работал грубовато, создавая проблемы разработчикам при создании полностью поддерживающего его кода. Но с новыми выпусками его движок постоянно совершенствовался.

В наиболее распространенных версиях Safari (1.3 для OS X 10.3, 2.0 для OS X 10.4) устранены наиболее существенные ошибки, и добавлены новые возможности. Теперь в том, что ваше приложение, загруженное в Safari, вполне прилично работает, уже нет ничего необычного.

Кроме того в этих браузерах введена поддержка нового элемента `<canvas>`, весьма полезного свойства высокодинамичных приложений, позволяющего разработчикам рисовать на веб-странице.

Opera

Браузер Opera — ветеран браузерных войн с Netscape и Microsoft. Его популярность взлетела вверх после того как этот норвежский браузер стал совершенно бесплатным (до этого он поддерживался за счет рекламных вставок или продажи лицензии). Наряду с Mozilla, Opera находится в стадии активной разработки и проектирования новой спецификации HTML 5, и реализации составляющих этой спецификации в своих браузерах.

Версия 8.5

Версия 8.5 стала первой, полностью свободной версией браузера Opera, поэтому она используется более широко, чем другие версии. Она надежно поддерживает все современные свойства, хотя разработчики время от времени сталкиваются с тем, что ее реализация CSS отличается от реализации в других браузерах; а вот реализация JavaScript более чем достаточна.

Версия 9.0

В самом последнем обновлении браузера Opera устранен ряд ошибок в движке JavaScript, а также добавлена поддержка нового элемента `<canvas>`.