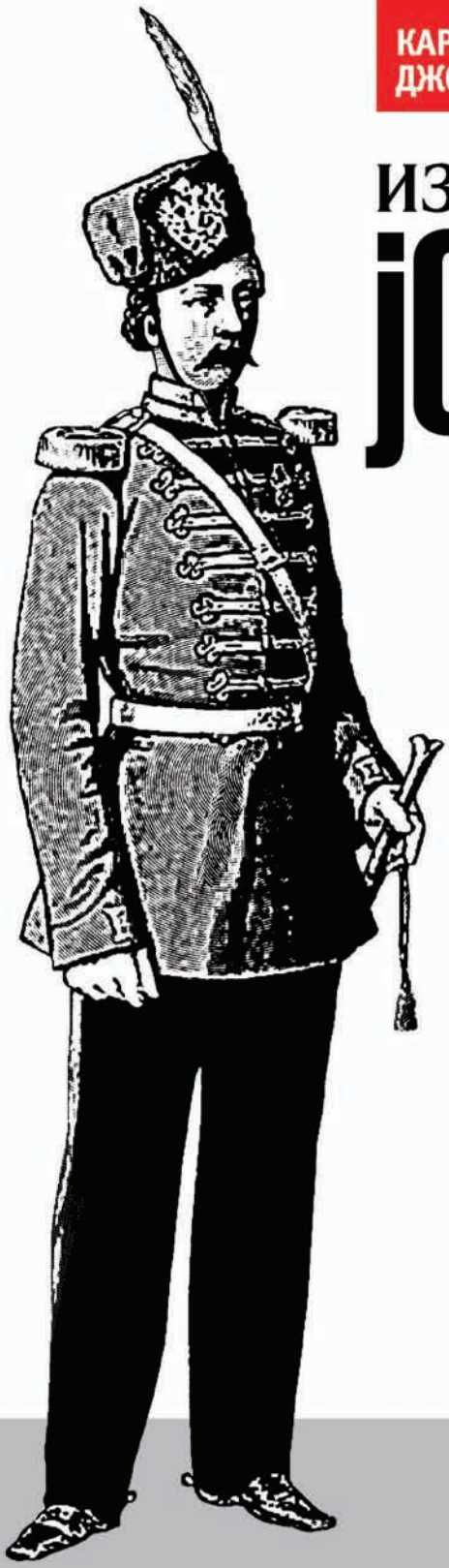
HIGH TECH

КАРЛ ШВЕДБЕРГ
ДЖОНАТАН ЧАФФЕР

ИЗУЧАЕМ jQuery 1.3

ЭФФЕКТИВНАЯ ВЕБ-РАЗРАБОТКА НА JAVASCRIPT



СИМВОЛ®

Learning jQuery 1.3

Better Interaction Design and Web Development with Simple JavaScript Techniques

Jonathan Chaffer, Karl Swedberg

[PACKT] PUBLISHING

HIGH TECH

Изучаем jQuery 1.3

Эффективная веб-разработка на JavaScript

Джонатан Чаффер, Карл Шведберг

Символ®

Санкт-Петербург — Москва
2010

Серия «High tech»

Джонатан Чаффер, Карл Шведберг

Изучаем jQuery 1.3
Эффективная веб-разработка на JavaScript

Перевод А. Киселева

Главный редактор	*А. Галунов*
Зав. редакцией	*Н. Макарова*
Выпускающий редактор	*П. Щеголев*
Редактор	*О. Меркулова*
Корректор	*С. Минин*
Верстка	*К. Чубаров*

Чаффер Дж., Шведберг К.

Изучаем jQuery 1.3. Эффективная веб-разработка на JavaScript. – Пер. с англ. – СПб.: Символ-Плюс, 2010. – 448 с., ил.

ISBN 978-5-93286-177-6

Издание, посвященное jQuery версии 1.3, знакомит с основами использования этой библиотеки для создания привлекательных интерактивных сайтов. jQuery поможет автоматизировать решение типичных задач и упростить решение более сложных. Опытные веб-дизайнеры, немного знакомые с программированием, смогут быстро приступить к использованию jQuery благодаря тому, что она основана на стандартах технологий HTML и CSS. Опытные программисты при изучении библиотеки оценят ее концептуальную целостность.

В книге рассматриваются методы использования селекторов, приемы организации взаимодействий и воспроизведения анимационных эффектов. Показано, как избежать ошибок, связанных с использованием AJAX, событий и расширенных возможностей языка JavaScript.

Издание предназначено для веб-дизайнеров, желающих использовать интерактивные элементы в своих страницах, и разработчикам, стремящимся создавать веб-приложения с более качественным пользовательским интерфейсом. Опыт работы с jQuery и другими библиотеками JavaScript не требуется, однако приветствуются навыки программирования на языке JavaScript, знание его синтаксических конструкций, а также базовые знания о HTML и CSS.

ISBN 978-5-93286-177-6
ISBN 978-1-847196-70-5 (англ)

Издательство «Символ-Плюс». 199034, Санкт-Петербург, 16 линия, 7,
тел. (812) 324-5353, www.symbol.ru. Лицензия ЛП N 000054 от 25.12.98.

Подписано в печать 21.12.2009. Формат 70×100 $^{1}/_{16}$. Печать офсетная.
Объем 28 печ. л. Тираж 1500 экз. Заказ №
Отпечатано с готовых диапозитивов в ГУП «Типография «Наука»
199034, Санкт-Петербург, 9 линия, 12.

Оглавление

Вступительное слово

Я был горд, когда узнал, что Карл Шведберг (Karl Swedberg) и Джонатан Чаффер (Jonathan Chaffer) взяли на себя труд по созданию книги «Изучаем jQuery 1.3». Эта книга во многом определила стандарт, которому стремятся соответствовать другие книги об этой платформе, а также вообще книги о JavaScript. Издание, безусловно, стало одной из наиболее продаваемых книг о JavaScript в немалой степени благодаря своему качеству и вниманию к деталям.

Я был особенно рад, что за написание книги принялись именно Карл и Джонатан, поскольку уже был знаком с ними и знал, что они прекрасно справятся с поставленной задачей. Являясь членом основной команды разработчиков jQuery, я имел возможность достаточно хорошо узнать Карла за последние пару лет, в частности в ходе работы над его книгой. Глядя на конечный результат, можно без сомнения сказать, что его навыки разработчика и бывшего учителя английского языка идеально подошли для выполнения непростого задания.

У меня также была возможность лично встретиться с обоими авторами, что само по себе большая редкость в мире распределенных проектов с открытыми исходными текстами. Как тогда, так и сейчас Карл и Джонатан являются активными членами сообщества jQuery.

Библиотека jQuery используется различными специалистами. Членами сообщества являются дизайнеры, разработчики, люди, как обладающие опытом программирования, так и не имеющие его. В состав команды jQuery входят люди с разными уровнями подготовки, которые своими отзывами способствуют определению дальнейшего пути развития проекта. Тем не менее всех пользователей jQuery объединяет одна общая цель: постараться максимально упростить разработку веб-приложений на JavaScript.

В настоящее время можно смело утверждать, что любой открытый проект ориентирован на поддержку сообщества или на оказание помощи начинающим пользователям. Но для jQuery это не просто пустые слова, это движущая сила проекта. В действительности, в команде проекта jQuery намного больше людей, занимающихся поддержкой сообщества, написанием документации или расширений, чем тех, кто сопровождает базовый программный код. Хотя библиотека сама по себе чрезвычайно важна, сообщество, окружающее этот программный код, может служить критерием, отличающим еле живой, посредственный проект от проекта, наилучшим образом удовлетворяющего все ваши потребности.

Принцип разработки проекта и порядок использования его программного кода принципиально различаются для большинства открытых проектов и большинства библиотек JavaScript. Участники проекта jQuery и его сообщество обладают широчайшими познаниями; мы понимаем, что разрабатываем jQuery для пользователей с разным опытом программирования, поэтому прилагаем максимальные усилия, чтобы передать эти знания всем пользователям.

Чтобы понять, что собой представляет сообщество jQuery, недостаточно просто прочитать о нем; необходимо личное участие в сообществе, чтобы его прочувствовать. Я надеюсь, у вас будет возможность принять в нем участие. Приходите к нам в форумы и блоги, подписывайтесь на рассылки, и мы поможем вам приобрести более глубокие знания об этой библиотеке.

Для меня jQuery означает гораздо больше, чем блок программного кода. Это совокупность опыта, накопленного на протяжении многих лет в процессе разработки библиотеки. Серьезные взлеты и падения, трудности совершенствования и захватывающее ощущение роста и успеха. Профессиональный рост вместе с пользователями и коллегами по команде, улучшение взаимопонимания между нами, усилия по дальнейшему развитию и адаптации проекта.

Когда я впервые увидел эту книгу, рассказывающую о jQuery как о едином инструменте, в противоположность моим собственным представлениям, я был удивлен и восхищен одновременно. Наблюдение за тем, как другие изучают, осмысливают и применяют jQuery к своим задачам, во многом делает этот проект таким захватывающим.

Я не единственный, кто пользуется этой библиотекой на уровне, существенно отличающемся от уровня взаимодействия обычного пользователя с инструментом. Не уверен, что смогу должным образом описать свои ощущения в тот момент, когда я в очередной раз наблюдал, как загораются глаза пользователей от осознания того, как много может предложить им jQuery.

В какую-то секунду в головах пользователей jQuery раздается щелчок, и они начинают понимать, что эта библиотека представляет собой нечто гораздо большее, чем простой инструмент, и неожиданно их взгляды на разработку динамических веб-приложений полностью меняются. Это самая потрясающая и, безусловно, моя самая любимая особенность проекта jQuery.

Я надеюсь, что у вас тоже появится возможность испытать это поразительное ощущение.

Джон Резиг (John Resig),
создатель библиотеки jQuery

Об авторах

Джонатан Чаффер (Jonathan Chaffer) – технический директор интерактивного подразделения в рекламном агентстве, находящемся в городе Гранд-Рапидс, штат Мичиган. Он руководит разработкой веб-проектов, использующих разнообразные технологии, а также сам продолжает заниматься повседневным программированием.

Чаффер принимал весьма активное участие в разработке открытого проекта Drupal CMS, где в качестве предпочтительной платформы JavaScript предлагалась библиотека jQuery. Он является автором популярного модуля Content Construction Kit, используемого для управления структурированным содержимым сайтов, построенных на основе системы Drupal. Джонатан отвечает за полный пересмотр системы меню Drupal и справочного руководства разработчика по прикладному интерфейсу системы.

Карл Шведберг (Karl Swedberg) – веб-программист в компании Fusionary Media в городе Гранд-Рапидс, штат Мичиган; большую часть времени он занимается веб-дизайном с упором на «веб-стандарты» – применение HTML-кода совместно с продуманными таблицами стилей CSS и «ненавязчивым» JavaScript. Будучи членом команды проектирования jQuery и активным участником обсуждений jQuery, Карл посещает симпозиумы и конференции и проводит корпоративное обучение в Европе и Северной Америке.

Прежде чем Карл занялся разработкой веб-приложений, он был литературным редактором, учителем английского языка в средней школе, владельцем кафе. Его увлечение программированием началось в начале 1990-х, когда он работал в корпорации Microsoft в городе Редмонд, штат Вашингтон, и не угасает по сей день.

Благодарности

Я хотел бы поблагодарить Дженни за ее неугасимый энтузиазм и поддержку, Карла – за стимул продолжать работу над книгой, когда дух ослабевал, а также сообщество Ars Technica – за постоянное вдохновение к техническому совершенствованию.

Джонатан Чаффер

Я благодарен моей супруге Саре за ее неустанную любовь и поддержку. Спасибо также моим милым деткам, Бенджамину и Люсии. Выражаю Джонатану Чафферу свое глубочайшее восхищение его опытом программирования и благодарность за его готовность написать эту книгу вместе со мной.

Большое спасибо Джону Резигу за создание лучшей в мире библиотеки JavaScript и за взращивание вокруг нее такого удивительного сообщества. Спасибо также сотрудникам издательства Packt Publishing, техническим редакторам этой книги, коллективу jQuery Cabal и всем тем, кто помогал и вдохновлял нас на всем протяжении работы над книгой.

Карл Шведберг

О технических редакторах

Акаш Мехта (Akash Mehta) – веб-программист, технический писатель и бизнес-консультант, живущий в городе Брисбен, Австралия. В число его разработок входят веб-сайты брошюрного типа, системы электронного обучения и информационные комплексы. Он пишет статьи о разработке веб-приложений для издательств, выпускающих продукцию как в печатном, так и в электронном виде, регулярно выступает на местных конференциях и оказывает поддержку нескольким видным блогам PHP.

Будучи студентом, Акаш занимался поддержкой веб-приложений на языке PHP и созданием пользовательских интерфейсов с помощью инструментария jQuery. Наряду с деятельностью, нацеленной на получение ученой степени в сферах коммерции и информационных технологий, Акаш продолжает разрабатывать веб-приложения на языках PHP и Python. Во внерабочее время он руководит организацией своей местной группы пользователей PHP.

При создании веб-приложений Акаш использует различные библиотеки, распространяемые с открытыми исходными текстами. Его арсенал содержит несколько платформ разработки приложений, включая Zend Framework, CakePHP и Django; платформы JavaScript, такие как jQuery, Prototype и Mootools; другие платформы, такие как Adobe Flash/Flex, и механизмы баз данных MySQL и SQLite.

В настоящее время Акаш предоставляет услуги по созданию документации и разработке веб-приложений на своем веб-сайте *http://bitmeta.org*.

Дейв Метвин (Dave Methvin) обладает более чем 25-летним опытом разработки программного обеспечения для операционных систем Windows и UNIX. В начале своей карьеры он занимался созданием встроенного программного обеспечения в области робототехники, телекоммуникаций и медицины. Позднее он переориентировался на разработку программного обеспечения для персональных компьютеров с использованием языка C/C++ и веб-технологий.

Дейв имеет также 20-летний опыт работы в компьютерной журналистике. Он был ответственным секретарем в журналах «PC Tech Journal» и «Windows Magazine», где занимался освещением вопросов, связанных с персональными компьютерами и Интернетом; его колонки о JavaScript содержали ценные решения типичных задач

веб-программирования. Помимо этого он является соавтором книги «Networking Windows NT» (John Wiley & Sons, 1997).

В настоящее время Дейв занимает должность главного технического директора веб-сайта PC Pitstop, помогающего пользователям оптимизировать производительность их компьютеров. Он также принимает активное участие в жизни сообщества jQuery.

Майк Алсуп (Mike Alsup) примкнул к проекту jQuery почти с самого начала и передал сообществу множество пользующихся популярностью модулей расширения. Он является активным участником группы jQuery Google Group, в рамках которой часто оказывает помощь начинающим пользователям jQuery.

Майк живет в штате Нью-Йорк, работает ведущим программистом в компании Click Commerce, Inc., где занимается разработкой веб-приложений и основное внимание уделяет языку Java и библиотеке Swing.

Созданные им модули расширения для jQuery можно найти по адресу: *http://jquery.malsup.com/*.

Предисловие

Проект был запущен в 2005 году Джоном Резигом, вундеркиндом JavaScript, который в настоящее время работает в корпорации Mozilla. Вдохновленный достижениями пионеров в этой области, таких как Дин Эдвардс (Dean Edwards) и Саймон Уиллисон (Simon Willison), Резиг создал библиотеку функций, упрощающих поиск элементов вебстраниц и реализацию их поведения. К тому моменту, когда он во всеуслышание объявил о своем проекте в январе 2006 года, им были добавлены в библиотеку функции, позволяющие манипулировать деревом DOM и воспроизводить простые анимационные эффекты. Резиг дал своему проекту название jQuery, чтобы подчеркнуть главенствующее положение функций поиска, или «запросов» (querying), элементов вебстраниц для последующего воздействия на них из программного кода на JavaScript. Спустя несколько лет были существенно расширены возможности библиотеки jQuery, улучшена ее производительность, и она нашла применение на некоторых наиболее популярных сайтах в Интернете. Хотя Резиг и остается ведущим разработчиком, jQuery продолжает развиваться как настоящий свободный проект и может сегодня похвастаться командой первоклассных разработчиков на JavaScript, а также энергичным сообществом, насчитывающим тысячи программистов.

Библиотека jQuery способна повысить качество ваших веб-сайтов независимо от вашего уровня подготовки. Она предоставляет широкий круг возможностей, имеет простой синтаксис и обеспечивает устойчивую совместимость с различными платформами, размещаясь при этом в одном компактном файле. Более того, существуют сотни модулей, расширяющих функциональность библиотеки jQuery, что делает ее весьма ценным инструментом для решения практически любых задач на стороне клиента.

Книга «Изучаем jQuery 1.3» представляет собой постепенное и пошаговое введение в концепции jQuery, позволяющее вам повысить интерактивность ваших страниц и добавить в них анимационные эффекты, даже если ваши предыдущие попытки использовать JavaScript были неудачными. Эта книга поможет вам избежать ошибок, связанных с использованием технологии AJAX, событиями, эффектами и дополнительными особенностями языка JavaScript. Кроме того, она содержит краткий справочник по библиотеке jQuery, к которому можно будет возвращаться снова и снова.

О чем рассказывается в этой книге

В *главе 1* вы начнете свое знакомство с библиотекой jQuery. Здесь дается описание jQuery и ее возможностей, рассказывается, как загрузить и установить библиотеку, а также демонстрируется первый сценарий.

В *главе 2* вы познакомитесь с выражениями-селекторами jQuery и методами обхода дерева DOM, выполняющими поиск элементов страницы независимо от их местоположения. В этой главе вы узнаете, как с помощью jQuery можно оказывать влияние на стили отображения различных типов элементов, причем в некоторых случаях способом, недоступным при использовании каскадных таблиц стилей CSS.

В *главе 3* вы научитесь использовать механизм jQuery обработки событий для реализации реакции элементов в ответ на события, возникающие в броузере. Вы увидите, насколько просто с помощью jQuery производится подключение обработчиков событий к элементам, даже когда страница еще не загружена полностью. Вашему вниманию будут представлены такие сложные темы, как всплытие событий, делегирование и пространства имен.

В *главе 4* вы познакомитесь с приемами воспроизведения анимационных эффектов средствами jQuery и увидите, как скрывать, отображать и перемещать элементы страниц с применением эффектов, полезных и приятных для глаз.

В *главе 5* вы узнаете, как изменять страницу по команде. Эта глава научит вас «на лету» изменять структуру документа HTML и его содержимое.

В *главе 6* вы познакомитесь с разнообразными предоставляемыми библиотекой jQuery способами, упрощающими доступ к функциональным возможностям на стороне сервера и не прибегающими при этом к полному обновлению страницы.

В следующих трех главах (7, 8 и 9) будет представлено несколько реальных примеров, использующих все, о чем рассказывалось в предыдущих главах, и обеспечивающих надежные решения часто встречающихся проблем.

В *главе 7* «Манипулирование таблицами» вы научитесь выполнять сортировку, отбор и стилизацию информации для создания красивого и функционального представления данных.

В *главе 8* «Интерактивные формы» вы овладеете тонкостями проверки данных на стороне клиента, создания адаптивных схем размещения элементов форм и приемов реализации интерактивных особенностей поведения клиент-серверных форм, таких как функция автодополнения.

В *главе 9* «Прокрутка и перемещение» вы узнаете, как дополнить страницу возможностью управления отображением ее элементов. Вы на-

учитесь перемещать информацию за и в пределы области видимости исходя из логики работы программы или по требованию пользователя.

В главах 10 и 11 вы познакомитесь с модулями расширений к библиотеке, написанными сторонними разработчиками, и узнаете о различных способах создания своих собственных расширений.

В *главе 10* «Использование модулей расширения» вы займетесь исследованием модуля Form и официальной коллекции расширений элементов пользовательского интерфейса, известной как jQuery UI. Кроме того, здесь вы узнаете, где можно найти массу других популярных расширений для jQuery и посмотреть, какие особенности они могут предложить.

В *главе 11* «Разработка модулей расширения» вы узнаете, как использовать внушительные возможности библиотеки jQuery для создания с нуля собственных модулей расширения. Здесь вы научитесь создавать собственные вспомогательные функции, добавлять методы к объектам jQuery, определять свои селекторы и многое другое.

Приложение A «Ресурсы в Интернете» содержит перечень информационных веб-сайтов, где обсуждается широкий круг тем, имеющих отношение к jQuery, JavaScript и к разработке веб-приложений вообще.

В *приложении B* «Инструменты разработчика» вы познакомитесь с дополнительными программными продуктами, предназначенными для редактирования и отладки программного кода, использующего библиотеку jQuery, в вашей личной среде разработки.

В *приложении C* «Замыкания в JavaScript» вы получите четкое представление о замыканиях – что это такое и как их использовать.

В *приложении D* «Краткий справочник» кратко представлена вся библиотека jQuery со всеми ее методами и селекторами. Справочник имеет удобный формат, который позволяет легко отыскивать необходимую информацию о методе или селекторе, когда вы точно знаете, что он существует, но не уверены, как пишется его имя.

Что необходимо для работы с этой книгой

Для самостоятельной проработки примеров программного кода, демонстрируемых в этой книге, необходимо следующее:

- Простой текстовый редактор.
- Современный веб-броузер, такой как Mozilla Firefox, Apple Safari или Microsoft Internet Explorer.
- Исходный файл библиотеки jQuery версии 1.3.1 или выше, который можно загрузить с сайта *http://jquery.com/*.

Кроме того, для запуска примеров, использующих технологию AJAX, из главы 6 потребуется сервер с поддержкой PHP.

Для кого предназначена эта книга

Эта книга предназначена для веб-дизайнеров, желающих использовать интерактивные элементы в своих страницах, и для разработчиков, стремящихся создавать веб-приложения с более качественным пользовательским интерфейсом. Наличие базовых навыков программирования на языке JavaScript является обязательным условием. Вы должны обладать базовыми знаниями о языке разметки HTML и таблицах CSS и уверенно разбираться в синтаксических конструкциях JavaScript. Знание библиотеки jQuery и опыт работы с любыми другими библиотеками на JavaScript не требуются.

Типографские соглашения

В этой книге вам встретятся различные способы оформления текста, используемые для выделения информации различного типа. Ниже даются несколько примеров таких способов оформления и пояснения к ним.

Элементы программного кода в тексте оформляются следующим образом: «Подключение других контекстов может осуществляться с помощью директивы `include`».

Блоки программного кода оформляются, как показано ниже:

```
<html>
  <head>
    <title>the title</title>
  </head>
  <body>
    <div>
      <p>This is a paragraph.</p>
      <p>This is another paragraph.</p>
      <p>This is yet another paragraph.</p>
    </div>
  </body>
</html>
```

Чтобы привлечь внимание к некоторому фрагменту в блоке программного кода, соответствующие строки или элементы будут выделяться жирным шрифтом:

```
$(document).ready(function() {
  $('a[href^=mailto:]').addClass('mailto');
  $('a[href$=.pdf]').addClass('pdflink');
  $('a[href^=http][href*=henry]')
    .addClass('henrylink');
});
```

Данные ввода-вывода в командной строке оформляются, как показано ниже:

```
outerFn():
Outer function
Inner function
```

Новые термины и **важные слова** выделены жирным шрифтом. Элементы интерфейса, которые отображаются на экране, например в меню или в диалогах, выделяются в тексте рубленым шрифтом.

Примечание

Так выделяются предупреждения или важные примечания.

Совет

Так выделяются советы и рекомендации.

Обратная связь с читателями

Мы всегда приветствуем любые отзывы наших читателей. Дайте нам знать, что вы думаете об этой книге, что вам понравилось или, наоборот, не понравилось. Мнения читателей очень важны для нас, так как они помогают нам создавать книги, от которых вы сможете получать максимальную отдачу.

Чтобы передать нам свой отзыв, просто отправьте его на адрес электронной почты *feedback@packtpub.com*, не забудьте при этом указать название книги.

Если вы хотите, чтобы мы опубликовали какую-либо книгу, заполните форму SUGGEST A TITLE на странице *www.packtpub.com* или отправьте электронное письмо по адресу: *suggest@packtpub.com*.

Если вы обладаете серьезными познаниями в какой-либо области и хотели бы написать или предложить нам уже написанную вами книгу, обращайтесь на страницу с правилами оформления для авторов по адресу: *www.packtpub.com/authors*.

Поддержка покупателей

Теперь, когда вы являетесь владельцем книги, выпущенной издательством Packt, мы можем предложить еще кое-что, что позволит извлечь максимум из вашей покупки.

Программный код примеров из книги

По адресу *http://www.packtpub.com/files/code/6705_Code.zip* вы можете загрузить программный код примеров.

Загружаемые файлы содержат инструкции по их использованию.

Опечатки

Хотя мы приняли все меры для обеспечения точности содержимого книги, ошибки не исключаются полностью. Если вы обнаружите ошибку в одной из наших книг, будь то ошибка в тексте или в программном коде, мы будем весьма признательны, если вы сообщите нам об этом. Тем самым вы сможете уберечь других читателей от разочарования и помочь улучшить последующие издания этой книги. В случае обнаружения опечаток сообщите о них, посетив страницу *http://www.packtpub.com/support*, где следует выбрать нужную книгу, щелкнуть на ссылке let us know и подробно описать найденную опечатку. После проверки вашего сообщения, оно будет принято, и замеченная вами опечатка будет добавлена в список существующих опечаток. Список существующих опечаток можно просмотреть, выбрав название книги на странице *http://www.packtpub.com/support*.

Пиратство

Пиратское распространение авторских материалов в Интернете независимо от типа носителя по-прежнему остается насущной проблемой. Издательство Packt очень серьезно относится к защите своих авторских прав и лицензий. Если в Интернете вам встретятся нелегальные копии наших произведений в любом виде, пожалуйста, сразу же сообщите местоположение копии или название веб-сайта, чтобы мы могли предпринять необходимые меры.

Сообщения о подозрительных копиях вы можете направлять по адресу *copyright@packtpub.com*.

Мы будем вам благодарны за помощь в защите наших авторов и нашей возможности издавать для вас ценные книги.

Вопросы

В случае появления проблем, связанных с какими-либо аспектами книги, вы можете направлять свои вопросы по адресу *questions@packtpub.com*, и мы приложим все усилия, чтобы разобраться с ними.

1

Введение в jQuery

В наши дни Всемирная паутина представляет собой динамичную среду, и ее пользователи предъявляют высокие требования как к оформлению, так и к функциональности сайтов. Для создания интересных интерактивных сайтов разработчики используют библиотеки JavaScript, такие как jQuery, чтобы автоматизировать решение наиболее типичных задач и упростить решение более сложных. Одной из причин высокой популярности библиотеки jQuery является ее способность помогать при решении весьма широкого круга задач.

На первый взгляд кажется, что сложно выбрать, с чего начать, потому что библиотека jQuery реализует весьма широкие функциональные возможности. Тем не менее она имеет согласованную и симметричную архитектуру; большая часть концепций заимствована из **HTML** и **каскадных таблиц стилей (Cascading Style Sheets, CSS)**. Архитектура библиотеки быстро осваивается дизайнерами даже с малым опытом программирования, поскольку многие веб-разработчики, как правило, имеют бóльший опыт работы с указанными технологиями, чем с JavaScript. Так, в первой главе мы напишем действующую программу, использующую jQuery и содержащую всего три строчки кода. В свою очередь опытным программистам будет помогать и концептуальная целостность библиотеки, в чем мы убедимся в последующих более сложных главах.

Итак, посмотрим, что нам может предложить библиотека jQuery.

Что делает библиотека jQuery

Библиотека jQuery предоставляет многоцелевой уровень абстракции для решения типичных задач разработки веб-приложений и потому может применяться практически в любых ситуациях. Библиотека имеет расширяемую архитектуру; так как постоянно появляются новые расширения и добавляются новые возможности, в одной книге просто не

охватить все функции и допустимые случаи использования jQuery. Однако базовые возможности позволяют решать следующие задачи:

- **Доступ к элементам документа**. Чтобы выполнить обход дерева **объектной модели документа (Document Object Model, DOM)** и отыскать определенные фрагменты HTML без применения библиотеки JavaScript, пришлось бы написать уйму строк программного кода. Библиотека jQuery предлагает надежный и эффективный механизм селекторов, который позволяет извлекать требуемые фрагменты документа для последующего анализа и модификации.
- **Изменение внешнего вида страницы**. Каскадные таблицы стилей (CSS) предлагают мощный механизм определения внешнего вида документа, но он оказывается бесполезным в случае веб-броузеров, не поддерживающих единые стандарты. С помощью jQuery разработчики могут восполнить этот недостаток, опираясь на стандарты, поддерживаемые всеми броузерами. Кроме того, библиотека jQuery позволяет изменять классы или отдельные стилевые свойства, применяемые к фрагменту документа, даже после того, как он будет отображен.
- **Изменение содержимого документа**. Библиотека jQuery позволяет не только выполнять простые косметические изменения документа, но и дает возможность модифицировать его содержимое. С помощью одного удобного в использовании **прикладного программного интерфейса (Application Programming Interface, API)** можно изменять текст, вставлять или изменять изображения, переупорядочивать списки и даже полностью изменять и расширять структуру документа HTML.
- **Отклик на действия пользователя**. Даже самые тщательно разработанные и мощные реализации поведения становятся бесполезными, если отсутствует возможность управления моментом, когда они должны запускаться. Библиотека jQuery предлагает элегантный способ, дающий возможность перехватывать самые разнообразные события, такие как щелчок мышью на ссылке, и не загромождать при этом код разметки HTML обработчиками событий. Кроме того, прикладной интерфейс механизма обработки событий ликвидирует существующие между броузерами противоречия, которые часто вызывают чувство досады у веб-разработчиков.
- **Воспроизведение анимационных эффектов в документе**. Для эффективного взаимодействия пользователя с документом дизайнер должен обеспечить обратную визуальную связь. Библиотека jQuery содействует решению этой задачи, предоставляя множество анимационных эффектов, таких как растворение и стирание элементов, а также удобные инструментальные средства для реализации новых эффектов.
- **Извлечение информации со стороны сервера без полного обновления страницы**. Этот шаблон программирования известен как **асинхронный JavaScript и XML (Asynchronous JavaScript And XML,**

AJAX) и помогает веб-разработчикам создавать полнофункциональные и быстро реагирующие сайты. Библиотека jQuery скрывает сложности, связанные с несовместимостью броузеров, позволяя разработчикам сконцентрироваться на реализации функциональности на стороне сервера.

- **Упрощение решения типичных задач программирования на JavaScript.** В дополнение ко всем возможностям, связанным с документами, библиотека jQuery предоставляет расширения к базовым конструкциям JavaScript, таким как обход массивов в цикле и манипулирование ими.

Чем обусловлен успех jQuery

Новый всплеск интереса к динамическому HTML привел к возникновению большого числа платформ JavaScript. Одни из них являются узкоспециализированными инструментами, уделяющими внимание только одной или двум из указанных выше задач. Другие пытаются предложить полный спектр возможностей и анимационных эффектов и представляют собой комплекты всего, что только может потребоваться. Чтобы обеспечить широкий круг возможностей, обозначенных выше, и остаться при этом достаточно компактной, библиотека jQuery использует несколько стратегий:

- **Использование знаний о CSS.** Благодаря тому, что в основу механизма поиска элементов страницы были положены **селекторы CSS**, библиотека jQuery унаследовала краткий и понятный способ выражения структуры документа. Для дизайнеров, стремящихся обеспечить интерактивность своих страниц, библиотека jQuery стала отправной точкой, поскольку знание синтаксиса CSS является необходимым и обязательным условием профессиональной веб-разработки.
- **Поддержка расширений.** Во избежание «расползания» функциональных возможностей библиотека jQuery оставляет реализацию особых случаев за **модулями расширения**. Порядок создания новых модулей прост и подробно описан в документации, что способствует разработке разнообразных изобретательных и полезных модулей. Даже многие базовые функциональные возможности библиотеки jQuery реализованы с использованием механизма расширений и могут удаляться из библиотеки по мере необходимости, что позволяет еще сильнее уменьшать ее размер.
- **Абстрактный способ обхода несовместимостей броузеров.** Суровая действительность веб-разработки такова, что каждый броузер имеет свои отклонения от общепринятых стандартов. Значительную часть любого веб-приложения составляет реализация функциональных особенностей для каждой из платформ. Вследствие невозможности реализовывать некоторые особенности одинаково для всех броузеров, библиотека jQuery добавляет **уровень абстракции**, позволяю-

щий унифицировать решение типичных задач, уменьшить объем программного кода и существенно упростить его.

- **Всегда работает с наборами**. Когда мы предписываем библиотеке jQuery *отыскать все элементы с классом* `collapsable` *и скрыть их*, нам не требуется производить обход всех элементов в цикле, потому что методы, такие как `.hide()`, автоматически работают не с отдельными объектами, а с их наборами. Благодаря этому приему, называемому **неявной итерацией**, многие конструкции цикла становятся ненужными, что приводит к существенному сокращению объема программного кода.
- **Позволяет выполнять множество операций в одной строке**. Во избежание чрезмерного использования временных переменных или ненужных повторов библиотека jQuery для основных своих методов использует шаблон программирования, называемый **сцеплением**. Благодаря этому результатом большинства операций над объектом является сам объект, готовый к выполнению следующей операции.

Применение этих стратегий позволило сохранить малый объем библиотеки (сжатый файл занимает менее 20 Кбайт) и в то же время обеспечить компактность специализированного программного кода, использующего эту библиотеку.

Элегантность библиотеки обеспечивается частично архитектурными решениями, а частично эволюционным процессом ее развития, поддерживаемым энергичным сообществом, возникшим вокруг этого проекта. Пользователи jQuery обсуждают не только разработку модулей расширения, но и улучшения самого ядра библиотеки. В приложении А приводится множество ссылок на ресурсы сообщества, доступные разработчикам jQuery.

Несмотря на значительные усилия, необходимые для разработки столь гибкой и надежной системы, конечный продукт остается бесплатным для любых видов использования. Этот открытый проект распространяется под двойной лицензией: **GNU Public License** (как и многие другие открытые проекты) и **MIT License** (в целях содействия использованию jQuery в проприетарном программном обеспечении).

Хронология развития проекта jQuery

В данной книге рассматриваются функциональные возможности и синтаксис версии **jQuery 1.3.x**, последней на момент написания этих строк. Главная цель библиотеки – обеспечить простой способ поиска элементов веб-страницы и манипулирования ими – остается неизменной в ходе разработки, но некоторые особенности синтаксиса и возможности изменяются от версии к версии. В приведенном далее кратком обзоре хронологии развития проекта описываются наиболее важные изменения, происходившие от версии к версии.

- **Этап общественной разработки**: в августе 2005 года Джон Резиг (John Resig) впервые объявил об усовершенствованной библиотеке «Behaviour», основанной на библиотеке Prototype. Официально новая платформа была выпущена 14 января 2006 года под названием **jQuery**.
- **jQuery 1.0** (август 2006): эта первая стабильная версия библиотеки уже имела надежную поддержку селекторов CSS, технологии AJAX и механизма событий.
- **jQuery 1.1** (январь 2007): в этой версии была произведена существенная модернизация прикладного программного интерфейса (API). Многие редко используемые методы были объединены, что привело к уменьшению числа методов, которые необходимо изучать и документировать.
- **jQuery 1.1.3** (июль 2007): в этой версии была существенно повышена производительность механизма селекторов jQuery. Начиная с этой версии, производительность библиотеки jQuery стала сопоставимой с производительностью родственных библиотек JavaScript, таких как Prototype, Mootools и Dojo.
- **jQuery 1.2** (сентябрь 2007): из этой версии был убран синтаксис **XPath** выбора элементов, так как он стал избыточным при наличии синтаксиса CSS. В этой версии стала более гибкой настройка эффектов, а благодаря добавлению **логики управления событиями в пространствах имен** упростилась разработка расширений.
- **jQuery UI** (сентябрь 2007): этот новый набор расширений был выпущен, чтобы заменить популярный, но устаревающий модуль Interface. В него была включена богатая коллекция готовых виджетов, а также ряд дополнительных инструментов для создания сложных элементов, таких как интерфейсы буксирования элементов мышью (drag-and-drop).
- **jQuery 1.2.6** (май 2008): в состав основной библиотеки были включены функциональные возможности популярного модуля расширения **Dimensions**, созданного Брендоном Аароном (Brandon Aaron).
- **jQuery 1.3** (январь 2009): значительная модернизация механизма селекторов (**Sizzle**) обеспечила гигантский прирост производительности библиотеки. Официально было объявлено о поддержке **делегирования событий**.

Примечание

Примечания к выпуску для более старых версий jQuery можно найти на вебсайте проекта по адресу: *http://docs.jquery.com/History_of_jQuery*.

Наша первая веб-страница, использующая библиотеку jQuery

Теперь, когда мы рассмотрели круг возможностей, доступных в библиотеке jQuery, можно попробовать задействовать ее.

Загрузка jQuery

На **официальном веб-сайте проекта jQuery** (*http://jquery.com/*) всегда можно найти самую последнюю версию библиотеки и самые свежие новости. Для начала нам необходимо получить копию библиотеки jQuery, которую можно загрузить непосредственно на главной странице сайта. Доступными могут быть несколько версий jQuery, но нам как разработчикам сайтов лучше всего подходит самая последняя несжатая версия. При вводе готового сайта в эксплуатацию ее можно заменить сжатой версией.

Никаких особенных действий при установке производить не требуется, достаточно просто поместить библиотеку в доступное место на сервере. Так как JavaScript является интерпретируемым языком программирования, можно не беспокоиться о сборке или компиляции. Чтобы обеспечить доступ к библиотеке внутри страницы, необходимо просто добавить ссылку на нее в документ HTML.

Подготовка документа HTML

Большинство примеров использования jQuery состоит из трех частей: собственно документ HTML, файлы CSS, где содержатся определения стилей, и файлы JavaScript, выполняющие операции над документом. В первом примере представлена страница с выдержкой из книги, к некоторым частям которой применяются классы CSS.

```
<!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"
    xml:lang="en" lang="en">
  <head>
    <meta http-equiv="Content-Type"
      content="text/html; charset=utf-8"/>

    <title>Through the Looking-Glass</title>

    <link rel="stylesheet" href="alice.css"
      type="text/css" media="screen" />

    <script src="jquery.js" type="text/javascript"></script>
    <script src="alice.js" type="text/javascript"></script>
  </head>
```

```
<body>
  <h1>Through the Looking-Glass</h1>
  <div class="author">by Lewis Carroll</div>

  <div class="chapter" id="chapter-1">
    <h2 class="chapter-title">1. Looking-Glass House</h2>
    <p>There was a book lying near Alice on the table,
      and while she sat watching the White King (for she
      was still a little anxious about him, and had the
      ink all ready to throw over him, in case he fainted
      again), she turned over the leaves, to find some
      part that she could read, <span class="spoken">
      "&mdash;for it's all in some language I don't know,"
      </span> she said to herself.</p>
    <p>It was like this.</p>
    <div class="poem">
      <h3 class="poem-title">YKCOWREBBAJ</h3>
      <div class="poem-stanza">
        <div>sevot yhtils eht dna ,gillirb sawT'</div>
        <div>;ebaw eht ni elbmig dna eryg diD</div>
        <div>,sevogorob eht erew ysmim llA</div>
        <div>.ebargtuo shtar emom eht dnA</div>
      </div>
    </div>
    <p>She puzzled over this for some time, but at last
      a bright thought struck her. <span class="spoken">
      "Why, it's a Looking-glass book, of course! And if
      I hold it up to a glass, the words will all go the
      right way again."</span></p>
    <p>This was the poem that Alice read.</p>
    <div class="poem">
      <h3 class="poem-title">JABBERWOCKY</h3>
      <div class="poem-stanza">
        <div>'Twas brillig, and the slithy toves</div>
        <div>Did gyre and gimble in the wabe;</div>
        <div>All mimsy were the borogoves,</div>
        <div>And the mome raths outgrabe.</div>
      </div>
    </div>
  </div>
</body>
</html>
```

Примечание

Фактическое местоположение файлов на сервере не имеет значения. Просто ссылки из одного файла на другой должны производиться в соответствии с выбранной схемой их размещения. В большинстве примеров в этой книге используются не абсолютные пути к файлам (`/images/foo.png`), а относительные (`../images/foo.png`). Это позволяет опробовать примеры на локальном компьютере без использования веб-сервера.

После обычной преамбулы HTML загружается таблица стилей. В данном примере используется следующая очень простая таблица стилей.

```
body {
  font: 62.5% Arial, Verdana, sans-serif;
}
h1 {
  font-size: 2.5em;
  margin-bottom: 0;
}
h2 {
  font-size: 1.3em;
  margin-bottom: .5em;
}
h3 {
  font-size: 1.1em;
  margin-bottom: 0;
}
.poem {
  margin: 0 2em;
}
.highlight {
  font-style: italic;
  border: 1px solid #888;
  padding: 0.5em;
  margin: 0.5em 0;
  background-color: #ffc;
}
```

Вслед за таблицей стилей подключаются файлы JavaScript. Очень важно, чтобы тег `script`, подключающий библиотеку jQuery, находился *перед* тегами, подключающими программный код наших сценариев, в противном случае платформа jQuery окажется недоступной, когда наш программный код попытается использовать ее.

Примечание

В оставшейся части книги будут приводиться только те фрагменты HTML и файлов CSS, которые важны для обсуждения. Полные версии файлов доступны на вспомогательном веб-сайте книги *http://book.learningjquery.com* и на веб-сайте издательства *http://www.packtpub.com/support*.

Теперь у нас имеется страница, которая выглядит так, как показано на рис. 1.1.

Примечание

В этом примере демонстрируется простейший случай использования jQuery. На практике такого рода оформление текста реализуется исключительно средствами CSS.

Рис. 1.1. *Первоначальный документ HTML*

Мы воспользуемся библиотекой jQuery, чтобы изменить оформление текста поэмы.

Подключение jQuery

Наш собственный программный код будет находиться во втором, пока еще пустом файле JavaScript, который подключается к документу HTML с помощью тега `<script src="alice.js" type="text/javascript"></script>`. Для данного примера нам потребуется написать всего три строки программного кода:

```
$(document).ready(function() {
  $('.poem-stanza').addClass('highlight');
});
```

Поиск текста поэмы

Фундаментальной операцией в библиотеке jQuery является выборка части документа. Ее выполняет конструкция `$()`. Обычно она принимает строковый параметр, который может содержать любое выражение селектора CSS. В данном случае нам требуется найти все части документа, к которым применяется класс `poem-staza`, поэтому селектор выглядит очень просто. Однако далее в книге мы будем рассматривать более сложные варианты. Подробнее о поиске различных частей документа будет рассказываться в главе 2.

В действительности функция `$()` является фабричной функцией, создающей **объект jQuery**, представляющий собой основной строительный блок, с которым нам предстоит работать. Объект jQuery объединяет ноль или более элементов DOM и позволяет воздействовать на них различными способами. В данном случае мы изменяем внешнее пред-

ставление выбранных частей страницы, причем выполняем это за счет изменения списка классов, применяемых к тексту поэмы.

Добавление нового класса

Метод `.addClass()` подобно большинству других методов jQuery имеет имя, говорящее само за себя, – он применяет класс CSS к выбранной части страницы. Метод принимает единственный параметр – имя добавляемого класса. Этот метод, как и его противоположность `.removeClass()`, позволяет исследовать работу библиотеки jQuery путем применения различных селекторов, доступных в библиотеке. Рассматриваемый пример просто добавляет класс `highlight`, который в нашей таблице стилей определен как курсивный текст с рамкой.

Обратите внимание, что для добавления класса нам не требуется выполнять обход всех строф поэмы в цикле. Как уже говорилось, в библиотеке jQuery используется прием **неявной итерации** в таких методах, как `.addClass()`, поэтому для изменения всех выбранных частей документа необходимо произвести единственный вызов функции.

Выполнение программного кода

Комбинации `$()` и `.addClass()` оказалось вполне достаточно, чтобы добиться поставленной цели, заключающейся в изменении внешнего вида текста поэмы. Однако если просто вставить эту строку программного кода в заголовок документа, она не даст никакого эффекта. Программный код JavaScript обычно запускается сразу же, как только обнаруживается броузером, то есть во время обработки заголовка, когда обрабатываемая этим кодом разметка HTML еще отсутствует. Нам необходимо отложить выполнение программного кода до момента, когда дерево DOM окажется доступным для использования.

Традиционным механизмом управления моментом запуска программного кода JavaScript являются **обработчики событий**. Многие обработчики служат для реализации реакции на события, инициируемые пользователем, такие как щелчок мышью или нажатие клавиши. Если бы в нашем распоряжении не было jQuery, нам пришлось бы использовать обработчик `onload`, который запускается после того, как отображена вся страница (вместе со всеми изображениями в ней). Чтобы запустить свой программный код по событию `onload`, необходимо было бы поместить его внутрь функции:

```
function highlightPoemStanzas() {
  $('.poem-stanza').addClass('highlight');
}
```

и затем подключить эту функцию как обработчик события, изменив тег `<body>`:

```
<body onload="highlightPoemStanzas();">
```

В этом случае программный код запустился бы по окончании загрузки страницы.

Такой подход имеет свои недостатки. Чтобы добиться требуемого эффекта, мы производим непосредственное изменение кода разметки HTML. Столь тесная связь между структурой и функциональностью приводит к загромождению кода, особенно при необходимости использовать одну и ту же функцию в нескольких страницах или в случае обработки других событий, таких щелчок мышью, в каждом экземпляре некоторого элемента страницы. Добавление новых эффектов может потребовать внесения изменений во многих местах, что увеличивает вероятность ошибки и осложняет параллельную работу дизайнеров и программистов.

Чтобы избежать подобных трудностей, библиотека jQuery предоставляет возможность запланировать однократный вызов функции после загрузки всего дерева DOM документа без ожидания загрузки изображений с помощью конструкции `$(document).ready()`. Для функции, определенной выше, эта конструкция приобретает вид:

```
$(document).ready(highlightPoemStanzas);
```

Данный прием не требует внесения изменений в разметку HTML; подключение функции производится непосредственно в файле JavaScript. Подробнее о том, как реализовывать обработку других **событий**, не влияя на структуру HTML, будет рассказываться в главе 3.

Однако описанная реализация все еще остается немного избыточной, так как функция `highlightPoemStanzas()` определяется лишь для однократного использования. Последнее означает, что для получения незначительной выгоды нам приходится использовать идентификатор в глобальном пространстве имен функций, который мы должны помнить, чтобы не использовать его повторно. В JavaScript, как и в некоторых других языках программирования, имеется возможность обойти такое неэффективное использование пространства имен, которая заключается в применении **анонимных функций** (иногда их называют **лямбда-функциями**). Благодаря анонимным функциям мы можем записать программный код в том виде, в каком он был представлен первоначально:

```
$(document).ready(function() {
  $('.poem-stanza').addClass('highlight');
});
```

С помощью ключевого слова `function`, за которым не следует имя функции, мы определяем функцию именно там, где она необходима, но не раньше. Тем самым мы ликвидируем лишний программный код JavaScript и сокращаем его общий объем до трех строк. Этот способ чрезвычайно удобен при работе с библиотекой jQuery, поскольку многие методы в качестве аргументов принимают функции, редко используемые более одного раза.

Если анонимные функции таким способом определяются в теле других функций, можно создать **замыкание**. Это продвинутая и мощная концепция, однако следует понимать, что широкое использование вложенных определений функций может приводить к непредсказуемым последствиям и чрезмерному расходованию памяти. Тема замыканий подробно рассматривается в Приложении С.

Конечный результат

Теперь, когда весь программный код JavaScript находится на своем месте, страница выглядит, как показано на рис. 1.2.

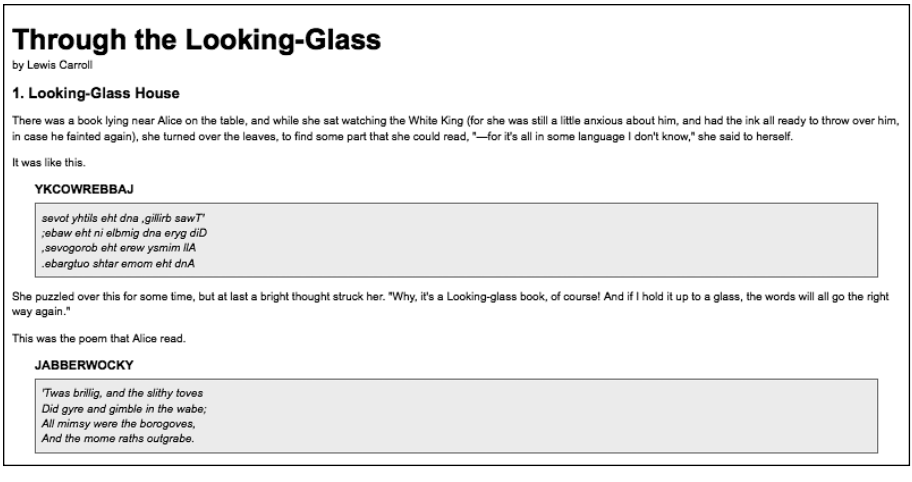


Рис. 1.2. *Страница после применения стилей*

Благодаря тому что программный код JavaScript добавляет класс `highlight`, строфы поэмы выводятся курсивом и заключены в рамки, как определено таблицей стилей `alice.css`.

В заключение

Итак, мы получили представление о том, почему даже для решения простых задач разработчики предпочитают использовать платформу JavaScript, вместо того чтобы писать весь программный код с самого начала. Кроме того, мы познакомились с некоторыми из преимуществ платформы jQuery, с учетом которых мы могли бы выбрать именно ее среди других вариантов. Мы обсудили также в общих чертах, решение каких задач упрощает jQuery.

В этой главе мы узнали, как обеспечить доступ к библиотеке jQuery из программного кода JavaScript в нашей веб-странице, как задействовать фабричную функцию `$()` для поиска элементов страницы, имеющих заданный класс CSS, как использовать метод `.addClass()` для применения

оформления к найденным элементам страницы и как вызывать метод `$(document).ready()` для запуска кода после загрузки страницы.

Для демонстрации работы библиотеки jQuery мы использовали простой, но мало полезный в реальных ситуациях пример. В следующей главе в ходе изучения сложного языка селекторов jQuery мы подробнее остановимся на программном коде и рассмотрим примеры его практического применения.

2

Селекторы

Для получения простого и быстрого доступа к элементам или группам элементов в объектной модели документа (Document Object Model, DOM) библиотека jQuery предоставляет мощные селекторы каскадных таблиц стилей (Cascading Style Sheets, CSS). В этой главе мы исследуем некоторые из этих селекторов, а также **дополнительные селекторы** библиотеки jQuery. Кроме того, мы рассмотрим **методы обхода дерева DOM**, обеспечивающие еще большую гибкость в достижении поставленных целей.

Объектная модель документа

Одной из наиболее сильных сторон jQuery является возможность легко производить выборку элементов дерева DOM. Объектная модель документа – своего рода древовидная структура. В языке HTML, как и в других языках разметки, эта модель применяется для описания взаимоотношений элементов страницы. Говоря о такого рода взаимоотношениях, мы будем использовать ту же терминологию, что и для описания взаимоотношений внутри семьи, – родители, потомки и так далее. Чтобы понять, как метафора генеалогического дерева применяется к документу, рассмотрим простой пример:

```
<html>
  <head>
    <title>the title</title>
  </head>
  <body>
    <div>
      <p>This is a paragraph.</p>
      <p>This is another paragraph.</p>
      <p>This is yet another paragraph.</p>
    </div>
  </body>
</html>
```

Здесь элемент <html> является **предком** для всех остальных элементов; другими словами, все остальные элементы являются **потомками** элемента <html>. Элементы <head> и <body> являются не только потомками, но еще и **дочерними** элементами по отношению к элементу <html>. Точно так же элемент <html> является не только предком для элементов <head> и <body>, но и их **родителем**. Элементы <p> являются дочерними (и потомками) по отношению к элементу , потомками по отношению к элементам <body> и <html> и **братскими** (то есть одного уровня) по отношению друг к другу. В Приложении B рассказывается, как можно визуализировать древовидную структуру DOM с использованием дополнительного программного обеспечения.

Прежде чем продолжать, следует заметить, что множества элементов, отбираемых методами с помощью селекторов, всегда обертываются в объекты jQuery. Эти объекты jQuery очень просты в обращении, когда требуется выполнить какие-то операции над элементами страницы. Мы легко можем привязывать к этим объектам обработчики **событий**, добавлять **эффекты**, а также объединять в **цепочки** несколько операций или эффектов. Однако объекты jQuery отличаются от обычных элементов DOM или узлов списков и не обязательно предоставляют те же методы и свойства. По этой причине в заключительной части этой главы мы рассмотрим способы доступа к элементам DOM, обернутым в объект jQuery.

Фабричная функция $()

Независимо от того, какого типа селектор будет использоваться, вызов любых операций из библиотеки jQuery всегда начинается со знака доллара и круглых скобок: $(). Почти все, что может использоваться в таблицах стилей, точно так же может обертываться в кавычки и помещаться между круглыми скобками, благодаря чему появляется возможность применять методы jQuery для поиска соответствующих элементов.

Примечание

Использование jQuery совместно с другими библиотеками JavaScript

Знак доллара $ в библиотеке jQuery представляет собой обычный «псевдоним» для идентификатора jQuery. При использовании в странице нескольких подобных библиотек может возникнуть конфликт имен, потому что функция $() очень часто встречается в библиотеках JavaScript. Этих конфликтов можно избежать, заменив в нашем программном коде, использующем библиотеку jQuery, каждый экземпляр $ на jQuery. Дополнительные решения этой проблемы рассматриваются в главе 10.

Селекторы составляются из трех основных строительных блоков: **имени тега**, **идентификатора (ID)** и **класса**. Они могут использоваться са-

мостоятельно или в комбинации с другими селекторами. В табл. 2.1 показано, как выглядит каждый из этих трех селекторов.

Таблица 2.1. Селекторы CSS и jQuery

Селектор	CSS	jQuery	Описание
Имя тега	`p`	`$('p')`	Выбирает все параграфы в документе
Идентификатор (ID)	`#some-id`	`$('#some-id')`	Выбирает единственный элемент документа с идентификатором `some-id`
Класс	`.some-class`	`$('.some-class')`	Выбирает все элементы документа, имеющие класс `some-class`

Как уже говорилось в главе 1, когда к фабричной функции `$()` присоединяются методы, они автоматически выполняют неявные итерации по элементам, заключенным в объект jQuery. Благодаря этому мы можем избежать **явного выполнения итераций**, например конструкции цикла `for`, которая часто бывает необходима при работе с деревом DOM.

Теперь, когда мы рассмотрели основы, можно приступать к исследованию более сложных случаев использования селекторов.

Селекторы CSS

Библиотека jQuery поддерживает практически все селекторы, включенные в спецификации CSS с 1 по 3, с которыми можно ознакомиться на сайте консорциума *World Wide Web Consortium*: *http://www.w3.org/Style/CSS/#specs*. Этот факт позволяет разработчикам развивать свои веб-сайты, не беспокоясь о том, что какие-то броузеры (в частности, Internet Explorer 6) могут не понимать некоторые селекторы при условии, что в броузеры включена поддержка JavaScript.

Примечание

Опытные разработчики, использующие jQuery, всегда должны применять в своем программном коде концепции **прогрессивного улучшения** (progressive enhancement) и **постепенной деградации** (graceful degradation), чтобы обеспечить точное отображение страницы, даже если с отключенной поддержкой JavaScript она будет выглядеть не так красиво, как с включенной. Мы будем рассматривать эти концепции на протяжении всей книги.

В ходе изучения принципов работы библиотеки jQuery с селекторами CSS мы будем использовать структуру, присутствующую на многих

веб-сайтах и часто применяемую для навигации, – многоуровневый неупорядоченный список.

```
<ul id="selected-plays">
  <li>Comedies
    <ul>
      <li><a href="/asyoulikeit/">As You Like It</a></li>
      <li>All's Well That Ends Well</li>
      <li>A Midsummer Night's Dream</li>
      <li>Twelfth Night</li>
    </ul>
  </li>
  <li>Tragedies
    <ul>
      <li><a href="hamlet.pdf">Hamlet</a></li>
      <li>Macbeth</li>
      <li>Romeo and Juliet</li>
    </ul>
  </li>
  <li>Histories
    <ul>
      <li>Henry IV (<a href="mailto:henryiv@king.co.uk">email</a>)
        <ul>
          <li>Part I</li>
          <li>Part II</li>
        </ul>
      <li><a href="http://www.shakespeare.co.uk/henryv.htm"> Henry V</a></li>

      <li>Richard II</li>
    </ul>
  </li>
</ul>
```

Обратите внимание, что первый тег `<ul>` имеет идентификатор `selected-plays`, но ни один из тегов `<li>` не имеет класса CSS. Без применения стилей этот список выглядит, как показано на рис. 2.1:

Многоуровневый список отображается в точности так, как и следовало ожидать, – маркированные элементы расположены вертикально и выровнены в соответствии с их уровнями.

Оформление уровней списка

Допустим, нам необходимо, чтобы элементы верхнего уровня, причем *только* эти элементы, были расположены по горизонтали. Для начала мы можем определить в таблице стилей класс `horizontal`:

```
.horizontal {
  float: left;
  list-style: none;
  margin: 10px;
}
```

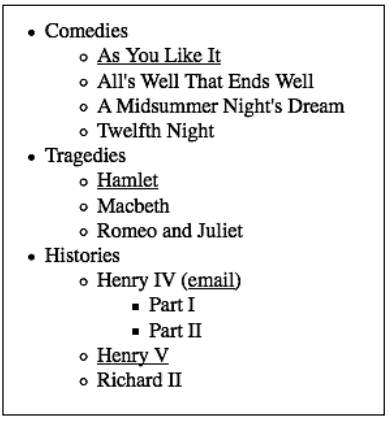


Рис. 2.1. *Внешний вид списка до применения стилей*

Класс `horizontal` располагает элемент левее элемента, следующего за ним, удаляет маркер элемента списка и добавляет кайму шириной 10 пикселов с обеих сторон элемента.

Чтобы продемонстрировать использование селекторов jQuery, мы не будем использовать класс `horizontal` непосредственно в коде разметки HTML, а динамически добавим его ко всем элементам верхнего уровня – **Comedies**, **Tragedies** и **Histories**:

```
$(document).ready(function() {
  $('#selected-plays > li').addClass('horizontal');
});
```

Чтобы запустить программный код после того, как дерево DOM будет загружено, мы используем обертку `$(document).ready()`, о которой говорилось в главе 1.

Для добавления класса `horizontal` лишь ко всем элементам верхнего уровня во второй строке используется **комбинатор дочерних элементов (>)**. В результате селектор внутри функции `$()` предписывает *отыскать все элементы списка (*`li`*), которые являются дочерними (>) по отношению к элементу с идентификатором* `selected-plays` *(*`#selected-plays`*).*

После добавления класса наш многоуровневый список выглядит так, как показано на рис. 2.2.

Изменение оформления всех остальных элементов, расположенных *не* на верхнем уровне, может быть выполнено несколькими способами. Так как мы уже задействовали класс `horizontal`, применив его к элементам верхнего уровня, один из способов выбора элементов нижележащих уровней заключается в использовании операции **отрицания псевдокласса** для идентификации всех элементов списка, *не* имеющих класс `horizontal`. Обратите внимание на добавленную третью строку программного кода:

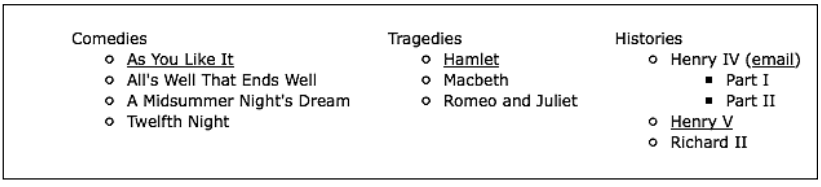


Рис. 2.2. *Внешний вид списка после добавления класса horizontal*

```
$(document).ready(function() {
  $('#selected-plays > li').addClass('horizontal');
  $('#selected-plays li:not(.horizontal)').addClass('sub-level');
});
```

На этот раз мы выбираем все элементы списка (`li`), которые:

1. Являются потомками элемента с идентификатором `selected-plays` (`#selected-plays`)
2. Не имеют класс `horizontal` (`:not(.horizontal)`)

После добавления к этим элементам класса `sub-level` они будут отображаться на сером фоне, как определено в таблице стилей. Теперь вложенный список выглядит, как показано на рис. 2.3.

Селекторы атрибутов

Селекторы атрибутов – это особенно полезное подмножество селекторов CSS. Они позволяют отбирать элементы по одному из свойств HTML, таких как атрибут `title` в ссылках или атрибут `alt` в изображениях. Например, чтобы выбрать все изображения, для которых определен атрибут `alt`, можно использовать следующую конструкцию:

```
$('img[alt]')
```

Примечание

Версии библиотеки jQuery, предшествовавшие версии 1.2, определяли селекторы атрибутов с помощью синтаксиса языка **XML Path (XPath)** и включали ряд других селекторов XPath. Эти селекторы были удалены из последующих версий ядра библиотеки, однако они по-прежнему доступны в виде расширения: *http://plugins.jquery.com/project/xpath/*.

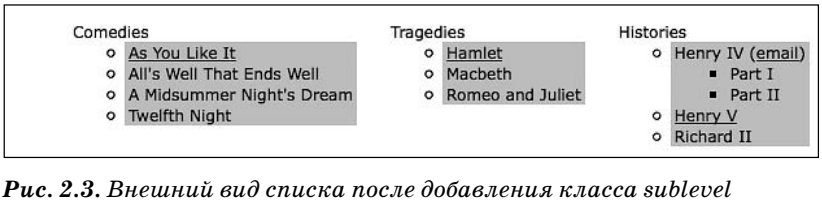


Рис. 2.3. *Внешний вид списка после добавления класса sublevel*

Оформление ссылок

При определении селекторов атрибутов используется синтаксис шаблонов, навеянный регулярными выражениями, для идентификации значения в начале (^) или в конце ($) строки. В селекторах атрибутов могут также использоваться звездочка (*) для указания значения в произвольной позиции внутри строки и восклицательный знак для инверсии значения.

Допустим, что к разным типам ссылок необходимо применить различные стили оформления. Для начала определим стили в таблице стилей:

```
a {
  color: #00c;
}
a.mailto {
  background: url(images/mail.png) no-repeat right top;
  padding-right: 18px;
}
a.pdflink {
  background: url(images/pdf.png) no-repeat right top;
  padding-right: 18px;
}
a.henrylink {
  background-color: #fff;
  padding: 2px;
  border: 1px solid #000;
}
```

Затем средствами библиотеки jQuery добавим три класса, `mailto`, `pdflink` и `henrylink`, к соответствующим ссылкам.

Чтобы добавить класс ко всем ссылкам с адресом электронной почты, мы создаем селектор, который отыскивает все якорные элементы (`a`), в которых атрибут `href` (`[href`) начинается с последовательности символов `mailto:` (`^=mailto:]`), как показано ниже:

```
$(document).ready(function() {
  $('a[href^=mailto:]').addClass('mailto');
});
```

Чтобы добавить класс ко всем ссылкам на файлы PDF, вместо символа крышки мы используем знак доллара. Благодаря этому отбираются ссылки, значение атрибута `href` в которых *оканчивается* последовательностью символов `.pdf`:

```
$(document).ready(function() {
  $('a[href^=mailto:]').addClass('mailto');
  $('a[href$=.pdf]').addClass('pdflink');
});
```

Кроме того, допускается объединять селекторы атрибутов. Например, мы можем добавить класс `henrylink` ко всем ссылкам, значение атрибу-

та href которых начинается с последовательности http и содержит слово henry:

```
$(document).ready(function() {
  $('a[href^=mailto:]').addClass('mailto');
  $('a[href$=.pdf]').addClass('pdflink');
  $('a[href^=http][href*=henry]')
    .addClass('henrylink');
});
```

После применения трех классов к ссылкам трех типов страница будет выглядеть так, как показано на рис. 2.4.

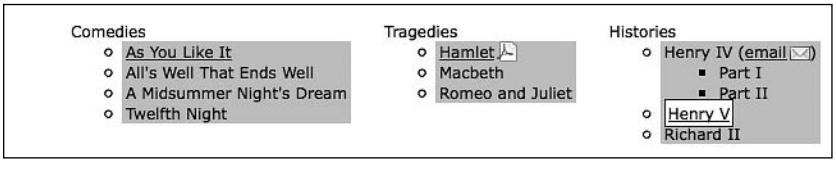


***Рис. 2.4.** Внешний вид списка после добавления классов оформления ссылок*

Обратите внимание на ярлык типа файлов PDF правее ссылки Hamlet, на ярлык с конвертом правее ссылки email и на белый фон и рамку вокруг ссылки Henry V.

Дополнительные селекторы

К широкому кругу селекторов CSS библиотека jQuery добавляет свои собственные селекторы. Большая часть этих дополнительных селекторов позволяет отбирать определенные элементы из списка, если можно так выразиться. При этом используется тот же самый синтаксис, что и в случае **псевдоклассов CSS**, когда селектор начинается с символа двоеточия (:). К примеру, выбрать второй элемент из множества элементов div с классом horizontal можно следующим способом:

```
$('div.horizontal:eq(1)')
```

Обратите внимание, что конструкция :eq(1) отбирает второй элемент из множества, потому что нумерация элементов массивов в языке JavaScript **начинается с 0**. В свою очередь, в таблицах стилей CSS нумерация начинается с единицы, поэтому селектор CSS, такой как $('div:nth-child(1)'), отберет все элементы div, которые являются первыми дочерними элементами (в данном конкретном случае можно было бы использовать конструкцию $('div:first-child')).

Оформление чередующихся строк

В библиотеке jQuery имеются два очень удобных селектора: :odd и :even. Посмотрим, как можно использовать один из них для выделения чередующихся строк в следующей таблице:

```
<table>
  <tr>
    <td>As You Like It</td>
    <td>Comedy</td>
    <td></td>
  </tr>
  <tr>
    <td>All's Well that Ends Well</td>
    <td>Comedy</td>
    <td>1601</td>
  </tr>
  <tr>
    <td>Hamlet</td>
    <td>Tragedy</td>
    <td>1604</td>
  </tr>
  <tr>
    <td>Macbeth</td>
    <td>Tragedy</td>
    <td>1606</td>
  </tr>
  <tr>
    <td>Romeo and Juliet</td>
    <td>Tragedy</td>
    <td>1595</td>
  </tr>
  <tr>
    <td>Henry IV, Part I</td>
    <td>History</td>
    <td>1596</td>
  </tr>
  <tr>
    <td>Henry V</td>
    <td>History</td>
    <td>1599</td>
  </tr>
</table>
```

Теперь добавим стили строк в таблицу стилей и определим класс `alt` **для четных строк:**

```
tr {
    background-color: #fff;
}
.alt {
    background-color: #ccc;
}
```

Наконец, напишем программный код, использующий библиотеку jQuery, который присоединит класс ко всем четным строкам в таблице (теги `<tr>`**):**

```
$(document).ready(function() {
  $('tr:odd').addClass('alt');
});
```

Но минутку! Почему для выбора четных строк используется селектор `:odd` (нечетный)? Дело в том, что селекторы `:odd` и `:even`, как и селектор `:eq()`, используют нумерацию, начинающуюся с 0. Поэтому первая строка получает порядковый номер 0 (четный), вторая – 1 (нечетный) и так далее. Учитывая вышесказанное, можно смело ожидать, что наш простой фрагмент программного кода воспроизведет таблицу, которая будет выглядеть, как показано на рис. 2.5.

As You Like It	Comedy	
All's Well that Ends Well	Comedy	1601
Hamlet	Tragedy	1604
Macbeth	Tragedy	1606
Romeo and Juliet	Tragedy	1595
Henry IV, Part I	History	1596
Henry V	History	1599

Рис. 2.5. *Реализация чередования окраски строк*

Следует отметить, что при наличии на странице нескольких таблиц можно получить неожиданный результат. Например, если последняя строка первой таблицы имеет белый фон, то первая строка следующей таблицы будет иметь «альтернативный» серый фон. Один из способов избежать такой проблемы заключается в использовании селектора `:nth-child()`. Этот селектор может принимать в качестве аргумента число или одно из значений `odd` (нечетный) и `even` (четный).

Заметим, что селектор `:nth-child()` является единственным в библиотеке jQuery, который начинает нумерацию с единицы. Программный код, дающий тот же эффект выделения чередующихся строк, что и выше, и при этом учитывающий возможность присутствия в странице нескольких таблиц, выглядит, как показано ниже:

```
$(document).ready(function() {
  $('tr:nth-child(even)').addClass('alt');
});
```

В заключение обсуждения дополнительных селекторов предположим, что по некоторым причинам необходимо выделить в таблице ячейки, в которых присутствует слово Henry. Все, что необходимо для этого сделать, – добавить в таблицу стилей класс, определяющий начертание текста жирным курсивом (`.highlight {font-weight:bold; font-style: italics;}`), и добавить одну строку программного кода, использующую селектор `:contains()`, в наш сценарий.

```
$(document).ready(function() {
  $('tr:nth-child(even)').addClass('alt');
  $('td:contains(Henry)').addClass('highlight');
});
```

Теперь мы можем увидеть (рис. 2.6) красиво оформленную таблицу с выделенными ячейками, содержащими слово Henry.

As You Like It	Comedy	
All's Well that Ends Well	Comedy	1601
Hamlet	Tragedy	1604
Macbeth	Tragedy	1606
Romeo and Juliet	Tragedy	1595
Henry IV, Part I	History	1596
Henry V	History	1599

Рис. 2.6. *Таблица с отдельным оформлением ячеек, содержащих слово Henry*

Важно отметить, что селектор `:contains()` различает регистр символов. Селектор `$('td:contains(henry)')`, где вместо заглавной буквы «H» используется прописная «h», не выберет ни одной ячейки.

Общеизвестно, что существуют способы выделения чередующихся строк без использования библиотеки jQuery и вообще без создания программного кода на стороне клиента. Однако jQuery вместе с CSS представляет собой прекрасную альтернативу другим средствам оформления подобного рода в тех случаях, когда содержимое генерируется динамически и нет доступа к разметке HTML или к программному коду на стороне сервера.

Селекторы форм

При работе с формами дополнительные селекторы jQuery могут упростить работу по выбору требуемых элементов. В табл. 2.2 описываются некоторые из таких селекторов.

Таблица 2.2. Селекторы форм

Селектор	Соответствуют
`:text`, `:checkbox`, `:radio`, `:image`, `:submit`, `:reset`, `:password`, `:file`	Элементы ввода с атрибутом `type`, совпадающим с именем селектора (за исключением символа двоеточия). Например, селектор `:text` отберет элемент `<input type="text">`
`:input`	Элементы `textarea`, `select` и `button`
`:button`	Элементы `button` и `input` с атрибутом `type`, имеющим значение `button`
`:enabled`	Активные элементы форм

:disabled	Неактивные элементы форм
:checked	Отмеченные радиокнопки или флажки
:selected	Выбранные элементы option

Методы обхода дерева DOM

Селекторы jQuery, изученные нами к настоящему моменту, позволяют отбирать множество элементов таким образом, что мы совершаем обход *поперек* и *вниз* по дереву DOM и фильтруем полученные результаты. Если бы селекторы были единственным способом выбора элементов, наши возможности были бы весьма ограничены (хотя, откровенно говоря, селекторы обеспечивают более высокую надежность по сравнению с обычными средствами, предоставляемыми объектной моделью документа). Во многих случаях бывает необходимо отобрать **родительские** или **дочерние** элементы, и в таких ситуациях удобно использовать методы jQuery обхода дерева DOM. С помощью этих методов мы легко можем передвигаться вверх, вниз и поперек дерева DOM.

Для некоторых методов имеются практически идентичные селекторы. Например, строку, с помощью которой мы ранее добавляли класс alt, `$('tr:odd').addClass('alt');`, можно переписать с использованием метода `.filter()`, как показано ниже:

```
$('tr').filter(':odd').addClass('alt');
```

Эти два способа выбора элементов в значительной степени дополняют друг друга. Кроме того, метод `.filter()`, в частности, обладает огромной мощью, так как способен принимать функцию в качестве аргумента. Это позволяет реализовывать сложные проверки элементов, которые должны включаться в возвращаемый набор. Например, предположим, что необходимо добавить класс ко всем внешним ссылкам. В библиотеке jQuery нет селектора для выполнения подобной операции. Без функции фильтрации нам пришлось бы явно обойти в цикле все элементы и проверить каждый из них в отдельности. С помощью следующей **функции фильтрации** мы по-прежнему можем опереться на механизм неявных итераций jQuery и обеспечить компактность программного кода:

```
$('a').filter(function() {
  return this.hostname && this.hostname != location.hostname;
}).addClass('external');
```

Вторая строка в этом фрагменте фильтрует элементы <a> по следующим двум критериям:

1. Они должны иметь атрибут `href` с доменным именем (`this.hostname`). Эта проверка используется для исключения ссылок типа `mailto` и подобных им.

2. Доменное имя, на которое они ссылаются (снова `this.hostname`), не должно совпадать (`!=`) с доменным именем текущей страницы (`location.hostname`).

Если говорить более точно, метод `.filter()` выполняет обход совпавшего множества элементов и проверяет значение, возвращаемое функцией для каждого из них. Если функция возвращает значение `false`, элемент исключается из совпавшего множества. Если она возвращает значение `true`, элемент остается.

Теперь снова обратимся к нашей таблице, где мы выделяли чередующиеся строки, и посмотрим, какие еще операции можно реализовать с помощью методов обхода.

Изменение оформления отдельных ячеек

Ранее мы добавляли класс `highlight` к ячейкам, содержащим текст **Henry**. Чтобы изменить оформление ячейки, следующей за той, что содержит текст **Henry**, можно использовать тот же селектор, просто присоединив к нему вызов метода `next()`:

```
$(document).ready(function() {
  $('td:contains(Henry)').next().addClass('highlight');
});
```

Теперь таблица должна выглядеть так, как показано на рис. 2.7.

As You Like It	Comedy	
All's Well that Ends Well	Comedy	1601
Hamlet	Tragedy	1604
Macbeth	Tragedy	1606
Romeo and Juliet	Tragedy	1595
Henry IV, Part I	***History***	1596
Henry V	***History***	1599

Рис. 2.7. *Таблица с отдельным оформлением ячеек, следующих за теми, что содержат слово Henry*

Метод `.next()` отбирает только следующий ближайший братский элемент. Чтобы изменить оформление всех ячеек, следующих за той, что содержит текст Henry, можно использовать метод `.nextAll()`.

```
$(document).ready(function() {
  $('td:contains(Henry)').nextAll().addClass('highlight');
});
```

Примечание

Как можно было ожидать, для методов `.next()` и `.nextAll()` в библиотеке имеются противоположные им методы `.prev()` и `.prevAll()`. Кроме того, существует метод `.siblings()`, который отбирает все элементы, расположенные на том же уровне в дереве DOM, что и указанный элемент, независимо от того, находятся ли они перед или после него.

Чтобы включить в возвращаемый набор исходную ячейку (содержащую текст Henry), мы можем добавить вызов метода `.andSelf()`:

```
$(document).ready(function() {
  $('td:contains(Henry)').nextAll().andSelf().addClass('highlight');
});
```

Безусловно, существует масса комбинаций селекторов и методов обхода, с помощью которых можно было бы отобрать тот же набор элементов. Ниже приводится пример еще одного способа выбора всех ячеек в строке, если хотя бы одна из них содержит текст Henry:

```
$(document).ready(function() {
  $('td:contains(Henry)').parent().children().addClass('highlight');
});
```

В данном случае, вместо того чтобы выполнять обход братских элементов, мы с помощью метода `.parent()` перемещаемся по дереву DOM на один уровень вверх до элемента `<tr>` и затем отбираем все ячейки строки с помощью метода `.children()`.

Составление цепочек методов

Комбинирование методов обхода, которое мы только что рассмотрели, иллюстрирует такую особенность библиотеки jQuery, как возможность **составления цепочек** методов. С ее помощью в одной строке программного кода можно отбирать несколько наборов элементов и выполнять множество операций над ними. Возможность составления цепочек не только помогает обеспечить компактность программного кода, но и позволяет увеличить производительность, когда альтернативное решение заключается в повторном использовании селектора.

К тому же возможно разбить одно строку программного кода на несколько строк для большей удобочитаемости. Например, последовательность вызовов методов может быть записана в виде одной строки...

```
$('td:contains(Henry)').parent().find('td:eq(1)')
    .addClass('highlight').end().find('td:eq(2)')
                            .addClass('highlight');
```

...или в виде семи строк...

```
$('td:contains(Henry)') // Найти все ячейки, содержащие текст "Henry"
.parent()               // Выбрать их родительские элементы
.find('td:eq(1)')       // Найти ячейки, являющиеся вторыми потомками
.addClass('highlight')  // Добавить класс "highlight"
.end()                  // Вернуть к родителю ячейки, содержащей текст
"Henry"
.find('td:eq(2)')       // Найти ячейки, являющиеся третьими потомками
.addClass('highlight'); // Добавить класс "highlight"
```

Безусловно, такая реализация обхода дерева DOM находится на грани абсурда. Мы определенно не рекомендуем использовать ее, так как

в нашем распоряжении имеются более простые и более прямые методы. Цель этого примера состоит лишь в том, чтобы продемонстрировать огромную гибкость, предоставляемую возможностью составления цепочек.

Цепочки можно представлять себе, как целые параграфы слов, произносимых на одном дыхании – они работают очень быстро, но иногда их весьма трудно понять. Разбиение на несколько строк и добавление грамотных комментариев в конечном итоге помогут сэкономить больше времени.

Доступ к элементам DOM

Все селекторы и большинство методов jQuery возвращают объект jQuery. Чаще всего это именно то, что нам требуется, так как позволяет пользоваться преимуществами неявной итерации и составлять цепочки методов.

Тем не менее, бывают моменты, когда необходимо получить прямой доступ к **элементу DOM**. Например, нам может потребоваться обеспечить доступность отобранных элементов для другой библиотеки JavaScript или, возможно, обратиться к имени тега элемента, доступному в виде **свойства** элемента DOM. Для таких редких случаев в библиотеке JQuery имеется метод `.get()`. Чтобы получить доступ к первому элементу DOM, на который ссылается объект jQuery, мы можем вызвать метод `.get(0)`. Чтобы обратиться к элементу DOM внутри цикла, можно использовать вызов `.get(index)`. Узнать имя тега элемента с `id="my-element"` можно, как показано ниже:

```
var myTag = $('#my-element').get(0).tagName;
```

Для еще большего удобства библиотека jQuery предоставляет более краткую версию метода `.get()`. Вместо строки, что приводится выше, мы можем записать приведенную ниже, в которой используются квадратные скобки сразу вслед за селектором:

```
var myTag = $('#my-element')[0].tagName;
```

Эта синтаксическая конструкция не случайно выглядит, как массив элементов DOM – квадратные скобки, как напильник, обдирают обертку jQuery, позволяя добраться до списка узлов, при этом **индекс** (в данном случае 0) играет роль пинцета, с помощью которого извлекается требуемый элемент DOM.

В заключение

С помощью приемов, рассмотренных в этой главе, мы теперь сможем извлекать элементы из многоуровневых списков, используя основные **селекторы CSS**; применять различные стили для разных типов ссылок, используя **селекторы атрибутов**; добавлять выделение чередующихся

строк в таблицах, используя либо **дополнительные селекторы jQuery** `:odd` и `:even`, либо усовершенствованный селектор CSS `:nth-child()`; выделять текст в отдельных ячейках таблицы, используя возможность составления цепочек методов jQuery.

До сих пор, чтобы добавить класс к соответствующему набору элементов, мы использовали событие `$(document).ready()`. В следующей главе мы рассмотрим способы добавления класса в ответ на различные события, вызванные действиями пользователя.

3

События

В языке JavaScript предусмотрено несколько встроенных способов реализации реакции на действия пользователя и другие события. Чтобы сделать страницу динамичной и отзывчивой, необходимо задействовать эту возможность, тогда в соответствующее время мы могли бы применить предоставляемые библиотекой jQuery средства, как уже рассмотренные к настоящему моменту, так и те, с которыми нам еще предстоит познакомиться. Поставленную задачу можно выполнить и с помощью чистого JavaScript, однако библиотека jQuery расширяет и дополняет базовые механизмы обработки событий, предоставляя в наше распоряжение более элегантный синтаксис и большую мощность.

Выполнение операций после загрузки страницы

Ранее уже демонстрировалось, как с помощью средств библиотеки jQuery реализовать реакцию на событие загрузки страницы. Для запуска программного кода можно использовать обработчик события `$(document).ready()`, о котором стоит рассказать подробнее.

Момент запуска программного кода

В главе 1 отмечалось, что обработчик `$(document).ready()` в библиотеке jQuery может использоваться для тех же целей, что и обработчик `onload`, встроенный в JavaScript. Несмотря на то что эти обработчики имеют похожее действие, они запускаются в едва различимые моменты времени.

Событие `window.onload` возникает тогда, когда документ полностью загружен броузером, то есть когда все элементы страницы доступны для выполнения операций над ними из программного кода JavaScript что позволяет писать полноценный программный код, не волнуясь об очередности загрузки.

С другой стороны, обработчик, регистрируемый с помощью `$(document).ready()`, вызывается тогда, когда дерево DOM полностью готово к использованию, то есть когда все элементы дерева доступны сценарию, но при этом, возможно, не все ассоциированные со страницей файлы загружены. Программный код может быть запущен, как только разметка HTML будет загружена и преобразована в дерево DOM.

Примечание

Чтобы обеспечить применение стилей оформления до запуска программного кода JavaScript, лучше всего поместить теги `<link rel="stylesheet">` перед тегами `<script>` внутри элемента `<head>` документа.

В качестве примера рассмотрим страницу, представляющую галерею изображений. Такая страница может содержать массу изображений, которые с помощью библиотеки jQuery мы можем скрывать, отображать, перемещать и так далее. Если выполнить настройку интерфейса по событию `onload`, пользователь вынужден будет ожидать, пока загрузятся все изображения, и только после этого у него появится возможность взаимодействовать со страницей. Хуже того, если реализация поведения еще не подключена к элементам, предусматривающим некоторую реакцию по умолчанию (например, ссылкам), то действия пользователя могут привести к не продуманным заранее результатам. В случае реализации настройки интерфейса с помощью `$(document).ready()` интерфейс окажется готовым к использованию намного раньше и будет иметь корректное поведение.

Примечание

Использование обработчика `$(document).ready()` практически всегда предпочтительнее, чем использование `onload`, но не следует забывать, что поскольку к моменту вызова обработчика `$(document).ready()` некоторые файлы могут быть еще не загружены, такие атрибуты, как высота и ширина изображений, не всегда могут быть доступны к этому моменту времени. Если подобные атрибуты необходимы программному коду, то имеет смысл реализовать обработчик `onload` (или даже лучше воспользоваться jQuery для установки обработчика события `load`). Оба механизма могут прекрасно сосуществовать друг с другом.

Множество сценариев в одной странице

Традиционный способ регистрации обработчиков событий в JavaScript (помимо добавления атрибутов обработчиков непосредственно в разметке HTML) заключается в присваивании функций соответствующим атрибутам элементов DOM. Например, предположим, что определена следующая функция:

```
function doStuff() {
  // Реализация некоторых действий...
}
```

Мы можем использовать ее в качестве значения атрибута в разметке HTML:

```
<body onload="doStuff();">
```

или выполнить присваивание ее в программном коде JavaScript:

```
window.onload = doStuff;
```

В обоих случаях функция будет вызвана сразу же после загрузки страницы. Преимущество второго подхода состоит в отделении реализации поведения от разметки.

Примечание

Обратите внимание, что при присваивании функции атрибуту события в коде разметки указывается только ее имя, без круглых скобок. Если указать круглые скобки, функция *вызовется* немедленно; имя без скобок просто *идентифицирует* функцию и может быть использовано для ее вызова позднее.

Такой подход может применяться при наличии лишь одной функции. Допустим, что имеется вторая функция:

```
function doOtherStuff() {
  // Реализация некоторых других действий...
}
```

Можно попытаться присвоить эту функцию обработчику события загрузки страницы:

```
window.onload = doOtherStuff;
```

Однако тем самым отменится первое присваивание. Атрибут `.onload` может хранить только одну ссылку на функцию, поэтому, используя данный способ, мы не сможем дополнить существующую реализацию поведения.

Механизм на основе `$(document).ready()` обрабатывает рассматриваемую ситуацию гораздо изящнее. При каждом обращении этот метод добавляет новую функцию во внутреннюю очередь. В результате, когда страница загрузится, будут выполнены все функции. Вызов функций будет происходить в порядке их регистрации.

Примечание

Справедливости ради следует отметить, что библиотека jQuery не обладает монополией на способы решения этой проблемы. Можно написать функцию JavaScript, формирующую новую функцию для вызова существующего обработчика `onload`, а затем вызывающую переданный ей обработчик. Такой прием используется, например, функцией `addLoadEvent()` Саймона Уиллисона (Simon Willison), которая предотвращает конфликты между альтернативными обработчиками, подобно методу `$(document).ready()`, но в ней отсутствуют некоторые другие преимущества, обсуждавшиеся выше. Методы, характерные

для разных броузеров, такие как `document.addEventListener()` и `document.attachEvent()`, предлагают похожие функциональные возможности, но библиотека jQuery позволяет решить данную задачу, не беспокоясь о проблеме совместимости броузеров.

Сокращения в программном коде

Конструкция `$(document).ready()` в действительности представляет собой вызов метода `.ready()` объекта jQuery, сконструированного из элемента DOM `document`. **Фабричная функция** `$()` дает возможность более кратко записать решение этой распространенной задачи. При вызове без аргументов фабричная функция ведет себя так, как если бы ей был передан аргумент `document`. Это означает, что вместо вызова:

```
$(document).ready(function() {
  // Здесь находится наш программный код...
});
```

мы можем записать:

```
$().ready(function() {
  // Здесь находится наш программный код...
});
```

Кроме того, фабричная функция может принимать в виде аргумента другую функцию. В таком случае jQuery неявно вызывает метод `.ready()`, и для получения того же результата мы можем воспользоваться следующей конструкцией:

```
$(function() {
  // Здесь находится наш программный код...
});
```

Несмотря на то, что эти варианты являются более краткими, авторы рекомендуют использовать более длинную версию, так как она более ясно демонстрирует действия, выполняемые программным кодом.

Сосуществование с другими библиотеками

В некоторых случаях может оказаться полезным использовать несколько библиотек JavaScript в одной странице. Но, так как во многих библиотеках используется идентификатор `$` (вследствие его краткости и удобству), нам необходимо иметь возможность предотвращать конфликты имен.

К счастью, библиотека jQuery предоставляет на этот случай метод с именем `.noConflict()`, возвращающий контроль над идентификатором `$` обратно другим библиотекам. Обычно метод `.noConflict()` используется, как показано ниже:

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

```
<script type="text/javascript">
  jQuery.noConflict();
</script>
<script src="myscript.js" type="text/javascript"></script>
```

Сначала выполняется подключение другой библиотеки (в данном случае библиотеки Prototype). Затем подключается сама библиотека jQuery и получает контроль над идентификатором `$`. После этого вызывается метод `.noConflict()`, который освобождает идентификатор `$`, благодаря чему контроль над ним возвращается первой подключенной библиотеке (Prototype). Теперь внутри своих сценариев мы можем использовать обе библиотеки, но при этом всякий раз, когда потребуется вызвать метод из библиотеки jQuery, нам придется вместо идентификатора `$` применять идентификатор `jQuery`.

Метод `.ready()` обладает еще одной хитрой особенностью, способной оказать нам помощь в подобной ситуации. Функция обратного вызова, что передается этому методу, может принимать единственный аргумент – сам объект jQuery. Это позволяет эффективно переименовать его, не опасаясь возможности появления конфликтов:

```
jQuery(document).ready(function($) {
  // Здесь можно использовать идентификатор $ как обычно!
});
```

Или используя более краткую форму записи, изученную выше:

```
jQuery(function($) {
  // Программный код, использующий идентификатор $.
});
```

Простые события

Помимо загрузки страницы существует множество других ситуаций, когда может потребоваться выполнить некоторые действия. JavaScript не только позволяет обрабатывать событие загрузки страницы с помощью `<body onload="">` или `window.onload`, но и предоставляет похожие обработчики для событий, инициируемых пользователем, таких как щелчок мыши (`onclick`), изменение полей формы (`onchange`) и изменение размера окна (`onresize`). При присваивании непосредственно в дереве DOM эти обработчики обладают теми же недостатками, что и обработчик `onload`. Поэтому библиотека jQuery опять же предлагает улучшенный способ обработки таких событий.

Простой переключатель стилей

Для иллюстрации некоторых приемов обработки событий предположим, что имеется одна страница, которая может отображаться с применением различных стилей в зависимости от желания пользователя.

Мы дадим пользователю возможность щелчком на кнопке переключаться между обычным представлением страницы, представлением, при котором вывод текста производится в пределах суженной области, и представлением, при котором текст выводится по всей ширине страницы увеличенным шрифтом.

В реальной ситуации грамотный веб-разработчик применит здесь принцип **прогрессивного улучшения**. Кнопки переключения стилей должны либо скрываться, если поддержка JavaScript отключена в броузере, либо, что еще лучше, функционировать как ссылки на альтернативные версии страницы. Будем предполагать, что все пользователи включили поддержку JavaScript в своих броузерах.

Ниже приводится разметка HTML для страницы с поддержкой переключения стилей:

```
<div id="switcher">
  <h3>Style Switcher</h3>
  <div class="button selected" id="switcher-default">
    Default
  </div>
  <div class="button" id="switcher-narrow">
    Narrow Column
  </div>
  <div class="button" id="switcher-large">
    Large Print
  </div>
</div>
```

Объединив этот фрагмент с остальной частью разметки HTML и с простой таблицей стилей CSS, мы получим страницу, аналогичную показанной на рис. 3.1.

Для начала мы реализуем действие кнопки Large Print (увеличенный размер). Чтобы обеспечить альтернативное представление страницы, необходимо определить класс CSS:

```
body.large .chapter {
  font-size: 1.5em;
}
```

Следующая наша цель – применить класс `large` к тегу `<body>`. Это позволит изменить оформление страницы в соответствии с таблицей стилей. Имея сведения, полученные в главе 2, мы уже знаем, что эта цель может быть достигнута с помощью инструкции:

```
$('body').addClass('large');
```

Однако нам необходимо, чтобы программный код выполнялся в результате щелчка на кнопке, а не в момент загрузки страницы, как это делалось до сих пор. Для этого мы будем использовать метод `.bind()`, который позволяет указать любое **событие JavaScript** и подключить к нему

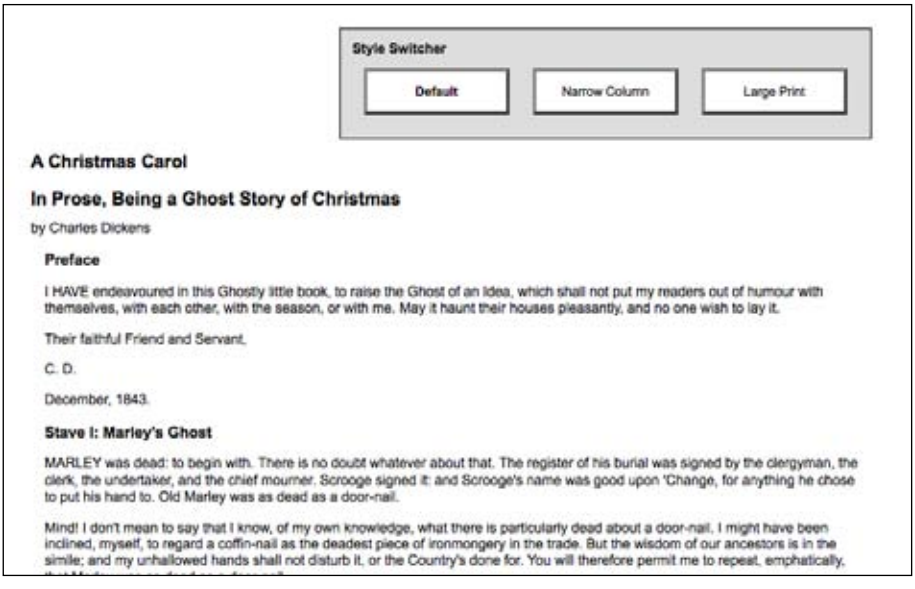


Рис. 3.1. *Первоначальный вид переключателя стилей*

его обработчик. В данном случае событие называется `click`, а обработчиком является функция, содержащая строку, представленную выше:

```
$(document).ready(function() {
  $('#switcher-large').bind('click', function() {
    $('body').addClass('large');
  });
});
```

Теперь наш программный код будет вызываться после щелчка на кнопке, и для вывода текста будет использоваться шрифт увеличенного размера, как показано на рис. 3.2.

Это все, что требуется для подключения обработчика события. Данный способ обладает преимуществами, приведенными при обсуждении метода `.ready()`. В случае необходимости мы можем многократно вызывать метод `.bind()`, добавляя дополнительные обработчики к одному и тому же событию.

Это не самое элегантное и эффективное решение поставленной задачи. Ниже в данной главе мы дополним и усовершенствуем его, превратив в то, чем уже можно будет гордиться.

Обработка других кнопок

Теперь у нас имеется работающая кнопка Large Print, но нам необходимо определить аналогичные обработчики еще для двух других кнопок – Default (по умолчанию) и Narrow Column (узкая колонка), чтобы они вы-

Рис. 3.2. *Вид страницы после щелчка на кнопке Large Print (увеличенный размер)*

полняли свои функции. Сделать это очень просто – с помощью метода `.bind()` мы добавим к каждой кнопке обработчики события `click`, в которых удалим и добавим необходимые классы. Новый программный код приводится ниже:

```
$(document).ready(function() {
  $('#switcher-default').bind('click', function() {
    $('body').removeClass('narrow');
    $('body').removeClass('large');
  });
  $('#switcher-narrow').bind('click', function() {
    $('body').addClass('narrow');
    $('body').removeClass('large');
  });
  $('#switcher-large').bind('click', function() {
    $('body').removeClass('narrow');
    $('body').addClass('large');
  });
});
```

В дополнение к нему определим новый класс CSS `narrow`:

```
body.narrow .chapter {
  width: 400px;
}
```

Теперь после щелчка на кнопке Narrow Column будет применяться соответствующее правило CSS, и текст будет отображаться по-другому, как показано на рис. 3.3.

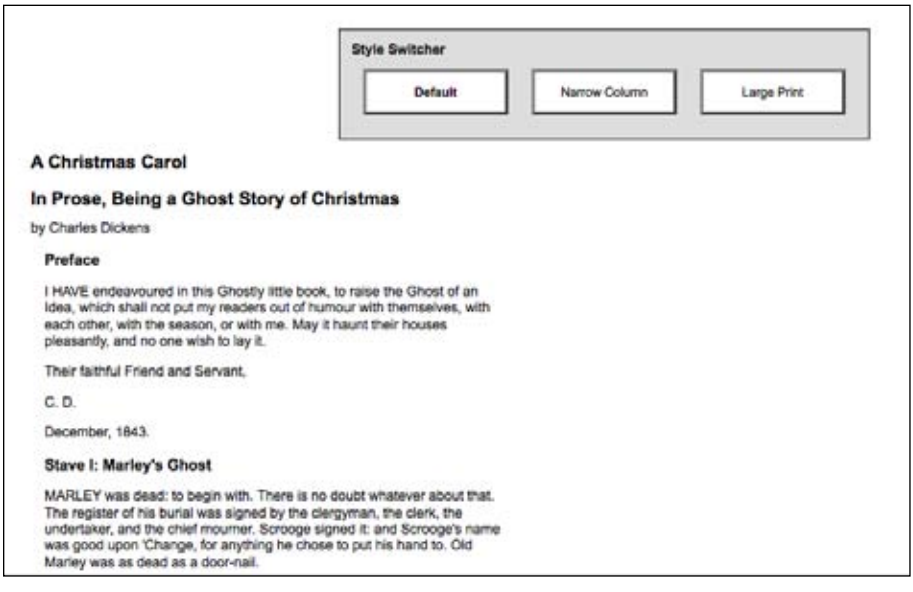


Рис. 3.3. *Вид страницы после щелчка на кнопке Narrow Column (узкая колонка)*

Обработчик щелчка на кнопке Default будет удалять оба имени класса CSS из тега `<body>`, возвращая страницу к первоначальному варианту отображения.

Контекст обработчика события

Наш переключатель действует вполне корректно, но мы не предоставили пользователю обратную связь, чтобы он мог видеть, какая кнопка в настоящий момент является активной. Для реализации такой обратной связи мы применим класс `selected` к кнопке, на которой был произведен щелчок, и удалим его из всех остальных кнопок. Класс `selected` просто обеспечивает вывод надписи на кнопке жирным шрифтом:

```
.selected {
  font-weight: bold;
}
```

Мы могли бы добавить этот класс тем же способом, что и выше, ссылаясь на каждую кнопку по ее идентификатору и применяя или удаляя классы по мере необходимости, но мы рассмотрим более элегантное и масштабируемое решение, основанное на использовании **контекста** обработчика события.

Когда вызывается обработчик некоторого события, в виде ключевого слова `this` ему передается ссылка на элемент DOM, в котором возникло это событие. Ранее уже отмечалось, что фабричная функция `$()` может принимать элемент DOM в виде аргумента – этим и обусловлена описанная особенность. Вызывая метод `$(this)` внутри обработчика события, мы создаем объект jQuery, соответствующий элементу, и получаем возможность оперировать им, как если бы он был получен с помощью селектора CSS.

Таким образом, мы можем записать вызов:

```
$(this).addClass('selected');
```

Поместив эту строку в каждый из трех обработчиков, мы сможем добавлять класс к кнопке по событию щелчка на ней. Для удаления класса у других кнопок можно воспользоваться механизмом неявных итераций jQuery и записать:

```
$('#switcher .button').removeClass('selected');
```

Эта строка удаляет класс у всех остальных кнопок, присутствующих в странице переключателя стилей. Разместив указанные строки в требуемом порядке, мы получим следующий программный код:

```
$(document).ready(function() {
  $('#switcher-default').bind('click', function() {
    $('body').removeClass('narrow');
    $('body').removeClass('large');
    $('#switcher .button').removeClass('selected');
    $(this).addClass('selected');
  });
  $('#switcher-narrow').bind('click', function() {
    $('body').addClass('narrow');
    $('body').removeClass('large');
    $('#switcher .button').removeClass('selected');
    $(this).addClass('selected');
  });
  $('#switcher-large').bind('click', function() {
    $('body').removeClass('narrow');
    $('body').addClass('large');
    $('#switcher .button').removeClass('selected');
    $(this).addClass('selected');
  });
});
```

Теперь переключатель стилей предоставляет подобающую обратную связь, как показано на рис. 3.4.

Обобщение инструкций за счет использования контекста обработчика события позволяет нам добиться еще большей эффективности. Мы можем оформить процедуру выделения кнопки в виде отдельного обработчика, как показано ниже, поскольку она остается неизменной для всех трех кнопок:

```
$(document).ready(function() {
  $('#switcher-default').bind('click', function() {
    $('body').removeClass('narrow').removeClass('large');
  });
  $('#switcher-narrow').bind('click', function() {
    $('body').addClass('narrow').removeClass('large');
  });
  $('#switcher-large').bind('click', function() {
    $('body').removeClass('narrow').addClass('large');
  });

  $('#switcher .button').bind('click', function() {
    $('#switcher .button').removeClass('selected');
    $(this).addClass('selected');
  });
});
```

Эта оптимизация использует преимущества трех особенностей библиотеки jQuery, обсуждавшихся выше. Во-первых, **неявная итерация** позволяет присоединить обработчики события `click` ко всем трем кнопкам с помощью одного вызова метода `.bind()`. Во-вторых, **очереди обработчиков** позволяют нам назначить вторую функцию для обработки того же события `click` без перезаписи ссылки на первую функцию. Наконец, мы можем использовать возможность **составления цепочек** методов для реализации добавления и удаления классов в одной строке.

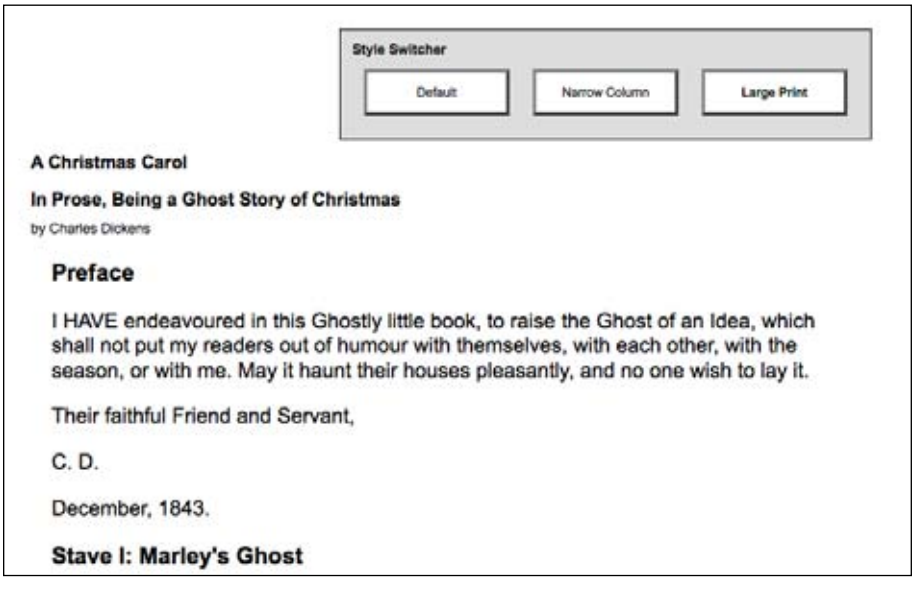


Рис. 3.4. *Кнопка, на которой был выполнен щелчок, выделяется шрифтом*

Дальнейшее уплотнение кода

Оптимизация программного кода, которую мы только что провели, является примером **рефакторинга** – модификации существующего программного кода для выполнения той же задачи более эффективным и элегантным способом. С целью изучения других возможностей рефакторинга рассмотрим детальнее обработчики событий, присоединяемые к каждой из кнопок. Аргумент метода `.removeClass()` является необязательным; если он отсутствует, метод удалит из элемента *все* классы. Мы можем упростить наш программный код, воспользовавшись этой особенностью, как показано ниже:

```
$(document).ready(function() {
  $('#switcher-default').bind('click', function() {
    $('body').removeClass();
  });
  $('#switcher-narrow').bind('click', function() {
    $('body').removeClass().addClass('narrow');
  });
  $('#switcher-large').bind('click', function() {
    $('body').removeClass().addClass('large');
  });

  $('#switcher .button').bind('click', function() {
    $('#switcher .button').removeClass('selected');
    $(this).addClass('selected');
  });
});
```

Обратите внимание, как изменился порядок выполнения операций из-за того, что мы выполняем удаление всех классов: метод `.removeClass()` должен вызываться первым, чтобы не отменить действие метода `.addClass()`, вызываемого в той же цепочке.

Примечание

Мы можем без оглядки удалять все классы только в данном случае, потому что мы сами отвечаем за разметку HTML. При разработке программного кода для многократного использования (например, при разработке модулей расширения) необходимо учитывать, что элементы могут содержать другие классы, которые следует оставлять незатронутыми.

Теперь в обработчиках событий для всех кнопок используется один и тот же программный код. Его легко можно превратить в универсальный обработчик события `click`:

```
$(document).ready(function() {
  $('#switcher .button').bind('click', function() {
    $('body').removeClass();
    $('#switcher .button').removeClass('selected');
```

```
    $(this).addClass('selected');
  });
  $('#switcher-narrow').bind('click', function() {
    $('body').addClass('narrow');
  });
  $('#switcher-large').bind('click', function() {
    $('body').addClass('large');
  });
});
```

Обратите внимание, что нам пришлось переместить универсальный обработчик выше специализированных. Метод `.removeClass()` должен вызываться раньше метода `.addClass()`, и мы можем учесть это обстоятельство благодаря тому, что библиотека jQuery всегда вызывает обработчики в порядке их регистрации.

Наконец, мы можем полностью избавиться от специализированных обработчиков, снова воспользовавшись контекстом обработчика события. Так как ключевое слово `this` содержит ссылку на элемент DOM, а не на объект jQuery, мы можем применить встроенные свойства DOM для определения идентификатора элемента, на котором был произведен щелчок. Благодаря этому мы можем присоединить один и тот же обработчик ко всем кнопкам и выполнять внутри него действия, различные для всех кнопок:

```
$(document).ready(function() {
  $('#switcher .button').bind('click', function() {
    $('body').removeClass();
    if (this.id == 'switcher-narrow') {
      $('body').addClass('narrow');
    }
    else if (this.id == 'switcher-large') {
      $('body').addClass('large');
    }
    $('#switcher .button').removeClass('selected');
    $(this).addClass('selected');
  });
});
```

Сокращенная форма подключения обработчиков

Операция подключения обработчика некоторого события (подобного простому событию `click`) является настолько распространенной, что в библиотеке jQuery предусмотрен еще более краткий способ ее реализации; **сокращенные методы событий** действуют точно так же, как и метод `.bind()`, но требуют от разработчика меньшего числа нажатий клавиш.

Например, в нашем переключателе стилей можно было бы вместо метода `.bind()` использовать метод `.click()`, как показано ниже:

```
$(document).ready(function() {
  $('#switcher .button').click(function() {
    $('body').removeClass();
    if (this.id == 'switcher-narrow') {
      $('body').addClass('narrow');
    }
    else if (this.id == 'switcher-large') {
      $('body').addClass('large');
    }

    $('#switcher .button').removeClass('selected');
    $(this).addClass('selected');
  });
});
```

Сокращенные методы событий определены для всех стандартных событий DOM:

- blur
- change
- click
- dblclick
- error
- focus
- keydown
- keypress
- keyup
- load
- mousedown
- mousemove
- mouseout
- mouseover
- mouseup
- resize
- scroll
- select
- submit
- unload

Каждый из этих методов присоединяет обработчик к событию с соответствующим именем.

Комбинированные события

Большинство методов обработки событий в библиотеке jQuery напрямую соответствует событиям JavaScript. Однако имеется несколько дополнительных обработчиков, добавленных для удобства и преодоления несовместимости броузеров. Один из них, метод `.ready()`, мы уже подробно рассмотрели. Методы `.toggle()` и `.hover()` представляют два других дополнительных обработчика событий; они оба являются **комбинированными обработчиками событий**, потому что отвечают за обработку комбинаций действий пользователя, для реализации реакции на которые требуется более одной функции.

Отображение и сокрытие дополнительных возможностей

Предположим, что нам требуется скрывать переключатель стилей, когда в нем нет необходимости. Осуществить это можно, например, сделав дополнительные возможности свертываемыми: пусть по одному щелчку на метке кнопки скрываются, оставляя только саму метку, а по второму щелчку на метке кнопки восстанавливаются. Тогда нам потребуется еще один класс, который будет присваиваться кнопкам для их сокрытия:

```
.hidden {
  display: none;
}
```

Мы могли бы реализовать эту возможность путем сохранения текущего состояния кнопок в переменной и анализа ее значения при каждом щелчке на метке с целью решения, добавлять к кнопкам класс `hidden` или удалять его. Мы могли бы также напрямую определять присутствие класса в кнопках и использовать эту информацию при выборе требуемого действия. Но вместо этого мы применим метод `.toggle()`, предоставляемый библиотекой jQuery, который самостоятельно выполняет все вспомогательные операции.

Примечание

В действительности в библиотеке jQuery определено два метода `.toggle()`. За дополнительной информацией о назначении второго метода с этим именем (который отличается от первого типами аргументов) обращайтесь по адресу: *http://docs.jquery.com/Effects/toggle*

Метод `.toggle()` принимает два или более аргументов, каждый из которых является функцией. Первый щелчок на элементе приводит к вызову первой функции, второй – к вызову второй функции, и так далее. После того как будут вызваны все функции, цикл начнется опять с первой функции. С помощью метода `.toggle()` мы легко можем организовать свертывание нашего переключателя стилей:

```
$(document).ready(function() {
  $('#switcher h3').toggle(function() {
    $('#switcher .button').addClass('hidden');
  }, function() {
    $('#switcher .button').removeClass('hidden');
  });
});
```

После первого щелчка все кнопки будут скрыты, как показано на рис. 3.5.

Рис. 3.5. *После первого щелчка кнопки скрываются*

А после второго – опять станут видимыми, как показано на рис. 3.6.

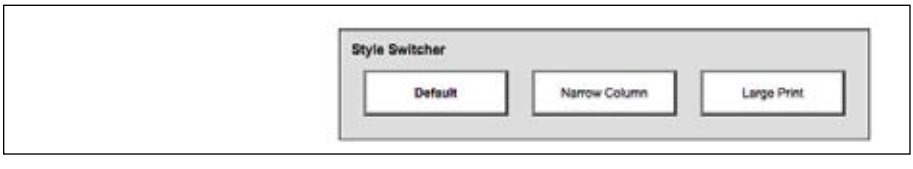


Рис. 3.6. *После второго щелчка кнопки опять становятся видимыми*

Здесь мы опять опираемся на использование механизма неявных итераций, на этот раз чтобы одним махом скрыть все кнопки, за исключением объемлющего элемента.

В данном конкретном случае библиотека jQuery предоставляет еще один механизм свертывания. Мы можем использовать метод `.toggleClass()`, который автоматически проверяет наличие требуемого класса, прежде чем добавить его или удалить:

```
$(document).ready(function() {
  $('#switcher h3').click(function() {
    $('#switcher .button').toggleClass('hidden');
  });
});
```

В данном случае решение на основе метода `.toggleClass()` выглядит более элегантно, однако метод `.toggle()` обеспечивает более универсальный способ поочередного выполнения двух или более операций.

Выделение элементов, предусматривающих реакцию на щелчок мышью

Чтобы показать, что некоторый элемент, который обычно не реагирует на щелчок мышью, предусматривает реакцию на это событие, необходимо создать интерфейс, который сам подсказывал бы, что кнопки,

в действительности простые элементы , являются *действующими* частями страницы, ожидающими действий пользователя. Для этого следует определить такое состояние кнопок, которое наглядно демонстрировало бы, что они способны некоторым образом взаимодействовать с мышью:

```
#switcher .hover {
  cursor: pointer;
  background-color: #afa;
}
```

Спецификация CSS включает псевдокласс с именем :hover, который позволяет определять стиль элемента, когда над ним находится курсор мыши. В Internet Explorer 6 такая возможность применима только к ссылкам, поэтому мы не можем ее использовать в случае требования обеспечения совместимости со всеми типами броузеров. Однако библиотека jQuery позволяет использовать JavaScript для изменения оформления элементов и, более того, для выполнения любых произвольных действий как в случае перемещения указателя мыши в пределы элемента, так и в случае выхода указателя за его пределы.

Метод .hover() принимает в виде аргументов две функции, как и метод .toggle() в примере выше. В данном случае первая функция будет вызываться при перемещении указателя мыши в пределы выбранного элемента, а вторая – при покидании указателем мыши пределов этого элемента. Для достижения требуемого эффекта мы можем в эти моменты изменять классы, применяемые к кнопкам:

```
$(document).ready(function() {
  $('#switcher .button').hover(function() {
    $(this).addClass('hover');
  }, function() {
    $(this).removeClass('hover');
  });
});
```

Рис. 3.7. *После наведения указателя мыши на кнопку можно наблюдать применение класса*

Мы снова использовали механизм неявной итерации и контекст события, чтобы сократить и упростить реализацию. Теперь при наведении указателя мыши на кнопку можно наблюдать применение класса, как показано на рис. 3.7.

Использование метода `.hover()` позволяет также избежать проблем, связанных с **распространением событий** в JavaScript. Чтобы понять, о чем идет речь, необходимо рассмотреть, как JavaScript определяет, какие элементы должны обрабатывать заданное событие.

Распространение события

Когда в странице возникает некоторое событие, целая иерархия элементов DOM получает шанс обработать его. Рассмотрим следующую модель страницы:

```
<div class="foo">
  <span class="bar">
    <a href="http://www.example.com/">
      The quick brown fox jumps over the lazy dog.
    </a>
  </span>
  <p>
    How razorback-jumping frogs can level six piqued gymnasts!
  </p>
</div>
```

Мы изобразили эту разметку в виде совокупности вложенных элементов, как показано на рис. 3.8.

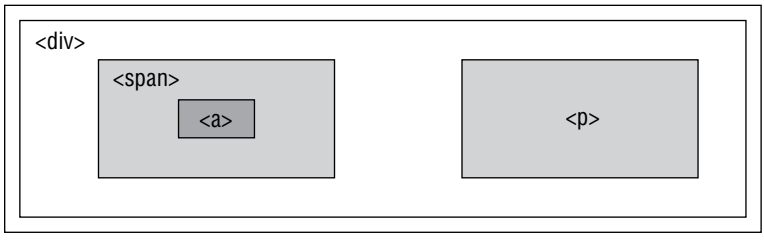


Рис. 3.8. *Разметка страницы в виде совокупности вложенных элементов*

Для каждого события существует множество элементов, которые логически могли бы обеспечить реакцию на него. Например, когда на рассматриваемой странице выполняется щелчок на ссылке, возможность отреагировать на него должны получить элементы `<div>`, `<span>` и `<a>`. Ведь в момент щелчка все три элемента одновременно находятся под указателем мыши. С другой стороны, элемент `<p>` вообще не участвует в этом взаимодействии.

Одна из стратегий, позволяющих нескольким элементам реагировать на щелчок, называется **перехватом события.** В случае перехвата события оно сначала передается самому внешнему элементу, а затем последовательно передается вложенным элементам. В нашем примере первым событие получит элемент , затем и, наконец, <a>, как показано на рис. 3.9.

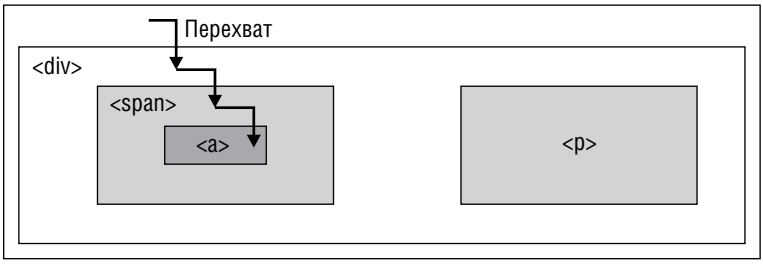


Рис. 3.9. *Порядок движения события в фазе перехвата*

Примечание

Технически реализация механизма перехвата событий в броузерах требует **регистрации** каждого конкретного элемента, который должен получать события, предназначенные для его потомков. Приближенная модель, представленная здесь, достаточно точно отвечает нашим потребностям.

Противоположная стратегия называется **всплытием события.** Сначала событие передается самому внутреннему элементу и после того, как этот элемент получает возможность среагировать на него, событие **всплывает,** то есть последовательно передается вмещающим элементам. При использовании такой стратегии первым элементом в нашем примере, который получит событие, будет элемент <a>, затем оно по порядку будет передано элементам и , как показано на рис. 3.10.

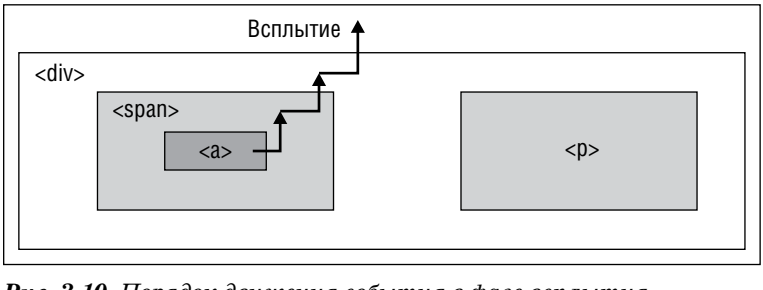


Рис. 3.10. *Порядок движения события в фазе всплытия*

Разработчики разных броузеров изначально выбрали различные модели распространения событий. Вследствие этого выработанный со временем стандарт DOM требует использования обеих стратегий: снача-

ла событие должно пройти фазу **перехвата**, распространяясь от самого внешнего элемента к внутреннему, а затем должно **всплыть** обратно к вершине дерева DOM. Обработчики событий могут регистрироваться для любой из этих фаз распространения.

Не все броузеры следуют этому стандарту, а те, что поддерживают фазу перехвата, обычно реализуют эту возможность отдельно. По этой причине, чтобы обеспечить совместимость между броузерами, библиотека jQuery всегда регистрирует обработчики событий для фазы всплытия. Мы всегда можем смело предполагать, что возможность отреагировать на любое событие первым получит самый внутренний элемент.

Побочные эффекты фазы всплытия

Всплытие события может вызывать побочные эффекты, особенно когда элементы по ошибке реагируют на событие `mouseover` или `mouseout`. Рассмотрим обработчик события `mouseout`, присоединенный в нашем примере к элементу `<div>`. Когда указатель мыши покидает пределы элемента `<div>`, вызывается обработчик события `mouseout`, как и ожидается. Поскольку данный элемент находится на вершине иерархии, никакой другой элемент не получает это событие. С другой стороны, когда указатель мыши покидает пределы элемента `<a>`, этому элементу передается событие `mouseout`. После чего это событие будет всплывать сначала к элементу `<span>`, а затем к элементу `<div>`, вызывая запуск того же обработчика события. Такая последовательность всплытия не всегда бывает желательной; для случая кнопок переключателя стилей в нашем примере это могло бы привести к преждевременному снятию выделения.

Метод `.hover()` учитывает описанные выше особенности всплытия событий, благодаря чему при использовании этого метода для подключения обработчика мы можем забыть о проблемах, связанных с передачей события `mouseover` или `mouseout` не тому элементу. Это делает метод `.hover()` очень привлекательной альтернативой технике присваивания отдельных обработчиков событий мыши.

Примечание

Если требуется реализовать реакцию только на событие наведения указателя мыши на элемент или вывода его за пределы элемента, но не на оба сразу, можно присвоить обработчики событий `mouseenter` и `mouseleave`, реализованные в библиотеке jQuery, которые также устраняют проблемы, связанные с всплытием событий. Однако эти обработчики так часто используются в паре, что, как правило, метод `.hover()` является оптимальным выбором.

Только что описанная ситуация, связанная с событием `mouseout`, наглядно демонстрирует необходимость ограничивать область видимости события. Несмотря на то, что метод `.hover()` корректно обрабатывает конкретный случай, мы будем сталкиваться с другими ситуаци-

ями, требующими ограничения распространения события в пространстве (предотвращения возможности передачи события определенным элементам) или во времени (предотвращения возможности передачи события в определенные моменты времени).

Изменение движения события: объект события

Мы уже видели одну ситуацию, когда **всплытие события** может породить проблемы. Чтобы показать случай, когда метод `.hover()` оказывается не в состоянии помочь в решении наших проблем, мы изменим реализацию скрытия элементов, продемонстрированную выше.

Предположим, что нам необходимо расширить область, на которой можно щелкнуть, чтобы свернуть или развернуть переключатель стилей. Один из способов добиться желаемого эффекта заключается в том, чтобы переместить обработчик события из метки, элемент `<h3>`, в содержащий ее элемент `<div>`:

```
$(document).ready(function() {
  $('#switcher').click(function() {
    $('#switcher .button').toggleClass('hidden');
  });
});
```

Это изменение позволит использовать всю область переключателя стилей для управления его видимостью. Недостаток такой реализации состоит в том, что щелчок на кнопке будет также приводить к свертыванию переключателя стилей после изменения оформления содержимого страницы. Это обусловлено всплытием события: сначала событие обрабатывается кнопками, а затем распространяется вверх по дереву DOM, пока не достигнет элемента `<div id="switcher">`, где активируется наш новый обработчик, и скрывает кнопки.

Чтобы решить данную проблему, нам необходимо получить доступ к **объекту события**, то есть к конструкции JavaScript, передаваемой при вызове каждого обработчика события. Этот объект содержит информацию о событии, такую как координаты указателя мыши в момент появления события. Кроме того, он обладает некоторыми методами, которые могут применяться для оказания влияния на дальнейшее распространение события по дереву DOM.

Чтобы получить возможность использовать объект события внутри обработчика, достаточно добавить параметр в функцию:

```
$(document).ready(function() {
  $('#switcher').click(function(event) {
    $('#switcher .button').toggleClass('hidden');
  });
});
```

Адресаты событий

Теперь у нас имеется возможность обратиться к объекту события через переменную `event`. Чтобы определить, *где* событие вступает в силу, можно использовать свойство `event.target`. Это свойство является частью прикладного интерфейса объектной модели документа (DOM API), но реализовано не во всех броузерах. В случае необходимости библиотека jQuery расширяет объект события, чтобы обеспечить доступность данного свойства во всех броузерах. С помощью свойства `.target` можно определить, какой элемент в дереве DOM первым должен принимать событие, то есть в случае события `click` фактический элемент, на котором выполнен щелчок. Помня о том, что ключевое слово `this` хранит ссылку на элемент DOM, обрабатывающий событие, мы можем написать следующую реализацию:

```
$(document).ready(function() {
  $('#switcher').click(function(event) {
    if (event.target == this) {
      $('#switcher .button').toggleClass('hidden');
    }
  });
});
```

Такой подход гарантирует, что действие будет выполняться только в случае щелчка на самом элементе `<div id="switcher">`, а не на одном из его подэлементов. Теперь щелчок на кнопке *не будет* приводить к свертыванию переключателя стилей, а щелчок в области фона переключателя *будет*. В свою очередь, щелчок на метке, элемент `<h3>`, не будет вызывать никакой реакции, потому что она также является подэлементом. Для достижения наших целей вместо выполнения подобной проверки мы можем изменить обработчики событий кнопок.

Остановка распространения события

Объект события предоставляет метод `.stopPropagation()`, с помощью которого можно полностью прервать процесс всплытия события. Подобно свойству `.target` этот метод является обычным методом JavaScript, но он определен не во всех броузерах. Однако если регистрация обработчика события производится с помощью библиотеки jQuery, мы можем использовать его без всякой опаски.

Удалим только что добавленную проверку `event.target == this`, но добавим следующие строки в обработчик события `click` кнопок:

```
$(document).ready(function() {
  $('#switcher .button').click(function(event) {
    $('body').removeClass();
    if (this.id == 'switche-rnarrow') {
      $('body').addClass('narrow');
    }
```

```
    else if (this.id == 'switcher-large') {
      $('body').addClass('large');
    }
    $('#switcher .button').removeClass('selected');
    $(this).addClass('selected');
    event.stopPropagation();
  });
});
```

Как и прежде, чтобы получить доступ к объекту события, мы передаем дополнительный параметр функции, которая будет играть роль обработчика события `click`. Затем мы просто вызываем метод `event.stopPropagation()`, чтобы предотвратить возможность обработки события любыми другими элементами дерева DOM. Теперь щелчки на кнопках будут обрабатываться кнопками и только кнопками; щелчки на любых других местах в пределах переключателя стилей будут приводить к его сворачиванию или разворачиванию.

Действия по умолчанию

Если бы наш обработчик события `click` был зарегистрирован не для элемента `<div>`, а для элемента ссылки (`<a>`), мы могли бы столкнуться с другой проблемой. После щелчка на ссылке броузер загружает новую страницу. Такое поведение обеспечивается не **обработчиком события** в том смысле, в каком обсуждалось, а является **действием по умолчанию**, предусмотренным для события щелчка мышью на ссылке. Аналогично, когда в процессе заполнения формы нажимается клавиша Enter, возбуждается событие `submit` формы, но после этого происходит фактическая передача формы.

Если действия, предусмотренные по умолчанию, нежелательны, вызов метода `.stopPropagation()` объекта события не поможет. Эти действия выполняются за пределами обычного процесса распространения события. В такой ситуации следует вызвать метод `.preventDefault()`, который служит для остановки распространения события, прежде чем будет предпринято действие по умолчанию.

Примечание

Вызов `.preventDefault()` часто удобно производить после выполнения некоторого анализа окружения события. Например, перед отправкой формы можно проверить, заполнены ли обязательные поля, и предотвращать действие по умолчанию только, если они не заполнены. Как это делается, будет показано в главе 8.

Распространение событий и действия по умолчанию – независимые механизмы. Мы можем препятствовать работе одного из них, в то время как другой будет продолжать работать. Если необходимо прервать работу обоих механизмов сразу, функция-обработчик должна вернуть

значение `false`, что является сокращенным эквивалентом вызова обоих методов, `.stopPropagation()` и `.preventDefault()`, объекта события.

Делегирование событий

Всплытие события не всегда является помехой, довольно часто его можно использовать с немалой выгодой. Один из приемов, использующих всплытие события, называется **делегированием события**. С его помощью можно задействовать обработчик событий одного элемента для обработки событий множества элементов.

Примечание

В версии jQuery 1.3 появилась пара новых методов, `.live()` и `.die()`. Они решают те же задачи, что и методы `.bind()` и `.unbind()`, но для получения преимуществ, описываемых в данном разделе, они используют прием делегирования событий. Описание этих методов можно найти по адресу: *http://docs.jquery.com/Events/live*

В нашем примере имеются всего три элемента `<div class="button">`, к которым подключаются обработчики события `click`. Но как быть, если элементов намного больше? Такая ситуация встречается намного чаще, чем может показаться. Представим, например, большую таблицу, где каждая строка содержит интерактивный элемент, для взаимодействия с которым требуется определить обработчик события `click`. Механизм неявной итерации позволяет легко присоединить обработчики событий ко всем таким элементам, но производительность при этом может пострадать, потому что внутренняя реализация библиотеки jQuery все равно выполняет цикл и для поддержки всех обработчиков потребуется большой объем памяти.

Вместо этого мы можем присвоить один обработчик события `click` родительскому элементу DOM. Вследствие процесса всплытия событие `click` в конечном итоге достигнет родительского элемента, и мы получим возможность обработать его.

Для примера применим эту методику к нашему переключателю стилей (невзирая на то, что малое число элементов не требует этого). Как отмечалось выше, для определения, какой элемент находился под указателем мыши в момент щелчка, можно использовать свойство `event.target`.

```
$(document).ready(function() {
  $('#switcher').click(function(event) {
    if ($(event.target).is('.button')) {
      $('body').removeClass();
      if (event.target.id == 'switcher-narrow') {
        $('body').addClass('narrow');
      }
      else if (event.target.id == 'switcher-large') {
        $('body').addClass('large');
      }
```

```
        $('#switcher .button').removeClass('selected');
        $(event.target).addClass('selected');
        event.stopPropagation();
      }
    });
});
```

Здесь мы использовали новый метод с именем `.is()`. Он принимает **селекторные выражения**, которые мы рассматривали в предыдущей главе, и проверяет соответствие селектору текущего объекта jQuery. Если хотя бы один элемент множества соответствует селектору, метод `.is()` возвращает значение `true`. В данном случае выражение `$(event.target).is('.button')` проверяет, имеет ли элемент, на котором был выполнен щелчок, класс `button`. Если проверка дает положительный результат, выполняется тот же программный, что и прежде, за одним важным исключением: теперь ключевое слово `this` ссылается на элемент `<div id="switcher">`, поэтому всякий раз, когда требуется выяснить, на какой кнопке произошел щелчок, необходимо использовать свойство `event.target`.

Примечание

Мы могли бы проверить наличие класса у элемента также с помощью метода `.hasClass()`. Однако метод `.is()` обеспечивает большую гибкость и может выполнять проверку на соответствие любому селектору.

Однако такая реализация имеет один не предусмотренный побочный эффект. Теперь щелчок на кнопке будет приводить к сворачиванию переключателя, как это происходило ранее, до того как был добавлен вызов метода `.stopPropagation()`. Обработчик, управляющий видимостью переключателя, сейчас подключен к тому же элементу, что и обработчик для кнопок, поэтому остановка распространения события не предотвращает сворачивание переключателя. Чтобы обойти эту проблему, можно удалить вызов `.stopPropagation()` и добавить еще одну проверку `.is()`:

```
$(document).ready(function() {
  $('#switcher').click(function(event) {
    if (!$(event.target).is('.button')) {
      $('#switcher .button').toggleClass('hidden');
    }
  });
});
$(document).ready(function() {
  $('#switcher').click(function(event) {
    if ($(event.target).is('.button')) {
      $('body').removeClass();
      if (event.target.id == 'switcher-narrow') {
        $('body').addClass('narrow');
      }
```

```
      else if (event.target.id == 'switcher-large') {
        $('body').addClass('large');
      }
      $('#switcher .button').removeClass('selected');
      $(event.target).addClass('selected');
    }
  });
});
```

Для данного случая этот пример слишком сложен, но с ростом числа элементов, к которым требуется подключить обработчики событий, делегирование событий становится более предпочтительным приемом.

Примечание

Делегирование событий с успехом может применяться и в других ситуациях, с которыми мы познакомимся позднее, например при добавлении новых элементов методами **манипулирования деревом DOM** (глава 5) или процедурами **AJAX** (глава 6).

Удаление обработчика события

Существуют ситуации, когда бывает необходимо убрать обработчик события, зарегистрированный ранее. Это может быть обусловлено существенным изменением страницы, когда действие, предусматриваемое обработчиком, теряет смысл. Обычно подобные ситуации можно разрешить с помощью условных инструкций в обработчиках, но гораздо элегантнее вообще **отключить** обработчик.

Предположим, нам необходимо, чтобы наш переключатель стилей оставался развернутым всегда, когда используется стиль, отличный от обычного. Иначе говоря, когда выбрана кнопка Narrow Column или Large Print, щелчок в области переключателя не должен вызывать никаких действий. Реализовать это можно вызовом метода `.unbind()`, который отключит обработчик, сворачивающий переключатель при выполнении щелчка на одной из кнопок переключателя, определяющих стиль оформления, отличный от стиля по умолчанию.

```
$(document).ready(function() {
  $('#switcher').click(function(event) {
    if (!$(event.target).is('.button')) {
      $('#switcher .button').toggleClass('hidden');
    }
  });
  $('#switcher-narrow, #switcher-large').click(function() {
    $('#switcher').unbind('click');
  });
});
```

Теперь, когда пользователь щелкнет на кнопке, такой как Narrow Column, обработчик события click элемента переключателя стилей будет отключен, в результате чего щелчок мышью в области переключателя не будет приводить к его сворачиванию. Однако после этого кнопки перестанут работать! Кнопки тоже подключены к обработчику события click элемента переключателя стилей, поскольку мы переписали программный код с использованием в нем приема делегирования события. Это означает, что вызов метода $('#switcher').unbind('click') отключит и обработку событий в кнопках.

Пространство имен события

Нам необходимо придать вызову метода .unbind() более высокую избирательность, чтобы он не отключал оба зарегистрированных обработчика события click. Один из способов сделать это заключается в определении **пространства имен событий**. При подключении обработчика мы можем указать дополнительную информацию, которая позднее позволит идентифицировать отдельные обработчики. Чтобы задействовать пространство имен, необходимо вернуться к использованию универсального метода .bind() подключения обработчиков.

Первым параметром методу .bind() передается имя события JavaScript, которое требуется обработать. Однако здесь допускается использование специального синтаксиса, позволяющего определять подкатегории событий.

```
$(document).ready(function() {
  $('#switcher').bind('click.collapse', function(event) {
    if (!$(event.target).is('.button')) {
      $('#switcher .button').toggleClass('hidden');
    }
  });
  $('#switcher-narrow, #switcher-large').click(function() {
    $('#switcher').unbind('click.collapse');
  });
});
```

Расширение .collapse недоступно системе обработки событий; события click обрабатываются данной функцией точно так же, как если бы мы записали вызов .bind('click'). Однако добавление пространства имен означает, что мы можем отключать именно этот обработчик, не оказывая влияния на обработчик события click для кнопок.

Примечание

Существуют и другие способы повышения избирательности метода .unbind(), с которыми мы познакомимся чуть дальше. Однако такой инструмент, как определение пространств имен для событий, полезно иметь в своем арсенале. Он особенно удобен при создании **модулей расширения**, в чем мы убедимся в главе 11.

Повторное подключение событий

Теперь щелчок на кнопке Narrow Column или Large Print отключает возможность сворачивания переключателя стилей. Однако нам требуется вернуть эту возможность в случае щелчка на кнопке Default. Другими словами, необходимо **повторно подключить** обработчик по щелчку на кнопке Default.

Сначала нам следует присвоить имя функции обработчика, чтобы ее можно было использовать многократно, не повторяя ее программный код:

```
$(document).ready(function() {
  var toggleStyleSwitcher = function(event) {
    if (!$(event.target).is('.button')) {
      $('#switcher .button').toggleClass('hidden');
    }
  };
  $('#switcher').bind('click.collapse', toggleStyleSwitcher);
});
```

Обратите внимание, что в этом примере используется новый синтаксис определения функции. Вместо определения функции с помощью ключевого слова `function` здесь выполняется присваивание **анонимной функции локальной переменной**. Такой способ определения обработчика больше напоминает определение других функций; функционально оба варианта являются идентичными.

Кроме того, метод `.bind()`во втором аргументе принимает **ссылку на функцию**. Важно запомнить, что при использовании здесь именованной функции следует опускать круглые скобки после имени функции; круглые скобки приведут к немедленному *вызову* функции, тогда как нам требуется всего лишь передать *ссылку* на нее.

Теперь, когда функция имеет имя, мы сможем подключить ее снова позднее, не повторяя определение функции:

```
$(document).ready(function() {
  var toggleStyleSwitcher = function(event) {
    if (!$(event.target).is('.button')) {
      $('#switcher .button').toggleClass('hidden');
    }
  };

  $('#switcher').bind('click.collapse', toggleStyleSwitcher);
  $('#switcher-narrow, #switcher-large').click(function() {
    $('#switcher').unbind('click.collapse');
  });
  $('#switcher-default').click(function() {
    $('#switcher')
      .bind('click.collapse', toggleStyleSwitcher);
  });
});
```

Теперь функция свертывания/развертывания подключается к переключателю во время загрузки документа, отключается в случае щелчка на кнопке Narrow Column или Large Print и повторно подключается в случае щелчка на кнопке Default.

Мы сумели обойти здесь потенциальную ошибку. Вспомним, что когда обработчик подключается к событию с помощью jQuery, предыдущий обработчик продолжает действовать. Казалось бы, это означает, что в случае выполнения нескольких щелчков на кнопке Default будет подключено несколько копий обработчика `toggleStyleSwitcher`, что приведет к необычному поведению при щелчке в пределах элемента `<div>`. В действительности, если бы мы использовали анонимную функцию, то так оно и было бы. Но поскольку мы присвоили функции имя и все время используем одну и ту же функцию, эта функция будет подключена всего один раз. Метод `.bind()` не выполняет повторное подключение обработчика к элементу, если он уже подключен.

Еще одно преимущество использования именованной функции заключается в том, что нам больше не требуется использовать определение пространства имен для событий. Метод `.unbind()` может принимать функцию во втором аргументе, в этом случае он отключает только указанный обработчик.

```
$(document).ready(function() {
  var toggleStyleSwitcher = function(event) {
    if (!$(event.target).is('.button')) {
      $('#switcher .button').toggleClass('hidden');
    }
  };

  $('#switcher').click(toggleStyleSwitcher);
  $('#switcher-narrow, #switcher-large').click(function() {
    $('#switcher').unbind('click', toggleStyleSwitcher);
  });
  $('#switcher-default').click(function() {
    $('#switcher').click(toggleStyleSwitcher);
  });
});
```

Для ситуации, когда обработчик события должен отключаться после первого срабатывания, имеется специализированный метод подключения. Этот метод называется `.one()`, а порядок его использования приводится ниже:

```
$(document).ready(function() {
  $('#switcher').one('click', toggleStyleSwitcher);
});
```

Этот программный код позволил бы выполнить операцию переключения всего один раз.

Имитация действий пользователя

Иногда бывает необходимо выполнить программный код, который подключен к обработке события, даже если это событие не возникает. Например, предположим, нам требуется, чтобы переключатель стилей изначально находился в свернутом состоянии. Этого можно было бы добиться, скрыв кнопки с помощью таблицы стилей или вызвав метод `.hide()` из обработчика `$(document).ready()`. Другой способ заключается в имитации щелчка в пределах переключателя стилей, вызывающего срабатывание уже установленного механизма переключения.

Осуществить это можно с помощью метода `.trigger()`:

```
$(document).ready(function() {
  $('#switcher').trigger('click');
});
```

Теперь переключатель стилей сворачивается сразу после загрузки страницы (рис. 3.11), как если бы в его пределах был выполнен щелчок мышью. Если бы нам потребовалось предусмотреть сокрытие содержимого, когда в броузере отключена поддержка JavaScript, было бы разумно использовать прием **постепенной деградации** функциональных возможностей.

Рис. 3.11. *Переключатель стилей сворачивается сразу после загрузки страницы*

Для метода `.trigger()` предоставляется тот же набор сокращенных аналогов, что и для метода `.bind()`. При вызове этих аналогов без аргументов вместо подключения обработчика запускается событие:

```
$(document).ready(function() {
  $('#switcher').click();
});
```

События клавиатуры

Нашему переключателю стилей мы можем также добавить возможность реагировать на комбинации горячих клавиш. Когда пользователь будет вводить первый символ отображаемого названия стиля, мы заставим страницу вести себя так, как если бы был выполнен щелчок на соответствующей кнопке. Для реализации этой особенности нам не-

обходимо познакомиться с **событиями клавиатуры**, которые ведут себя несколько иначе, чем **события мыши**.

Существует два типа событий клавиатуры: те, что являются реакцией непосредственно на действия с клавиатурой (keyup и keydown), и те, что являются реакцией на ввод текста (keypress). Один символ может быть результатом нажатия нескольких клавиш, например одновременное нажатие клавиши Shift и клавиши X выводит заглавный символ **X**. Несмотря на то, что в каждом броузере работа с клавиатурой реализована по-разному (что совсем не удивительно), можно уверенно следовать следующему правилу: если требуется узнать, какая клавиша была нажата, необходимо использовать событие keyup или keydown; если требуется узнать, какой символ появится на экране, необходимо использовать событие keypress. Для рассматриваемого случая нам достаточно знать, какая из трех клавиш, D, N или L, была нажата, поэтому мы будем использовать событие keyup.

Далее нам необходимо определить, какой элемент должен реагировать на событие. Это не так очевидно, как в случае событий мыши, когда имеется курсор, явно указывающий на требуемый элемент. В случае события клавиатуры его обработкой должен заниматься элемент, который в момент события обладает **фокусом ввода**. Перемещение фокуса ввода между элементами может выполняться несколькими способами, включая щелчки мышью и нажатия на клавишу табуляции. Не все элементы способны принять фокус ввода, а только те, для которых определена реакция по умолчанию на события клавиатуры, например поля форм, ссылки и элементы, обладающие свойством .tabIndex.

В данном случае нас, вообще говоря, не волнует, какой элемент обладает фокусом ввода; нам лишь требуется, чтобы наш переключатель выполнял соответствующие действия, когда пользователь нажимает одну из клавиш. Здесь нам на помощь снова приходит механизм всплытия событий, благодаря которому мы можем присоединить наш обработчик события keyup к элементу document и пребывать в уверенности, что любое событие клавиатуры достигнет его.

Наконец, в обработчике события keyup нам необходимо определить, какая клавиша была нажата. Для этого мы воспользуемся объектом события. Свойство .keyCode события содержит идентификатор нажатой клавиши, причем для алфавитных клавиш он представляет собой символ ASCII в верхнем регистре. Благодаря этому мы можем определять полученное значение и вызывать соответствующий обработчик щелчка мышью:

```
$(document).ready(function() {
  $(document).keyup(function(event) {
    switch (String.fromCharCode(event.keyCode)) {
      case 'D':
        $('#switcher-default').click();
        break;
```

```
      case 'N':
        $('#switcher-narrow').click();
        break;
      case 'L':
        $('#switcher-large').click();
        break;
    }
  });
});
```

Теперь нажатие одной из трех указанных клавиш будет имитировать щелчок мышью на соответствующей кнопке, при условии, что события клавиатуры не перехватываются такими механизмами, как механизм «поиска по мере ввода», присутствующий в броузере Firefox.

В качестве альтернативы использованию метода `.trigger()` для имитации щелчка мышью рассмотрим способ, как можно выделить программный код в функцию, чтобы его можно было вызывать из нескольких обработчиков, в данном случае – из обработчиков событий `click` и `keyup`. Хотя в рассматриваемой ситуации в этом нет необходимости, такой прием может пригодиться для уменьшения объема программного кода.

```
$(document).ready(function() {
  // Активировать реакцию кнопок переключателя стилей на событие
  // наведения указателя мыши.
  $('#switcher .button').hover(function() {
    $(this).addClass('hover');
  }, function() {
    $(this).removeClass('hover');
  });

  // Разрешить разворачивание и сворачивание переключателя стилей.
  var toggleStyleSwitcher = function(event) {
    if (!$(event.target).is('.button')) {
      $('#switcher .button').toggleClass('hidden');
    }
  };
  $('#switcher').click(toggleStyleSwitcher);

  // Имитировать щелчок мышью, чтобы изначально переключатель
  // находился в свернутом состоянии.
  $('#switcher').click();
  // Функция setBodyClass() изменяет стиль страницы.
  // Она изменяет также состояние переключателя стилей.
  var setBodyClass = function(className) {
    $('body').removeClass();
    $('body').addClass(className);
    $('#switcher .button').removeClass('selected');
    $('#switcher-' + className).addClass('selected');

    if (className == 'default') {
      $('#switcher').click(toggleStyleSwitcher);
```

```
    }
    else {
      $('#switcher').unbind('click', toggleStyleSwitcher);
      $('#switcher .button').removeClass('hidden');
    }
  };

  // Вызвать setBodyClass() в случае щелчка на кнопке.
  $('#switcher').click(function(event) {
    if ($(event.target).is('.button')) {
      if (event.target.id == 'switcher-default') {
        setBodyClass('default');
      }
      if (event.target.id == 'switcher-narrow') {
        setBodyClass('narrow');
      }
      else if (event.target.id == 'switcher-large') {
        setBodyClass('large');
      }
    }
  });

  // Вызвать setBodyClass() в случае нажатия на клавишу.
  $(document).keyup(function(event) {
    switch (String.fromCharCode(event.keyCode)) {
      case 'D':
        setBodyClass('default');
        break;
      case 'N':
        setBodyClass('narrow');
        break;
      case 'L':
        setBodyClass('large');
        break;
    }
  });
});
```

В заключение

Возможности, рассмотренные нами в этой главе, позволяют:

- Обеспечивать совместное сосуществование нескольких библиотек JavaScript в одной странице с помощью метода `.noConflict()`.
- Использовать **обработчики событий мыши** для реализации отклика элементов страницы на щелчок мышью с помощью методов `.bind()` или `.click()`.
- Использовать **контекст события** для выполнения различных действий в зависимости от того, на каком элементе страницы был вы-

полнен щелчок, причем даже когда один и тот же обработчик подключен к нескольким элементам.

- Поочередно разворачивать и сворачивать элемент страницы с помощью метода `.toggle()`.
- Визуально выделять элементы, находящиеся под указателем мыши, с помощью метода `.hover()`.
- Использовать факт **распространения события** для определения, какие элементы должны откликаться на событие, с помощью методов `.stopPropagation()` и `.preventDefault()`.
- Использовать прием **делегирования события** для уменьшения числа подключенных обработчиков в странице.
- Вызывать метод `.unbind()` для отключения обработчика события по мере необходимости.
- Выделять взаимосвязанные обработчики событий с помощью **пространства имен событий**, чтобы можно было действовать на них, как на группу.
- Обеспечивать вызов обработчиков событий с помощью метода `.trigger()`.
- Использовать **обработчики событий клавиатуры** для реализации отклика на нажатие клавиши пользователем с помощью метода `.keyup()`.

При совместном использовании всех этих возможностей мы можем создавать действительно интерактивные страницы. В следующей главе мы узнаем, как обеспечивать обратную визуальную связь с пользователем в ходе взаимодействий.

4

Эффекты

Как известно, дела звучат громче слов, а в мире JavaScript эффекты заставляют дела звучать еще громче. С помощью библиотеки jQuery мы легко можем усилить наши действия, используя набор простых визуальных **эффектов** и даже создавая наши собственные, более сложные **анимационные эффекты.**

Эффекты, реализованные в библиотеке jQuery, безусловно, добавляют привлекательности, как, например, в случае постепенного появления элементов в поле зрения, а не всех сразу. Кроме этого, они могут значительно повысить удобство работы со страницей и помочь пользователю ориентироваться в изменениях, происходящих на ней (что особенно справедливо для приложений с поддержкой технологии AJAX). В данной главе мы рассмотрим ряд этих эффектов и интересные способы их объединения.

Изменение встроенных свойств стиля CSS

Прежде чем перейти к обсуждению эффектов, реализованных в библиотеке jQuery, кратко рассмотрим принципы работы со стилями CSS. В предыдущих главах мы изменяли внешний вид документа, определяя классы стилей в отдельных таблицах стилей, а затем добавляя или удаляя эти классы средствами библиотеки jQuery. Обычно это наиболее предпочтительный способ внедрения стилей CSS в разметку HTML, потому что он усиливает роль таблиц стилей в оформлении визуального представления таблиц. Однако иногда бывают ситуации, когда возникает необходимость применить стили, которые отсутствуют, или не так-то просто определить в таблицах стилей. К счастью, для решения подобных задач библиотека jQuery предлагает метод `.css()`.

Данный метод действует и как метод **записи**, и как метод **чтения**. Чтобы получить значение свойства стиля, достаточно просто передать методу имя этого свойства в виде строки, например `.css('backgroundColor')`.

Сложносоставные имена свойств могут передаваться методу в нотации CSS, когда слова в имени свойства отделяются дефисом (`background-color`), или в нотации DOM, когда каждое следующее слово в имени начинается с символа верхнего регистра (`backgroundColor`). Для записи значений свойств стилей существуют две версии метода `.css()`: одна из них принимает одно свойство и его значение, а вторая принимает **отображение** пар свойство-значение:

```
.css('property','value')
.css({property1: 'value1', 'property-2': 'value2'})
```

Опытные разработчики на JavaScript без труда узнают в этих отображениях jQuery **литералы объектов** JavaScript.

Примечание

Числовые значения не должны окружаться кавычками, тогда как строковые значения должны. При использовании версии метода, принимающей отображения, имена свойств в нотации DOM могут не заключаться в кавычки.

Метод `.css()` мы можем использовать точно так же, как мы использовали метод `.addClass()`: он может объединяться в **цепочку** с селектором или **подключаться** к событию. Для демонстрации вышесказанного мы вновь вернемся к примеру переключателя стилей из главы 3, но разметка HTML будет несколько иной:

```
<div id="switcher">
  <div class="label">Text Size</div>
  <button id="switcher-default">Default</button>
  <button id="switcher-large">Bigger</button>
  <button id="switcher-small">Smaller</button>
</div>
<div class="speech">
  <p>Fourscore and seven years ago our fathers brought forth
     on this continent a new nation, conceived in liberty,
     and dedicated to the proposition that all men are created
     equal.</p>
</div>
```

За счет присоединения таблицы стилей с некоторыми основными правилами оформления изначально страница может выглядеть так, как показано на рис. 4.1.

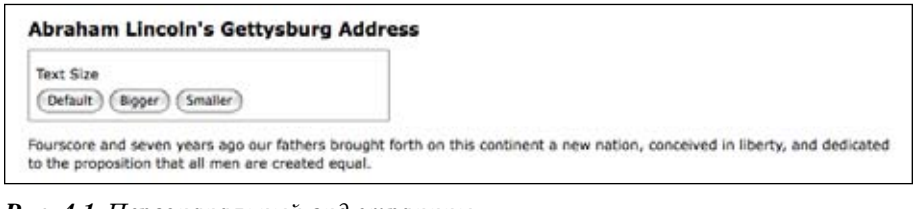


***Рис. 4.1.** Первоначальный вид страницы*

В этой версии переключателя стилей мы используем элементы <button>. Щелчок на кнопках Bigger (крупнее) и Smaller (мельче) будет соответственно увеличивать и уменьшать размер шрифта, которым отображается текст в элементе , а щелчок на кнопке Default (по умолчанию) будет восстанавливать размер шрифта для элемента в его первоначальное значение.

Если бы нам требовалось однократно изменять размер шрифта, устанавливая его в некоторое предопределенное значение, мы могли бы по-прежнему использовать метод .addClass(). Но предположим, что теперь нам необходимо, чтобы размер шрифта продолжал увеличиваться или уменьшаться с каким-то шагом при каждом щелчке на соответствующей кнопке. Теоретически вполне возможно определить для каждого щелчка отдельный класс CSS и выполнять итерации по ним, однако самый простой способ заключается в том, чтобы каждый раз вычислять новый размер шрифта, получая текущее его значение и увеличивая его на некоторый коэффициент (например, на 40%).

Наш программный код начнется с определения обработчиков событий $(document).ready() и $('#switcher-large').click():

```
$(document).ready(function() {
  $('#switcher-large').click(function() {
  });
});
```

Далее размер шрифта легко можно определить с помощью метода .css(): $('div.speech').css('fontSize'). Однако возвращаемое значение содержит в конце символы 'px', поэтому, прежде чем использовать полученное значение в вычислениях, нам необходимо удалить эти символы. Кроме того, если объект jQuery планируется использовать многократно, предпочтительнее будет **кэшировать** селектор, сохранив полученный объект jQuery в переменной.

```
$(document).ready(function() {
  var $speech = $('div.speech');
  $('#switcher-large').click(function() {
    var num = parseFloat($speech.css('fontSize'), 10);
  });
});
```

Теперь первая строка внутри $(document).ready() сохраняет объект jQuery с элементом в переменной. Обратите внимание на символ $ в имени переменной $speech. Так как JavaScript допускает использование символа $ в именах переменных, мы можем применять его в качестве напоминания, что переменная хранит объект jQuery.

Внутри обработчика .click() мы используем функцию parseFloat(), чтобы получить значение свойства fontSize в виде числа. Функция parseFloat() просматривает строку слева направо, пока не встретит первый нецифровой символ. Затем строка цифр преобразуется в вещественное (десятич-

ное) число. Например, эта функция преобразует строку '12' в число 12. Кроме того, она удаляет все нецифровые символы в конце строки, благодаря чему строка '12px' превращается в число 12. Если строка начинается с нецифрового символа, функция parseFloat() возвращает значение NaN (**Not a Number** – нечисло). Второй аргумент функции parseFloat() позволяет указать, что строка должна интерпретироваться не как восьмеричное или какое-либо другое число, а как десятичное.

Все, что осталось теперь сделать для увеличения размера шрифта на 40%, – это просто умножить число num на 1.4 и затем установить новый размер шрифта, присоединив к числу num символы 'px':

```
$(document).ready(function() {
  var $speech = $('div.speech');
  $('#switcher-large').click(function() {
    var num = parseFloat($speech.css('fontSize'), 10 );
    num *= 1.4;
    $speech.css('fontSize', num + 'px');
  });
});
```

Примечание

Выражение num *= 1.4 представляет собой сокращенную форму записи num = num * 1.4. Аналогичные сокращенные формы записи можно использовать и для других основных арифметических операторов: сложения, num += 1.4; вычитания, num -= 1.4; деления, num /= 1.4; и деления по модулю (остатка от деления), num %= 1.4.

Теперь, когда пользователь щелкнет на кнопке Bigger, размер шрифта текста увеличится. Следующий щелчок на этой кнопке увеличит размер шрифта еще больше, как показано на рис. 4.2.

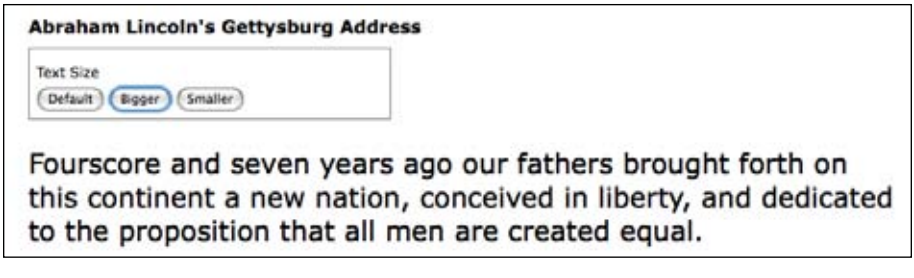


Рис. 4.2. *После щелчка на кнопке Bigger (крупнее) размер шрифта увеличивается*

Чтобы в результате щелчка на кнопке Smaller размер шрифта уменьшался, мы будем уже не умножать, а делить текущее значение размера: num /= 1.4. По-прежнему лучше всего объединить обработчики .click() всех элементов <button>, находящихся внутри . Тогда пос-

ле получения числового значения можно будет выполнить операцию умножения или деления в зависимости от значения атрибута ID кнопки, на которой был выполнен щелчок. Ниже показано, как теперь выглядит наш программный код:

```
$(document).ready(function() {
  var $speech = $('div.speech');
  $('#switcher button').click(function() {
    var num = parseFloat( $speech.css('fontSize'), 10 );
    if (this.id == 'switcher-large') {
      num *= 1.4;
    } else if (this.id == 'switcher-small') {
      num /= 1.4;
    }
    $speech.css('fontSize', num + 'px');
  });
});
```

В главе 3 говорилось, что доступ к свойству `id` элемента DOM можно получить с помощью ключевого слова `this`, которое присутствует в инструкциях `if` и `else if`. В данном случае этот способ является более эффективным, чем создание объекта jQuery для простой проверки значения свойства.

Кроме всего прочего, не лишним было бы иметь возможность возвращать размер шрифта в его первоначальное значение. Чтобы дать пользователю такую возможность, мы можем просто сохранить размер шрифта в переменной сразу же, как только дерево DOM будет готово к работе с ним. После этого мы сможем задействовать это значение в обработчике щелчка на кнопке Default. Чтобы обработать данное событие, мы могли бы добавить еще одну инструкцию `else if`. Однако инструкция `switch` лучше подходит для таких случаев.

```
$(document).ready(function() {
  var $speech = $('div.speech');
  var defaultSize = $speech.css('fontSize');
  $('#switcher button').click(function() {
    var num = parseFloat( $speech.css('fontSize'), 10 );
    switch (this.id) {
      case 'switcher-large':
        num *= 1.4;
        break;
      case 'switcher-small':
        num /= 1.4;
        break;
      default:
        num = parseFloat(defaultSize, 10);
    }
    $speech.css('fontSize', num + 'px');
  });
});
```

Здесь мы точно так же проверяем значение this.id и в соответствии с ним изменяем размер шрифта, но если это значение не равно ни 'switcher-large', ни 'switcher-small', то устанавливается размер шрифта по умолчанию.

Простые эффекты скрытия и отображения

Простые методы .hide() и .show() при вызове без параметров могут использоваться как интеллектуальные сокращенные версии метода .css('display','string'), где 'string' – это значение свойства display. В результате, как и следует ожидать, соответствующий набор элементов будет немедленно скрыт или отображен без применения каких-либо анимационных эффектов.

Метод .hide() устанавливает **встроенный атрибут style** в значение display:none. Интеллектуальность заключается в том, что этот метод запоминает значение свойства display, обычно block или inline, прежде чем оно будет замещено значением none. В свою очередь, метод .show() восстанавливает значение свойства display соответствующего набора элементов, имевшееся до того, как было установлено значение display:none.

Примечание

Дополнительную информацию о свойстве display и о том, как его значения влияют на визуальное представление элементов веб-страницы, можно найти на сайте Mozilla Developer Center по адресу: *https://developer.mozilla.org/en/CSS/display/*, а примеры – по адресу: *https://developer.mozilla.org/samples/cssref/display.html*.

Эту особенность методов .show() и .hide() весьма удобно использовать для сокрытия элементов, для которых значение по умолчанию свойства display переопределяется в таблицах стилей. Допустим, по умолчанию элемент <li> имеет значение свойства display:block, но в случае оформления горизонтального меню мы можем изменить его на display:inline. К счастью, применение метода .show() к скрытым элементам, например к одному из таких тегов <li>, не просто сбрасывает свойство в значение по умолчанию display:block, что могло бы привести к размещению каждого элемента <li> в отдельной строке, а восстанавливает предыдущее значение display:inline, сохраняя тем самым горизонтальное расположение пунктов меню.

Использование этих двух методов можно коротко продемонстрировать на примере присоединения второго параграфа и ссылки «read more» (читать дальше) после первого параграфа в следующем примере разметки HTML:

```
<div id="switcher">
  <div class="label">Text Size</div>
  <button id="switcher-default">Default</button>
```

```
    <button id="switcher-large">Bigger</button>
    <button id="switcher-small">Smaller</button>
  </div>
  <div class="speech">
    <p>Fourscore and seven years ago our fathers brought forth
       on this continent a new nation, conceived in liberty,
       and dedicated to the proposition that all men are
       created equal.
    </p>
    <p>Now we are engaged in a great civil war, testing whether
       that nation, or any nation so conceived and so dedicated,
       can long endure. We are met on a great battlefield of
       that war. We have come to dedicate a portion of that
       field as a final resting-place for those who here gave
       their lives that the nation might live. It is altogether
       fitting and proper that we should do this. But, in a
       larger sense, we cannot dedicate, we cannot consecrate,
       we cannot hallow, this ground.
    </p>
    <a href="#" class="more">read more</a>
  </div>
```

Когда дерево DOM готово к работе, второй параграф скрывается:

```
$(document).ready(function() {
  $('p:eq(1)').hide();
});
```

А текст речи выглядит, как показано на рис. 4.3.

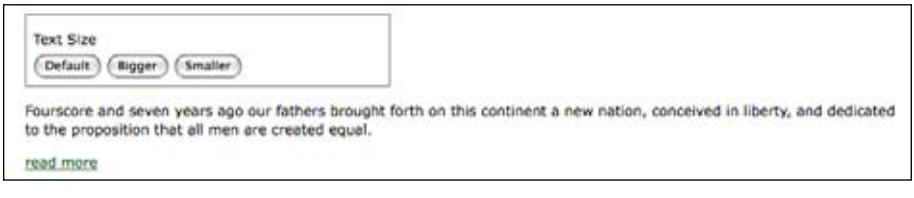


Рис. 4.3. *Вид страницы сразу после загрузки*

Затем, когда пользователь щелкает на ссылке read more (читать дальше), находящейся в конце первого параграфа, она скрывается, и отображается второй параграф:

```
$(document).ready(function() {
  $('p:eq(1)').hide();
  $('a.more').click(function() {
    $('p:eq(1)').show();
    $(this).hide();
    return false;
  });
});
```

Обратите внимание, как с помощью инструкции `return false` предотвращается выполнение **действия по умолчанию** для ссылки. Теперь страница с текстом речи выглядит, как показано на рис. 4.4.

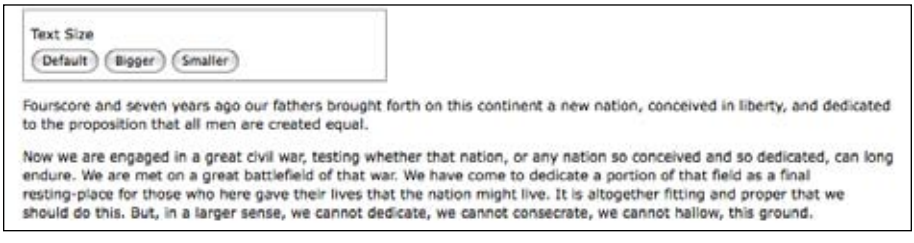


Рис. 4.4. *Вид страницы после щелчка на ссылке «read more» (читать дальше)*

Методы `.hide()` и `.show()` очень быстрые и удобные, но они не обладают особой эффектностью. Чтобы придать ее, мы можем определить для этих методов скорость выполнения.

Эффекты и скорость выполнения

Когда для методов `.show()` или `.hide()` определяется **скорость** (или, точнее говоря, **продолжительность**) воспроизведения эффекта, они становятся анимированными, то есть действие эффекта растягивается на указанный период времени. Например, метод `.hide('speed')` одновременно уменьшает высоту, ширину и непрозрачность элемента, пока все три значения не станут равными нулю – в этот момент применяется правило CSS `display:none`. Метод `.show('speed')` будет увеличивать высоту элемента сверху вниз, ширину слева направо и непрозрачность от 0 до 1, пока его содержимое не станет видимо полностью.

Скорость

Для любых эффектов jQuery мы можем использовать три предустановленных значения скорости: `'slow'` (медленно), `'normal'` (нормальная скорость) и `'fast'` (быстро). Вызов `.show('slow')` будет воспроизводить эффект в течение 0,6 секунды, `.show('normal')` – в течение 0,4 секунды, и `.show('fast')` – в течение 0,2 секунды. Для достижения большей точности можно указывать число миллисекунд, например `.show(850)`. В отличие от символических имен, обозначающих скорость, числа не должны заключаться в кавычки.

Давайте добавим скорость воспроизведения эффекта в наш пример с отображением второго параграфа **Геттисбергского послания**, знаменитой речи президента Линкольна:

```
$(document).ready(function() {
  $('p:eq(1)').hide();
  $('a.more').click(function() {
```

```
    $('p:eq(1)').show('slow');
    $(this).hide();
    return false;
  });
});
```

Если зафиксировать состояние параграфа примерно в середине воспроизведения эффекта, можно наблюдать картину, показанную на рис. 4.5.

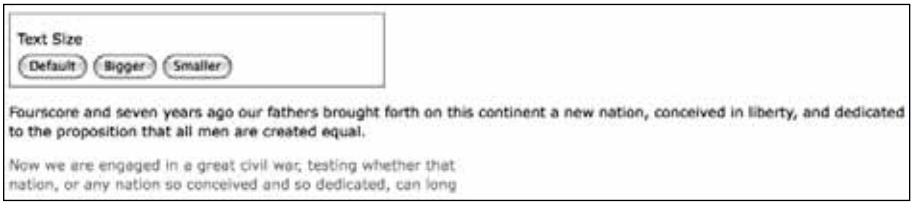


Рис. 4.5. *Состояние параграфа в середине воспроизведения эффекта show()*

Эффекты проявления и растворения

Несмотря на то, что анимированные методы `.show()` и `.hide()` определенно обладают эффектностью, иногда она может оказаться излишней. К счастью, библиотека jQuery предлагает пару других встроенных и более тонких анимационных эффектов. Например, чтобы обеспечить появление параграфа только за счет плавного увеличения непрозрачности, можно воспользоваться методом `.fadeIn('slow')`:

```
$(document).ready(function() {
  $('p:eq(1)').hide();
  $('a.more').click(function() {
    $('p:eq(1)').fadeIn('slow');
    $(this).hide();
    return false;
  });
});
```

На этот раз, если зафиксировать состояние параграфа примерно в середине воспроизведения эффекта, можно наблюдать картину, показанную на рис. 4.6.

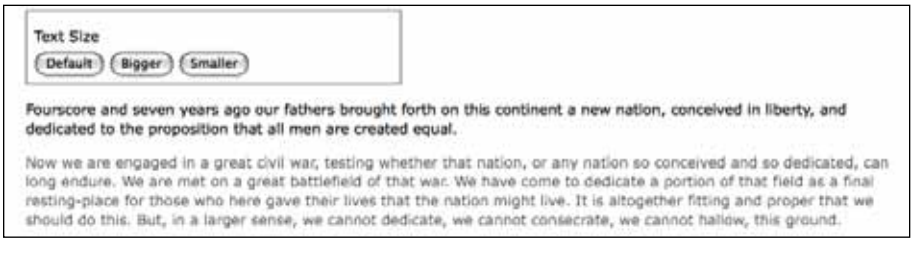


Рис. 4.6. *Состояние параграфа в середине воспроизведения эффекта fadeIn()*

Отличие заключается в том, что эффект, воспроизводимый методом .fadeIn(), начинается с установки нормального размера параграфа, чтобы его содержимое просто постепенно проявлялось в нем. Для плавного уменьшения непрозрачности можно использовать метод .fadeOut().

Составные эффекты

Иногда вместо однократного изменения отображения элементов, как в предыдущем примере, бывает необходимо переключение их видимости. Эффекта переключения можно добиться, если на первом этапе проверить видимость соответствующего набора элементов, а затем вызвать подходящий метод. Снова используя эффекты проявления и растворения, мы можем изменить пример сценария, как показано ниже:

```
$(document).ready(function() {
  var $firstPara = $('p:eq(1)');
  $firstPara.hide();
  $('a.more').click(function() {
    if ($firstPara.is(':hidden')) {
      $firstPara.fadeIn('slow');
      $(this).text('read less');
    } else {
      $firstPara.fadeOut('slow');
      $(this).text('read more');
    }
    return false;
  });
});
```

Как и ранее в этой главе, мы применяем **кэширование** селектора, чтобы избежать необходимости повторной операции обхода дерева DOM. Примечательно также, что на этот раз мы не скрываем ссылку, на которой был произведен щелчок, а просто изменяем ее текст.

Применение условной инструкции if else является наиболее очевидным способом управления видимостью элемента. Но при использовании **составных эффектов** из библиотеки jQuery мы можем полностью отказаться от него (хотя в данном конкретном примере условная инструкция все равно необходима для реализации изменения текста ссылки). Библиотека jQuery предоставляет метод .toggle(), который действует так же, как методы .show() и .hide(), и, подобно им способен принимать аргумент, определяющий скорость вспроизведения эффекта. В качестве еще одного примера метода воспроизведения составного эффекта можно привести метод .slideToggle(), который отображает или скрывает элементы, постепенно увеличивая или уменьшая их высоту. Ниже показано, как выглядит сценарий, в котором используется метод .slideToggle():

```
$(document).ready(function() {
var $firstPara = $('p:eq(1)');
  $firstPara.hide();
```

```
  $('a.more').click(function() {
    $firstPara.slideToggle('slow');
    var $link = $(this);
    if ( $link.text() == "read more" ) {
      $link.text('read less');
    } else {
      $link.text('read more');
    }
    return false;
  });
});
```

На этот раз, чтобы избежать повторного использования выражения `$(this)` и тем самым повысить скорость выполнения и удобочитаемость сценария, мы сохраняем результат этого выражения в переменной `$link`. Кроме того, условная инструкция проверяет текст ссылки, а не видимость второго параграфа, потому что она используется только для изменения текста.

Создание собственных анимационных эффектов

Помимо методов, реализующих встроенные анимационные эффекты, библиотека jQuery предоставляет мощный метод `.animate()`, позволяющий нам создавать свои **собственные анимационные эффекты** с возможностью тонкого управления ими. Метод `.animate()` обеспечивает два варианта использования. В первом он принимает до четырех аргументов:

1. **Отображение** с набором свойств и значений стилей, напоминающее отображение, используемое при вызове метода `.css()`, обсуждавшегося выше в этой главе.
2. Необязательный параметр **скорости**, который может быть одной из предопределенных строк или числом, указывающим продолжительность эффекта в миллисекундах.
3. Необязательный **тип перехода** – этот дополнительный параметр будет рассматриваться в главе 10.
4. Необязательную **функцию обратного вызова** – об этом аргументе будет рассказываться ниже в этой же главе.

Вызов метода со всеми четырьмя аргументами выглядит, как показано ниже:

```
.animate({property1: 'value1', property2: 'value2'},
  speed, easing, function() {
    alert('The animation is finished.');
  }
);
```

Вторая форма метода принимает два аргумента: отображение со свойствами и отображение с параметрами.

```
.animate({properties}, {options})
```

В действительности второй аргумент в этом случае заключает в себе со второго по четвертый аргументы первой формы вызова метода и добавляет еще два параметра. Если разбить аргументы на отдельные строки для большей удобочитаемости, вторая форма вызова метода будет выглядеть, как показано ниже:

```
.animate({
  property1: 'value1',
  property2: 'value2'
}, {
  duration: 'value',
  easing: 'value',
  complete: function() {
    alert('The animation is finished.');
  },
  queue: boolean,
  step: callback
});
```

Пока что мы будем использовать первую форму вызова метода `.animate()`, но мы еще вернемся ко второй форме ниже в этой же главе, когда будем рассматривать вопрос постановки эффектов в очередь.

Переключение эффекта проявления/растворения

Заметили ли вы при обсуждении составных эффектов, что не для всех методов существуют соответствующие методы переключения? Это действительно так: для методов выдвижения имеется соответствующий метод `.slideToggle()`, однако нет метода `.fadeToggle()`, соответствующего методам `.fadeIn()` и `.fadeOut()`! Но несмотря на это, мы легко можем реализовать собственную версию метода переключения эффектов проявления и растворения с помощью метода `.animate()`. Во фрагменте ниже мы заменили строку с вызовом метода `.slideToggle()` из предыдущего примера своей собственной реализацией анимационного эффекта:

```
$(document).ready(function() {
  $('p:eq(1)').hide();
  $('a.more').click(function() {
    $('p:eq(1)').animate({opacity: 'toggle'}, 'slow');
    var $link = $(this);
    if ( $link.text() == "read more" ) {
      $link.text('read less');
    } else {
      $link.text('read more');
    }
    return false;
  });
});
```

Этот пример демонстрирует, что метод .animate() позволяет указывать **сокращенные формы значений** для свойств CSS – 'show', 'hide' и 'toggle', что упрощает реализацию эффектов в случае, когда существующие *методы* не подходят для решения поставленной задачи.

Управление сразу несколькими свойствами

С помощью метода .animate() можно изменять значения любых комбинаций свойств одновременно. Например, чтобы создать эффект одновременного выдвижения и проявления второго параграфа, можно просто добавить пару свойство-значение для свойства height в отображение свойств при вызове метода .animate():

```
$(document).ready(function() {
  $('p:eq(1)').hide();
  $('a.more').click(function() {
    $('p:eq(1)').animate({
      opacity: 'toggle',
      height: 'toggle'
    },
    'slow');
    var $link = $(this);
    if ( $link.text() == "read more" ) {
      $link.text('read less');
    } else {
      $link.text('read more');
    }
    return false;
  });
});
```

Кроме свойств стиля, для управления которыми имеются сокращенные формы методов, мы можем управлять и другими свойствами, такими как: left, top, fontSize, margin, padding и borderWidth. Вернемся к сценарию, который увеличивает размер шрифта текста речи. Мы можем анимировать увеличение и уменьшение размера шрифта простой заменой метода .css() методом .animate():

```
$(document).ready(function() {
  var $speech = $('div.speech');
  var defaultSize = $speech.css('fontSize');
  $('#switcher button').click(function() {
    var num = parseFloat( $speech.css('fontSize'), 10 );
    switch (this.id) {
      case 'switcher-large':
        num *= 1.4;
        break;
      case 'switcher-small':
        num /= 1.4;
        break;
```

```
        default:
          num = parseFloat(defaultSize, 10);
      }
      $speech.animate({fontSize: num + 'px'}, 'slow');
    });
});
```

Использование дополнительных свойств позволяет создавать гораздо более сложные эффекты. Например, можно перемещать элемент по странице слева направо и одновременно увеличивать его высоту на 20 пикселов и ширину рамки до 5 пикселов.

Попробуем воспроизвести этот эффект для элемента `<div id="switcher">`. На рис. 4.7 показано, как выглядит страница до воспроизведения эффекта.

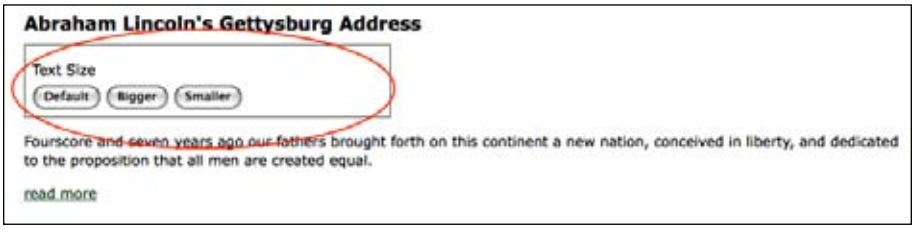


Рис. 4.7. *Вид страницы до воспроизведения эффекта*

Учитывая, что размер окна броузера может изменяться, нам необходимо вычислить расстояние, которое должен преодолеть элемент, прежде чем он достигнет правого края страницы. Предполагая, что параграф имеет ширину 100%, мы можем вычесть ширину блока с меткой Text Size (размер текста) из ширины параграфа. Обычно в таких вычислениях используется метод `.width()` из библиотеки jQuery, но он не учитывает ширину отступов слева и справа и ширину левой границы блока. Однако, начиная с версии jQuery 1.2.6, в нашем распоряжении имеется метод `.outerWidth()`. Мы будем использовать этот метод, чтобы избежать необходимости добавлять ширину отступа и границы. В данном примере мы будем запускать анимационный эффект щелчком мыши на метке Text Size, расположенной над кнопками. Ниже приводится необходимый программный код:

```
$(document).ready(function() {
  $('div.label').click(function() {
    var paraWidth = $('div.speech p').outerWidth();
    var $switcher = $(this).parent();
    var switcherWidth = $switcher.outerWidth();
    $switcher.animate({left: paraWidth - switcherWidth,
            height: '+=20px', borderWidth: '5px'}, 'slow');
  });
});
```

Обратите внимание на оператор `+=`, стоящий перед значением пикселов для свойства `height`. Такое выражение, впервые появившееся в версии jQuery 1.2, указывает, что мы имеем дело с **относительным значением**. Следовательно, высота будет увеличиваться не до 20 пикселов, а на 20 пикселов относительно текущего значения.

Данный программный код благополучно увеличивает высоту элемента `<div>` и расширяет его границы, однако положение левой границы элемента остается неизменным. Нам необходимо разрешить изменение ее положения в таблице стилей CSS.

Позиционирование с помощью CSS

При работе с методом `.animate()` важно помнить об ограничениях, накладываемых таблицами стилей CSS на изменяемые элементы. Например, изменение свойства `left` не будет оказывать желаемого эффекта на набор элементов, если в таблицах стилей CSS значение свойства `position` этих элементов не определено как `relative` или `absolute`. По умолчанию свойство `position` CSS всех блочных элементов имеет значение `static`, что говорит о том, что эти элементы будут оставаться на своих местах, если мы попытаемся переместить их, предварительно не изменив значение свойства `position`.

Примечание

За дополнительной информацией об абсолютном и относительном позиционировании элементов обращайтесь к статье «Absolutely Relative» Джо Джиллеспи (Joe Gillespie), по адресу: *http://www.wpdfd.com/issues/78/absolutely_relative/*

Взгляните на нашу таблицу стилей, где мы определили, что элемент `<div id="switcher">` имеет относительное позиционирование:

```
#switcher {
  position: relative;
}
```

С учетом изменения в таблице стилей CSS после щелчка на метке Text Size и завершения анимационного эффекта страница будет выглядеть так, как показано на рис. 4.8.

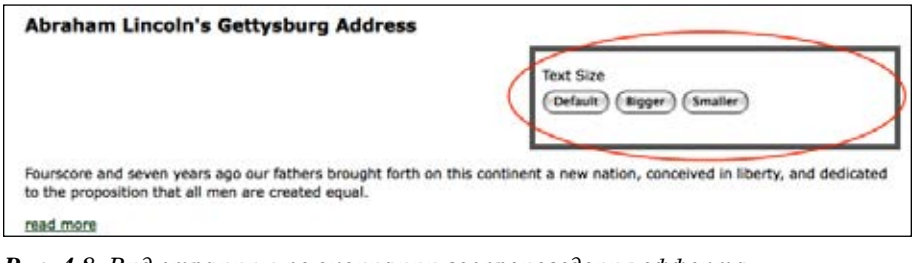


***Рис. 4.8.** Вид страницы по окончании воспроизведения эффекта*

Одновременное и поочередное выполнение эффектов

Как уже говорилось, метод .animate() очень удобно использовать для одновременного воспроизведения эффектов на соответствующем наборе элементов. Однако иногда бывает необходимо воспроизводить эффекты поочередно, то есть один за другим.

Работа с одним набором элементов

При применении множества эффектов к одному и тому же набору элементов **поочередного** их воспроизведения легко добиться за счет составления **цепочки** эффектов. Для демонстрации такого способа поочередного воспроизведения мы снова реализуем перемещение области с меткой Text Size к правому краю страницы, увеличение его высоты и ширины его границы. Только теперь эти три эффекта будут воспроизводиться поочередно, для чего каждый эффект будет оформляться в виде отдельного вызова .animate() и все три будут объединяться в цепочку:

```
$(document).ready(function() {
  $('div.label').click(function() {
    var paraWidth = $('div.speech p').outerWidth();
    var $switcher = $(this).parent();
    var switcherWidth = $switcher.outerWidth();
    $switcher
      .animate({left: paraWidth - switcherWidth},
                                          'slow')
      .animate({height: '+=20px'}, 'slow')
      .animate({borderWidth: '5px'}, 'slow');
  });
});
```

Напомним, что при объединении в цепочку все три вызова метода .animate() можно произвести в одной строке, однако здесь мы поместили их в отдельных строках для большей удобочитаемости.

Допускается объединять в цепочки не только вызовы метода .animate(), но и любые другие методы jQuery воспроизведения эффектов. Например, можно поочередно воспроизвести эффекты для элемента в следующем порядке:

1. Довести непрозрачность до значения .5 с помощью метода .fadeTo().
2. Переместить вправо с помощью метода .animate().
3. Сделать его полностью непрозрачным с помощью метода .fadeTo().
4. Скрыть с помощью метода .slideUp().
5. Отобразить снова с помощью метода .slideDown().

Для этого необходимо лишь объединить в цепочку данные методы в указанном порядке:

```
$(document).ready(function() {
  $('div.label').click(function() {
    var paraWidth = $('div.speech p').outerWidth();
    var $switcher = $(this).parent();
    var switcherWidth = $switcher.outerWidth();
    $switcher
      .fadeTo('fast',0.5)
      .animate({
        'left': paraWidth - switcherWidth
      }, 'slow')
      .fadeTo('slow',1.0)
      .slideUp('slow')
      .slideDown('slow');
  });
});
```

Но как быть, если одновременно с перемещением элемента `<div>` вправо потребуется сделать его полупрозрачным? Если два анимационных эффекта должны протекать с одной и той же скоростью, их можно просто объединить в одном вызове метода `.animate()`. Но в данном примере эффект растворения воспроизводится со скоростью `'fast'`, тогда как эффект перемещения воспроизводится со скоростью `'slow'`. В таком случае нам на выручку приходит вторая форма вызова метода `.animate()`:

```
$(document).ready(function() {
  $('div.label').click(function() {
    var paraWidth = $('div.speech p').outerWidth();
    var $switcher = $(this).parent();
    var switcherWidth = $switcher.outerWidth();
    $switcher
      .fadeTo('fast',0.5)
      .animate({
        'left': paraWidth - switcherWidth
      }, {duration: 'slow', queue: false})
      .fadeTo('slow',1.0)
      .slideUp('slow')
      .slideDown('slow');
  });
});
```

Во втором аргументе, который является отображением параметров, допускается указывать параметр `queue`, который в случае установки в значение `false` предписывает запустить анимационный эффект одновременно с запущенным ранее.

И последнее замечание, касающееся поочередного воспроизведения эффектов для одного набора элементов: очередность выполнения не соблюдается автоматически для других методов, не являющихся методами воспроизведения эффектов, таких как `.css()`. Предположим, что необходимо изменить цвет фона элемента `<div id="switcher">` на красный после воспроизведения эффекта `.slideUp()`, но перед эффектом

.slideDown(). Мы могли бы попытаться реализовать это так, как показано ниже:

```
$(document).ready(function() {
$('div.label').click(function() {
  var paraWidth = $('div.speech p').outerWidth();
  var $switcher = $(this).parent();
  var switcherWidth = $switcher.outerWidth();
  $switcher
    .fadeTo('fast',0.5)
    .animate({
      'left': paraWidth - switcherWidth
    }, 'slow')
    .fadeTo('slow',1.0)
    .slideUp('slow')
    .css('backgroundColor','#f00')
    .slideDown('slow');
  });
});
```

Однако даже хотя программный код, изменяющий цвет фона, помещен в требуемое место в цепочке, цвет изменяется сразу после щелчка мышью.

Один из способов обеспечить необходимую очередность выполнения методов, не являющихся методами воспроизведения эффектов, заключается в использовании метода с соответствующим именем .queue(). Ниже показано, как можно его задействовать в нашем примере:

```
$(document).ready(function() {
  $('div.label').click(function() {
    var paraWidth = $('div.speech p').outerWidth();
    var $switcher = $(this).parent();
    var switcherWidth = $switcher.outerWidth();
    $switcher
      .fadeTo('fast',0.5)
      .animate({
        'left': paraWidth - switcherWidth
      }, 'slow')
      .fadeTo('slow',1.0)
      .slideUp('slow')
      .queue(function() {
        $switcher
          .css('backgroundColor', '#f00')
          .dequeue();
      })
      .slideDown('slow');
  });
});
```

При получении **функции обратного вызова**, как в данном примере, метод .queue() добавляет ее в очередь эффектов соответствующего набора элементов. Внутри функции мы устанавливаем красный цвет фона

и затем добавляем дополнительный вызов метода .dequeue(). Вызов этого метода .dequeue() позволяет очереди анимационных эффектов определить момент, с которого следует ее продолжить, и завершить цепочку последующим вызовом метода .slideDown('slow'). Если не вызвать метод .dequeue(), воспроизведение последовательности анимационных эффектов остановится.

Примечание

Дополнительная информация о методах .queue() и .dequeue() и примеры их использования приводятся по адресу: *http://docs.jquery.com/Effects*.

При исследовании способов поочередного воспроизведения эффектов на множестве наборов элементов мы рассмотрим другие способы объединения в очередь методов, не являющихся методами воспроизведения эффектов.

Работа с несколькими наборами элементов

В отличие от случая с одним набором элементов при применении эффектов к нескольким наборам они могут протекать практически одновременно. Чтобы увидеть одновременное протекание эффектов, мы попробуем развернуть один параграф сверху вниз, а другой свернуть снизу вверх. Для начала добавим в разметку HTML еще один фрагмент Геттисбергского послания, разделив его на два параграфа:

```
<div id="switcher">
  <div class="label">Text Size</div>
  <button id="switcher-default">Default</button>
  <button id="switcher-large">Bigger</button>
  <button id="switcher-small">Smaller</button>
</div>
<div class="speech">
  <p>Fourscore and seven years ago our fathers brought forth
     on this continent a new nation, conceived in liberty, and
     dedicated to the proposition that all men are created
     equal.
  </p>
  <p>Now we are engaged in a great civil war, testing whether
     that nation, or any nation so conceived and so dedicated,
     can long endure. We are met on a great battlefield of
     that war. We have come to dedicate a portion of that
     field as a final resting-place for those who here gave
     their lives that the nation might live. It is altogether
     fitting and proper that we should do this. But, in a
     larger sense, we cannot dedicate, we cannot consecrate,
     we cannot hallow, this ground.
  </p>
  <a href="#" class="more">read more</a>
  <p>The brave men, living and dead, who struggled
```

```
      here have consecrated it, far above our poor
      power to add or detract. The world will little
      note, nor long remember, what we say here, but it
      can never forget what they did here. It is for us
      the living, rather, to be dedicated here to the
      unfinished work which they who fought here have
      thus far so nobly advanced.
    </p>
    <p>It is rather for us to be here dedicated to the
      great task remaining before us–that from
      these honored dead we take increased devotion to
      that cause for which they gave the last full
      measure of devotion–that we here highly
      resolve that these dead shall not have died in
      vain–that this nation, under God, shall
      have a new birth of freedom and that government
      of the people, by the people, for the people,
      shall not perish from the earth.
    </p>
  </div>
```

Затем, чтобы упростить наблюдение за протеканием эффекта, окружим третий параграф рамкой шириной в 1 пиксел, а для четвертого параграфа определим серый цвет фона. Кроме того, скроем четвертый параграф, как только дерево DOM будет готово к работе:

```
$(document).ready(function() {
  $('p:eq(2)').css('border', '1px solid #333');
  $('p:eq(3)').css('backgroundColor', '#ccc').hide();
});
```

В заключение добавим обработчик события `.click()` к третьему параграфу, чтобы после щелчка мышью на третьем параграфе он сворачивался вверх (и исчезал из поля зрения), а четвертый параграф разворачивался вниз (и появлялся в поле зрения):

```
$(document).ready(function() {
$('p:eq(2)')
  .css('border', '1px solid #333')
  .click(function() {
    $(this).slideUp('slow')
    .next().slideDown('slow');
  });
  $('p:eq(3)').css('backgroundColor', '#ccc').hide();
});
```

Снимок с экрана (рис. 4.9), выполненный в середине воспроизведения этих двух эффектов, подтверждает, что они выполняются практически одновременно.

На снимке видно, что третий параграф, который изначально был видимым, свернут наполовину, и в то же время четвертый параграф, который изначально был невидимым, развернут наполовину.

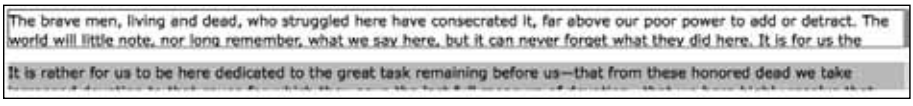


Рис. 4.9. *Два эффекта воспроизводятся одновременно*

Функции обратного вызова

Чтобы обеспечить возможность поочередного воспроизведения эффектов для различных наборов элементов, библиотека jQuery позволяет указывать **функцию обратного вызова** в каждом методе, отвечающем за воспроизведение эффекта. Как мы уже видели при обсуждении обработчиков событий и метода `.queue()`, функции обратного вызова – это обычные функции, которые передаются в виде аргументов методов. В случае анимационных эффектов они передаются в виде последнего аргумента.

Используя функцию обратного вызова для организации поочередного воспроизведения эффектов сворачивания и разворачивания, мы можем сначала развернуть четвертый параграф сверху вниз, а затем свернуть третий снизу вверх. Но прежде рассмотрим, как производится вызов метода `.slideDown()` с функцией обратного вызова:

```
$(document).ready(function() {
  $('p:eq(2)')
    .css('border', '1px solid #333')
    .click(function() {
      $(this).next().slideDown('slow',function() {
        // этот программный код выполняется после того, как закончится
        // разворачивание 3-го параграфа
      });
    });
  $('p:eq(3)').css('backgroundColor', '#ccc').hide();
});
```

Здесь следует быть очень внимательными в отношении определения элемента, который должен сворачиваться. Так как функция обратного вызова находится внутри метода `.slideDown()`, происходит изменение контекста для конструкции `$(this)`. Здесь `$(this)` больше не является третьим параграфом, как это было в случае метода `.click()`; поскольку метод `.slideDown()` связан с вызовом `$(this).next()`, то все, что находится внутри этого метода, будет интерпретировать `$(this)` как следующий братский узел, то есть четвертый параграф. Таким образом, если мы поместим внутрь функции обратного вызова инструкцию `$(this).slideUp('slow')`, в результате свернется тот же параграф, который только что был развернут.

Самый простой способ обеспечить неизменность ссылки `$(this)` заключается в том, чтобы сохранить ее в переменной внутри метода `.click()`, например `var $thirdPara = $(this)`.

Теперь переменная $thirdPara будет ссылаться на третий параграф как внутри, так и за пределами функции обратного вызова. Ниже приводится программный код, использующий новую переменную:

```
$(document).ready(function() {
  var $thirdPara = $('p:eq(2)');
  $thirdPara
    .css('border', '1px solid #333')
    .click(function() {
      $(this).next().slideDown('slow',function() {
        $thirdPara.slideUp('slow');
      });
    });
  $('p:eq(3)').css('backgroundColor', '#ccc').hide();
});
```

Использование переменной $thirdPara внутри функции обратного вызова для метода .slideDown() опирается на одну из особенностей **замыканий**. Эта важная, но сложная для освоения тема будет рассматриваться в приложении С.

На этот раз на снимке экрана (рис. 4.10), выполненном в середине воспроизведения эффектов, можно одновременно наблюдать третий и четвертый параграфы: четвертый параграф уже развернулся полностью, а третий только начал сворачиваться.

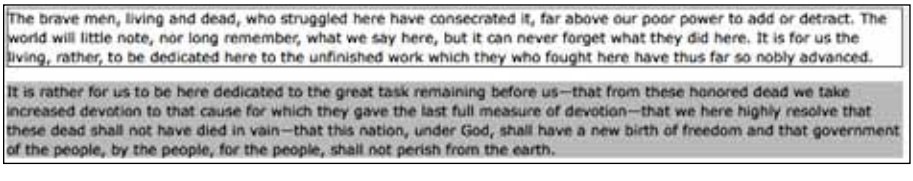


Рис. 4.10. *В середине воспроизведения эффектов можно одновременно наблюдать третий и четвертый параграфы*

Теперь, когда мы обсудили функции обратного вызова, можно вернуться к программному коду, приводившемуся выше в этой главе, где мы добавляли изменение цвета фона ближе к концу последовательности эффектов. Вместо вызова метода .queue(), как это делалось ранее, мы можем просто задействовать функцию обратного вызова:

```
$(document).ready(function() {
  $('div.label').click(function() {
  var paraWidth = $('div.speech p').outerWidth();
  var $switcher = $(this).parent();
  var switcherWidth = $switcher.outerWidth();
    $switcher
    .fadeTo('slow',0.5)
    .animate({
      'left': paraWidth - switcherWidth
```

```
    }, 'slow')
    .fadeTo('slow',1.0)
    .slideUp('slow', function() {
      $switcher
        .css('backgroundColor', '#f00');
    })
    .slideDown('slow');
  });
});
```

Здесь цвет фона элемента `<div id="switcher">` также изменяется на красный после воспроизведения эффекта сворачивания и перед воспроизведением эффекта разворачивания.

В двух словах

С учетом всех возможных вариантов применения эффектов может оказаться сложным запомнить, когда эффекты будут воспроизводиться одновременно, а когда последовательно. В этом случае вам поможет следующая схема:

1. Эффекты, применяемые к одному набору элементов, воспроизводятся:
   - *одновременно*, когда применяются к множеству свойств в одном вызове метода `.animate()`;
   - *последовательно*, когда применяются в виде цепочки методов, при условии, что параметр `queue` установлен в значение `false`.
2. Эффекты, применяемые к нескольким наборам элементов, воспроизводятся:
   - *одновременно* по умолчанию;
   - *последовательно*, когда применяются внутри функций обратного вызова для других эффектов или внутри функции обратного вызова метода `.queue()`.

В заключение

С помощью методов воспроизведения эффектов, рассмотренных в этой главе, мы теперь в состоянии обеспечить пошаговое изменение размера шрифта, которым отображается текст (используя метод `.css()` или `.animate()`)с Кроме того, теперь мы умеем применять различные эффекты постепенного сокрытия или отображения элементов страницы различными способами, а также разными способами воспроизводить несколько анимационных эффектов одновременно или последовательно.

Все примеры в первых четырех главах книги были связаны с управлением элементами, жестко определенными в разметке HTML страницы. В главе 5 мы рассмотрим, как с помощью jQuery создавать новые элементы и вставлять их в любое место дерева DOM по нашему выбору.

5

Манипулирование деревом DOM

Подобно фокуснику, достающему букет цветов из воздуха, библиотека jQuery способна как по волшебству создавать внутри веб-страницы элементы, атрибуты и текст. Но это еще не все! С помощью jQuery мы можем также удалять любые из этих составляющих. А можем взять букет и преобразовать его в голубя: `<div class="magic" id="flowerstodove"> dove</div>`.

Манипулирование атрибутами

На протяжении первых четырех глав для демонстрации способов изменения внешнего вида элементов страницы мы использовали методы `.addClass()` и `.removeClass()`. В действительности эти два метода манипулируют атрибутом `class` (или, говоря на языке DOM, свойством `className`). Метод `.addClass()` создает или добавляет атрибут, а метод `.removeClass()` удаляет или укорачивает значение этого атрибута. Присовокупим к ним метод `.toggleClass()`, который может добавлять или удалять класс, и мы получим эффективный и надежный набор средств для работы с классами CSS.

Однако атрибут `class` – это лишь один из множества атрибутов, значения которых нам может потребоваться получить или изменить (примерами таких атрибутов являются: `id`, `rel` и `href`). Для работы с этими атрибутами библиотека jQuery предоставляет методы `.attr()` и `.removeAttr()`. Методы `.attr()` и `.removeAttr()` могут использоваться и для изменения значения атрибута `class`, но в данном случае предпочтительнее использовать специализированные методы `.addClass()` и `.removeClass()`, потому что они корректно обрабатывают ситуации, когда к одному элементу применяется несколько классов, например `<div class="first second">`.

Атрибуты, отличные от атрибута class

Манипулировать некоторыми атрибутами совсем непросто без помощи библиотеки jQuery. Кроме того, jQuery позволяет одновременно изме-

нять значения нескольких атрибутов, подобно тому как она позволяет одновременно работать с несколькими свойствами CSS при помощи метода `.css()`, о котором рассказывалось в главе 4.

Например, мы легко можем одновременно изменить значения атрибутов `id`, `rel` и `title` у всех ссылок. Для начала приведем фрагмент разметки HTML:

```
<h1 id="f-title">Flatland: A Romance of Many Dimensions</h1>
<div id="f-author">by Edwin A. Abbott</div>
<h2>Part 1, Section 3</h2>
<h3 id="f-subtitle">
    Concerning the Inhabitants of Flatland
</h3>
<div id="excerpt">an excerpt</div>

<div class="chapter">

  <p class="square">Our Professional Men and Gentlemen are
    Squares (to which class I myself belong) and Five-Sided
    Figures or <a
    href="http://en.wikipedia.org/wiki/Pentagon">Pentagons
    </a>.
  </p>
  <p class="nobility hexagon">Next above these come the
    Nobility, of whom there are several degrees, beginning at
    Six-Sided Figures, or <a
    href="http://en.wikipedia.org/wiki/Hexagon">Hexagons</a>,
    and from thence rising in the number of their sides till
    they receive the honourable title of <a
    href="http://en.wikipedia.org/wiki/Polygon">Polygonal</a>,
    or many-Sided. Finally when the number of the sides
    becomes so numerous, and the sides themselves so small,
    that the figure cannot be distinguished from a <a
    href="http://en.wikipedia.org/wiki/Circle">circle</a>, he
    is included in the Circular or Priestly order; and this is
    the highest class of all.
  </p>

  <p><span class="pull-quote">It is a <span class="drop">Law
    of Nature</span> with us that a male child shall have
    <strong>one more side</strong> than his father</span>, so
    that each generation shall rise (as a rule) one step in
    the scale of development and nobility. Thus the son of a
    Square is a Pentagon; the son of a Pentagon, a Hexagon;
    and so on.
  </p>
<!-- ... продолжение кода разметки ... -->

</div>
```

Теперь можно выполнить итерации по всем ссылкам, находящимся внутри элемента , и поочередно изменить значения атрибутов. Если нам необходимо всего лишь установить некоторое общее значение атрибута для всех ссылок, мы можем реализовать это с помощью одной строки программного кода в обработчике $(document).ready():

```
$(document).ready(function() {
  $('div.chapter a').attr({'rel': 'external'});
});
```

Мы можем использовать такой подход, потому что нам требуется установить новое значение атрибута rel для всех ссылок. Однако нередко атрибуты, которые мы добавляем или изменяем, должны иметь различные значения во всех элементах. Например, в любом документе все значения атрибута id должны быть уникальными, если мы хотим, чтобы наш программный код JavaScript вел себя предсказуемо. Чтобы для каждой ссылки установить уникальное значение атрибута id, мы откажемся от однострочного решения в пользу метода .each(), предоставляемого библиотекой jQuery.

```
$(document).ready(function() {
  $('div.chapter a').each(function(index) {
    $(this).attr({
      'rel': 'external',
      'id': 'wikilink-' + index
    });
  });
});
```

Метод .each(), который действует как **явный итератор**, фактически является более удобной формой цикла for. Он может использоваться, когда программный код, который должен выполнять обработку каждого элемента в наборе элементов, соответствующих селектору, оказывается слишком сложным для синтаксической конструкции, выполняющей **неявные итерации**. В нашей ситуации метод .each() передает анонимной функции индекс, который мы можем добавить к каждому значению атрибута id. Аргумент index действует как счетчик, он принимает значения, начиная с 0 и увеличиваясь на 1 для каждой следующей ссылки. В результате выражение 'wikilink-' + index для первой ссылки вернет значение wikilink-0, для второй – wikilink-1, и так далее.

Примечание

В действительности мы могли бы использовать здесь синтаксис неявных итераций, потому что метод .attr() может принимать функцию во втором аргументе подобно методу .filter(), о котором рассказывалось в главе 2 (подробности по адресу: *http://docs.jquery.com/Attributes/attr#keyfn*). Однако использовать метод .each(), на наш взгляд, удобнее.

Мы задействуем атрибут title, чтобы предложить пользователям поближе познакомиться со связанным понятием в Википедии. Все ссылки в представленном фрагменте HTML указывают на сайт Википедии. Однако, вероятно, было бы лучше сделать селектор более конкретным, то есть отбирающим только те ссылки, которые содержат слово wikipedia в значении атрибута href, на тот случай, если позднее нам придется добавить в разметку HTML ссылки, не связанные с Википедией:

```
$(document).ready(function() {
  $('div.chapter a[href*=wikipedia]').each(function(index) {
    var $thisLink = $(this);
    $thisLink.attr({
      'rel': 'external',
      'id': 'wikilink-' + index,
      'title': 'learn more about ' + $thisLink.text() + ' at Wikipedia'
    });
  });
});
```

Обратите внимание, что здесь мы сохраняем значение $(this) в переменной с именем $thisLink, делается это по той простой причине, что данное значение используется более одного раза.

После установки значений всех трех атрибутов разметка HTML, например, первой ссылки будет выглядеть так:

```
<a href="http://en.wikipedia.org/wiki/Pentagon" rel="external"
   id="wikilink--0" title="learn more about Pentagons at
   Wikipedia">Pentagons</a>
```

Еще раз о фабричной функции $()

С самого начала книги мы использовали функцию $(), чтобы получить доступ к элементам документа. В некотором смысле эта функция составляет фундамент библиотеки jQuery, так как она используется всякий раз, когда возникает необходимость подключить обработчик события, воспроизвести эффект или добавить свойство к соответствующему набору элементов.

К тому же функция $() обладает еще одной мощной особенностью – она позволяет изменять не только визуальное представление элементов, но и фактическое содержимое страницы. Просто помещая фрагменты разметки HTML в круглые скобки, мы можем из ничего создавать целые структуры DOM.

Совет

Напоминание о доступности

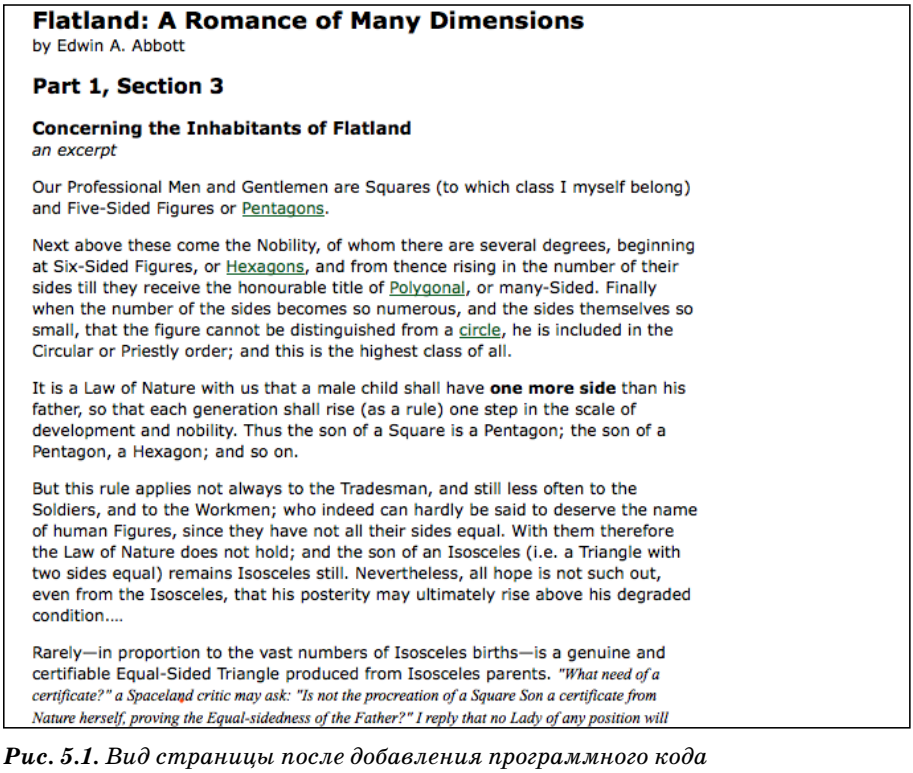
Мы не должны забывать о некоторой рискованности создания определенных функциональных возможностей, визуального оформления или текстовой информации, воспроизводимых, только если веб-броузеры обладают поддерж-

кой JavaScript. Ведь важная информация должна быть доступна для всех, а не только для тех, кто использует подходящее программное обеспечение.

Очень часто на страницах FAQ (сборников ответов на часто задаваемые вопросы) можно встретить ссылку back to top (вернуться к началу) в конце каждой пары вопрос-ответ. Можно смело утверждать, что эти ссылки не несут никакой смысловой нагрузки и поэтому могут добавляться в страницу с помощью JavaScript как дополнительное удобство для подмножества посетителей страницы. В нашем следующем примере мы реализуем добавление ссылки back to top после каждого параграфа, а также добавим якорный элемент, на который будут указывать эти ссылки. Для начала просто создадим новые элементы:

```
$(document).ready(function() {
  $('<a href="#top">back to top</a>');
  $('<a id="top"></a>');
});
```

Теперь наша страница выглядит так, как показано на рис. 5.1.



Flatland: A Romance of Many Dimensions
by Edwin A. Abbott

Part 1, Section 3

Concerning the Inhabitants of Flatland
an excerpt

Our Professional Men and Gentlemen are Squares (to which class I myself belong) and Five-Sided Figures or Pentagons.

Next above these come the Nobility, of whom there are several degrees, beginning at Six-Sided Figures, or Hexagons, and from thence rising in the number of their sides till they receive the honourable title of Polygonal, or many-Sided. Finally when the number of the sides becomes so numerous, and the sides themselves so small, that the figure cannot be distinguished from a circle, he is included in the Circular or Priestly order; and this is the highest class of all.

It is a Law of Nature with us that a male child shall have **one more side** than his father, so that each generation shall rise (as a rule) one step in the scale of development and nobility. Thus the son of a Square is a Pentagon; the son of a Pentagon, a Hexagon; and so on.

But this rule applies not always to the Tradesman, and still less often to the Soldiers, and to the Workmen; who indeed can hardly be said to deserve the name of human Figures, since they have not all their sides equal. With them therefore the Law of Nature does not hold; and the son of an Isosceles (i.e. a Triangle with two sides equal) remains Isosceles still. Nevertheless, all hope is not such out, even from the Isosceles, that his posterity may ultimately rise above his degraded condition....

Rarely—in proportion to the vast numbers of Isosceles births—is a genuine and certifiable Equal-Sided Triangle produced from Isosceles parents. *"What need of a certificate?" a Spaceland critic may ask: "Is not the procreation of a Square Son a certificate from Nature herself, proving the Equal-sidedness of the Father?" I reply that no Lady of any position will*

Рис. 5.1. *Вид страницы после добавления программного кода*

Но куда делись якорь и ссылки back to top? Разве они не должны были появиться на странице? Конечно, нет. Две строки программного кода

выше просто создали элементы, но они еще не были добавлены в страницу. Чтобы это осуществить, необходимо задействовать один из множества **методов добавления** элементов в страницу, входящих в состав библиотеки jQuery.

Добавление новых элементов

В библиотеке jQuery имеется два метода, которые позволяют добавлять элементы перед другими элементами: `.insertBefore()` и `.before()`. Эти два метода реализуют одну и ту же операцию и отличаются только способом **объединения в цепочку** с другими методами. Другие два метода, `.insertAfter()` и `.after()`, точно так же взаимосвязаны между собой, но, как следует из их названий, они вставляют элементы после других элементов. Для добавления ссылок back to top мы будем использовать метод `.insertAfter()`:

```
$(document).ready(function() {
  $('<a href="#top">back to top</a>')
    .insertAfter('div.chapter p');
  $('<a id="top"></a>');
});
```

Метод `.after()` дает тот же эффект, что и метод `.insertAfter()`, но в его случае выражение селектора должно предшествовать методу, а не следовать за ним. При использовании метода `.after()` первая строка внутри `$(document).ready()` могла бы выглядеть так:

```
$('div.chapter p').after('<a href="#top">back to top</a>');
```

При использовании метода `.insertAfter()` мы можем продолжать воздействовать на вновь созданный элемент `<a>`, добавляя в цепочку другие методы. А при использовании метода `.after()` дополнительные методы в цепочке будут воздействовать на элементы, соответствующие селектору `$('div.chapter p')`.

Теперь, когда мы действительно вставили ссылки в страницу (и в дерево DOM) после каждого параграфа, находящегося в пределах элемента `<div class="chapter">`, можно наблюдать появление ссылок back to top на экране, как показано на рис. 5.2.

К сожалению, эти ссылки пока не работают. Нам все еще необходимо добавить в страницу якорь со значением атрибута `id="top"`. Для этого можно воспользоваться одним из методов добавления элементов внутрь других элементов.

```
$(document).ready(function() {
  $('<a href="#top">back to top</a>')
    .insertAfter('div.chapter p');
  $('<a id="top" name="top"></a>')
    .prependTo('body');
});
```

Дополнительная строка программного кода вставляет якорь непосредственно в начало элемента <body>, или, другими словами, в самое начало страницы. Теперь, задействовав метод .insertAfter() для добавления ссылок и метод .prependTo() для добавления якоря, мы получим множество вполне работоспособных ссылок back to top.

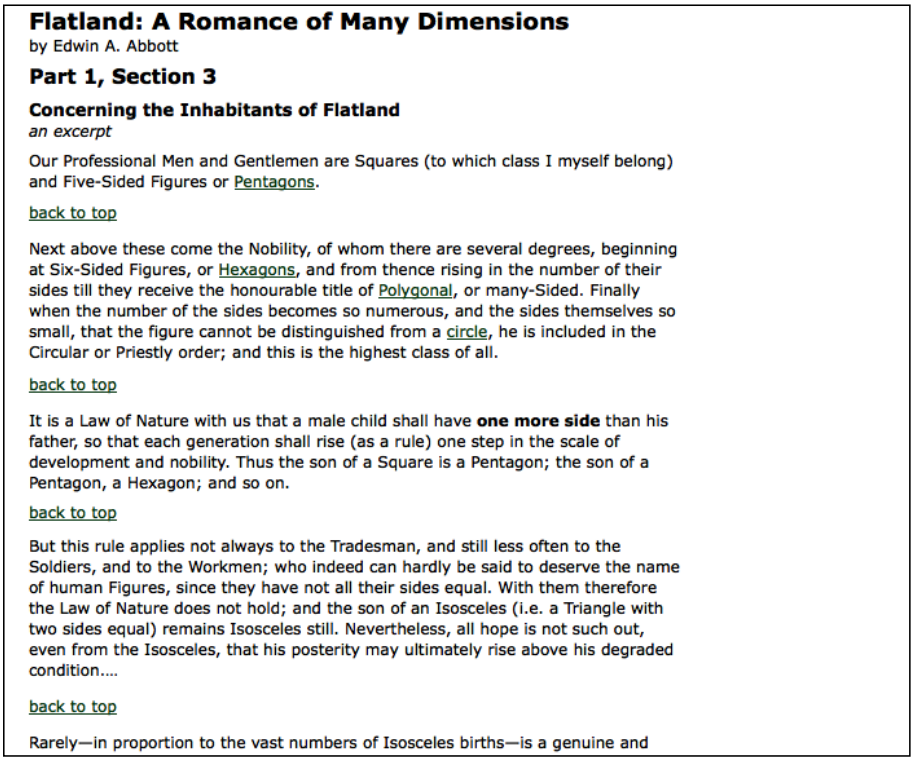


Рис. 5.2. *Вид страницы после добавления на страницу ссылок back to top (вернуться к началу)*

Нет большого смысла добавлять ссылки back to top таким образом, чтобы они появлялись там, откуда начало страницы все еще видимо. С целью усовершенствования сценария мы могли бы начинать добавлять ссылки только, скажем, после четвертого параграфа. Это легко можно реализовать, немного изменив селектор: .insertAfter('div.chapter p:gt(2)'). Но почему здесь используется число 2? Как мы помним, нумерация элементов массива в JavaScript начинается с 0, поэтому первый параграф будет иметь порядковый номер 0, второй – 1, третий – 2, а четвертый – 3. Наш селектор начнет добавлять ссылки после того, как значение индекса достигнет 3, потому что это первый индекс, который больше 2.

Эффект такого изменения селектора теперь очевиден (рис. 5.3).

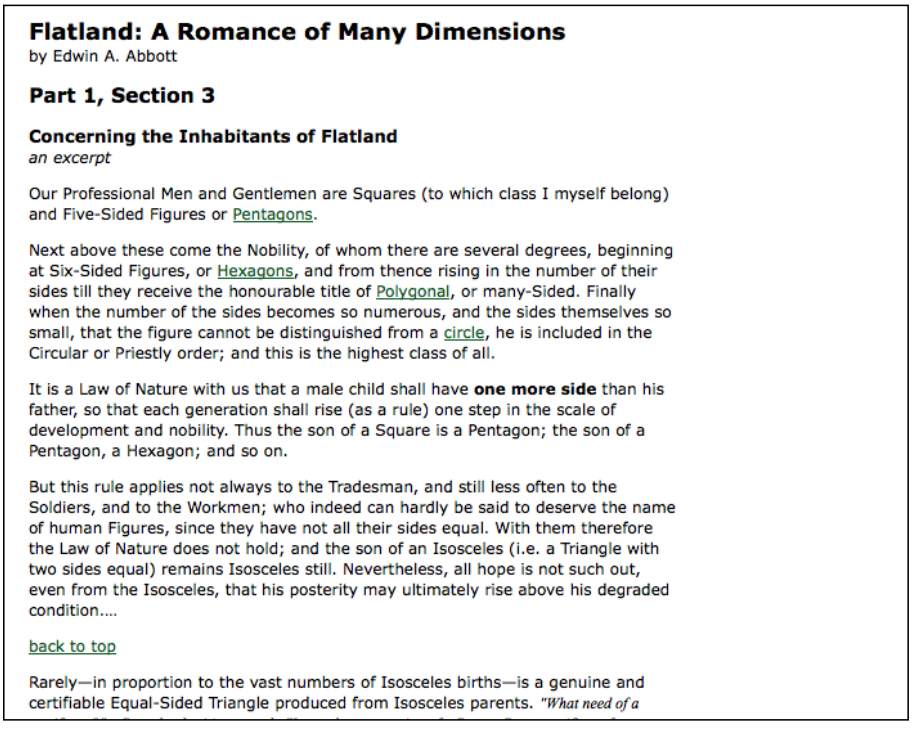


Рис. 5.3. *Теперь ссылки back to top (вернуться к началу) не будут вставляться после первых трех параграфов*

Перемещение элементов

В примере со ссылками back to top мы создали новые элементы и добавили их в страницу. Имеется также возможность извлекать элементы из одного места в странице и вставлять в другое место. Такого рода операция применяется, скажем, в приложении, допускающем динамическое размещение содержимого и оформление сносок. В оригинальном тексте «Flatland», используемом для нашего примера, уже присутствует одна сноска, но в целях демонстрации мы оформили как сноски еще пару фрагментов текста:

```
<p>Rarely&mdash;in proportion to the vast numbers of Isosceles
   births&mdash;is a genuine and certifiable EqualSided
   Triangle produced from Isosceles parents. <span
   class="footnote">"What need of a certificate?" a Spaceland
   critic may ask: "Is not the procreation of a Square Son a
   certificate from Nature herself, proving the Equalsidedness
   of the Father?" I reply that no Lady of any position will
   marry an uncertified Triangle. Square offspring has
   sometimes resulted from a slightly Irregular Triangle;
   but in almost every such case the Irregularity of the
```

```
    first generation is visited on the third; which either
    fails to attain the Pentagonal rank, or relapses to the
    Triangular.</span> Such a birth requires, as its
    antecedents, not only a series of carefully arranged
    intermarriages, but also a long-continued exercise of
    frugality and self-control on the part of the wouldbe
    ancestors of the coming Equilateral, and a patient,
    systematic, and continuous development of the Isosceles
    intellect through many generations.
</p>
<p>The birth of a True Equilateral Triangle from Isosceles
    parents is the subject of rejoicing in our country for many
    furlongs round. After a strict examination conducted by the
    Sanitary and Social Board, the infant, if certified as
    Regular, is with solemn ceremonial admitted into the class
    of Equilaterals. He is then immediately taken from his
    proud yet sorrowing parents and adopted by some childless
    Equilateral. <span class="footnote">The Equilateral is
    bound by oath never to permit the child henceforth to enter
    his former home or so much as to look upon his relations
    again, for fear lest the freshly developed organism may, by
    force of unconscious imitation, fall back again into his
    hereditary level.</span>
</p>
<p>How admirable is the Law of Compensation! <span
    class="footnote">And how perfect a proof of the natural
    fitness and, I may almost say, the divine origin of the
    aristocratic constitution of the States of Flatland!</span>
    By a judicious use of this Law of Nature, the Polygons and
    Circles are almost always able to stifle sedition in its
    very cradle, taking advantage of the irrepressible and
    boundless hopefulness of the human mind.&hellip;
</p>
```

Каждый из этих трех параграфов имеет одну сноску, оформленную как элемент `<span class="footnote"></span>`. Используя такую разметку HTML, мы можем выделять содержимое сносок. Учитывая, что в соответствии с правилами в таблице стилей CSS текст сноски выводится курсивным шрифтом, три параграфа текста будут выглядеть, как показано на рис. 5.4.

Теперь мы можем извлечь сноски и вставить их между элементами `<div class="chapter">` и `<div id="footer">`. Имейте в виду, что даже в случае неявных итераций порядок вставки предопределен, вставка в дерево DOM производится сверху вниз. Поскольку очень важно соблюсти правильный порядок следования сносок в новом месте их расположения, нам следует использовать метод `.insertBefore('#footer')`.

В результате каждая следующая сноска будет вставляться непосредственно перед элементом `<div id="footer">`, то есть сноска 1 будет помещена между элементами `<div class="chapter">` и `<div id="footer">`, сно-

ска 2 будет помещена между сноской 1 и элементом `<div id="footer">`, и так далее. Использование метода `.insertAfter('div.chapter')` привело бы к размещению сносок в обратном порядке. В настоящее время наш программный код выглядит, как показано ниже:

```
$(document).ready(function() {
  $('span.footnote').insertBefore('#footer');
});
```

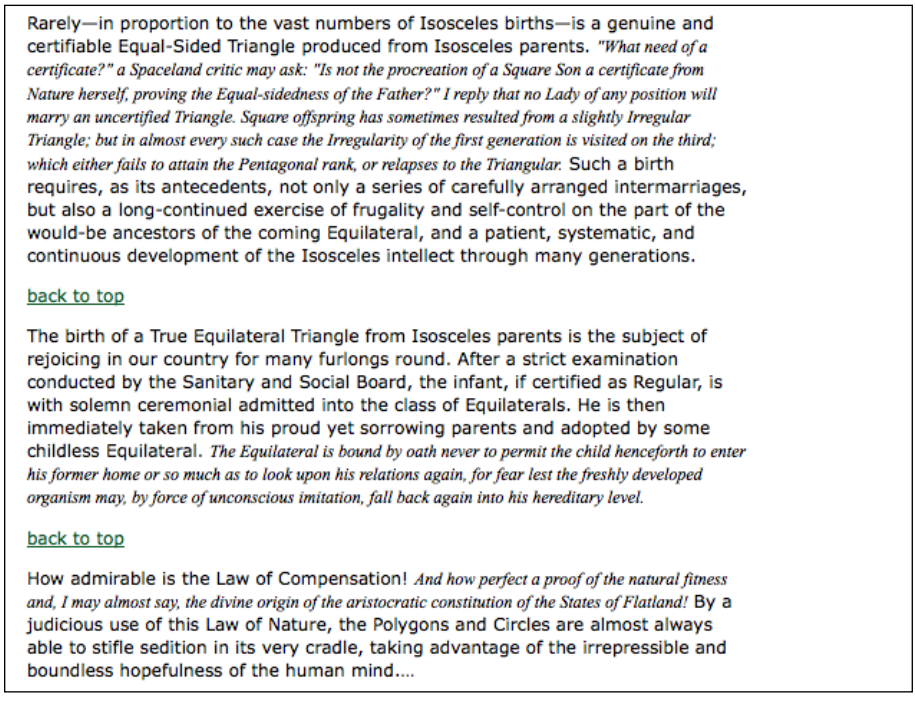


Рис. 5.4. *Текст сносок выводится курсивом*

К сожалению, мы столкнулись с крупной проблемой. Сноски располагаются внутри тегов `<span>`, а это означает, что по умолчанию они отображаются в одной строке друг за другом без разделения, как показано на рис. 5.5.

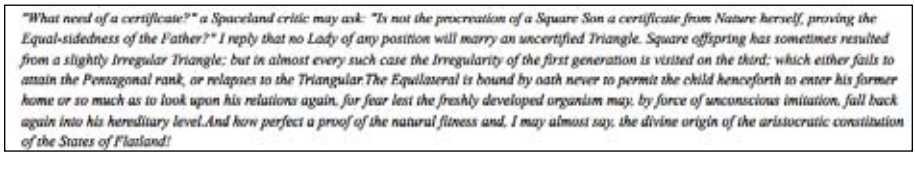
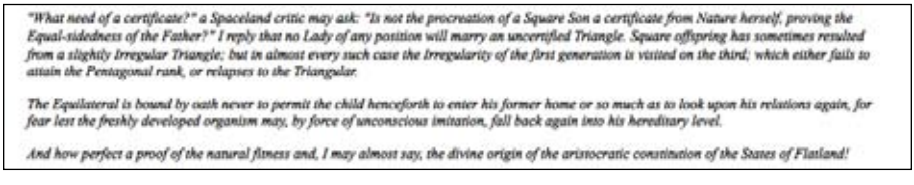


Рис. 5.5. *Сноски отображаются друг за другом без промежутков*

Одно из решений этой проблемы состоит в изменении свойств CSS таким образом, чтобы элементы отображались как блочные элементы, но только те из них, которые находятся внутри элемента :

```
span.footnote {
  font-style: italic;
  font-family: "Times New Roman", Times, serif;
  display: block;
  margin: 1em 0;
}
.chapter span.footnote {
  display: inline;
}
```

Теперь сноски начинают обретать форму, как показано на рис. 5.6.


"What need of a certificate?" a Spaceland critic may ask: "Is not the procreation of a Square Son a certificate from Nature herself, proving the Equal-sidedness of the Father?" I reply that no Lady of any position will marry an uncertified Triangle. Square offspring has sometimes resulted from a slightly Irregular Triangle; but in almost every such case the Irregularity of the first generation is visited on the third, which either fails to attain the Pentagonal rank, or relapses to the Triangular.

The Equilateral is bound by oath never to permit the child henceforth to enter his former home or so much as to look upon his relations again, for fear lest the freshly developed organism may, by force of unconscious imitation, fall back again into his hereditary level.

And how perfect a proof of the natural fitness and, I may almost say, the divine origin of the aristocratic constitution of the States of Flatland!

Рис. 5.6. *Теперь сноски отделены друг от друга*

Сейчас сноски, по крайней мере, отделены друг от друга, и все же с ними еще много чего можно сделать. Более надежное решение должно:

1. Отмечать место в тексте, откуда была изъята каждая сноска.
2. Пронумеровать все местоположения и присвоить каждой сноске соответствующий номер.
3. Создать в тексте ссылку на соответствующую сноску, а в сноске – ссылку на местоположение в тексте.

Все это можно реализовать внутри метода .each(), но сначала подготовим контейнерный элемент внизу страницы, куда будут помещаться сноски:

```
$(document).ready(function() {
  $('<ol id="notes"></ol>').insertAfter('div.chapter');
});
```

Использование упорядоченного списка <ol id="notes"> для сносок выглядит вполне разумным решением – в конце концов, нам необходимо пронумеровать их, так почему бы не использовать элемент, который пронумерует их автоматически? Мы присвоили списку идентификатор notes и вставили его после элемента .

Маркировка, нумерация и создание ссылок на контекст

Теперь мы готовы пометить все места в тексте, откуда извлекаются сноски, и пронумеровать их:

```
$(document).ready(function() {
  $('<ol id="notes"></ol>').insertAfter('div.chapter');
  $('span.footnote').each(function(index) {
    $(this)
      .before(
        ['<a href="#foot-note-',
          index+1,
          '" id="context-',
          index+1,
          '" class="context">',
          '<sup>' + (index+1) + '</sup>',
          '</a>'
        ].join('')
      )
  });
});
```

Здесь мы начинаем с того же селектора, который использовался в упрощенном примере работы со сносками, но к нему мы присоединили вызов метода `.each()`.

Внутри метода `.each()` мы начинаем работу с вызова функции `$(this)`, которая последовательно возвращает каждую сноску, и присоединяем к ней метод `.before()`.

Результатом **объединения элементов массива** в круглых скобках внутри метода `.before()` является ссылка, оформленная в виде верхнего индекса, которая будет вставлена перед каждым элементом `<span>` со ссылкой. Первая ссылка, к примеру, после вставки в дерево DOM будет выглядеть, как показано ниже:

```
<a href="#foot-note-1" id="context-1"
                            class="context"><sup>1</sup></a>
```

На первый взгляд синтаксис может показаться незнакомым, поэтому потратим минуту и разберемся с ним. Внутри круглых скобок метода `.before()` мы сначала помещаем пару квадратных скобок (`[]`), которые представляют **литерал массива**. За каждым элементом массива следует запятая (за исключением последнего элемента, что очень важно). Для большей удобочитаемости мы поместили каждый элемент в отдельной строке. После сборки массива мы преобразуем его в строку с помощью метода `.join()` языка JavaScript. В качестве аргумента этому методу передается пустая строка, представленная парой апострофов, поскольку мы не хотим, чтобы при выводе массива в виде разметки HTML между его элементами присутствовали еще какие-либо символы.

Обратите внимание, что всюду используется выражение `index+1`. Так как счет итераций начинается с 0, мы добавляем 1, чтобы установить

атрибут `href` первой ссылки в значение `#foot-note-1`, атрибут `id` в значение `#context-1` и фактический текст ссылки в значение 1. Атрибут `href` особенно важен, потому что его значение должно в точности совпадать со значением атрибута `id` ссылки (за исключением символа `#`, конечно).

Безусловно, для получения того же результата можно использовать не операцию объединения элементов массива, а операцию конкатенации строк:

```
.before('<a href="#foot-note-' + (index+1) +
                        '" id="context-' + (index+1) +
                        '" class="context"><sup>' +
                        (index+1) + '</sup></a>');
```

Однако в данном случае реализация на основе массива проще в сопровождении, на наш взгляд.

Примечание

В Интернете можно найти массу материалов, сравнивающих производительность операций объединения элементов массива и конкатенации строк. Для самых любопытных в следующей статье приводятся сравнительные результаты ряда испытаний производительности этих двух приемов: *http://www.sitepen.com/blog/2008/05/09/string-performance-an-analysis/*

Однако в большинстве ситуаций имеющиеся различия незаметны. Когда производительность сценария имеет большое значение, есть смысл попытаться применить другие приемы, которые могут оказывать более существенное влияние (например, «кэширование» селекторов, о чем мы уже говорили ранее).

Теперь наши метки, ссылающиеся на сноски, выглядят, как показано на рис. 5.7.

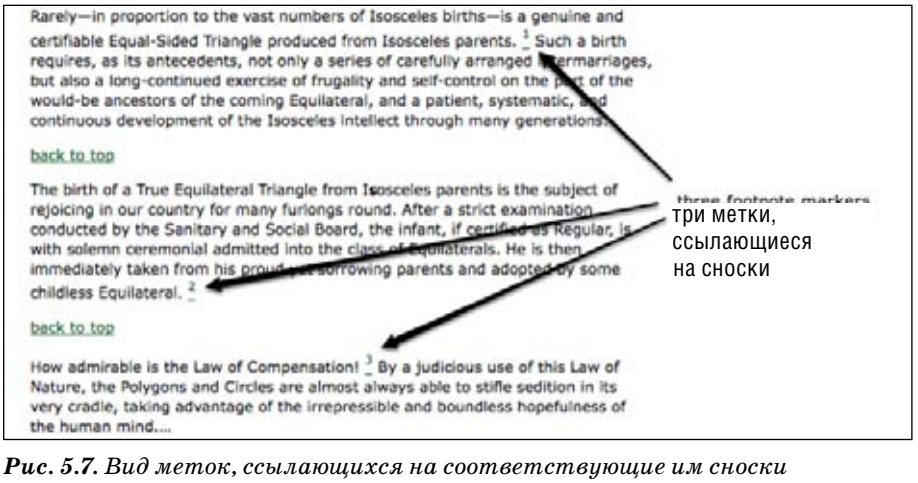


Рис. 5.7. *Вид меток, ссылающихся на соответствующие им сноски*

Добавление сносок

Следующий шаг состоит в перемещении элементов , как это делалось в упрощенном примере. Однако на этот раз элементы будут помещаться во вновь созданный элемент <ol id="notes">. Здесь мы будем использовать метод .appendTo(), что обеспечит надлежащий порядок следования, при котором каждая последующая сноска будет добавляться в конец элемента:

```
$(document).ready(function() {
  $('<ol id="notes"></ol>').insertAfter('div.chapter');
  $('span.footnote').each(function(index) {
    $(this)
      .before(
        ['<a href="#foot-note-',
          index+1,
          '" id="context-',
          index+1,
          '" class="context">',
          '<sup>' + (index+1) + '</sup>',
          '</a>'
        ].join('')
      )
      .appendTo('#notes')
  });
});
```

Важно помнить, что метод .appendTo() по-прежнему объединяется в цепочку с вызовом функции $(this), таким образом мы предписываем библиотеке jQuery: *добавить элемент `span` со сноской в конец элемента со значением атрибута `id`, равным `'notes'`.*

В конец каждой перемещенной сноски мы добавляем ссылку, указывающую на номер в тексте:

```
$(document).ready(function() {
  $('<ol id="notes"></ol>').insertAfter('div.chapter');
  $('span.footnote').each(function(index) {
    $(this)
      .before(
        ['<a href="#foot-note-',
          index+1,
          '" id="context-',
          index+1,
          '" class="context">',
          '<sup>' + (index+1) + '</sup>',
          '</a>'
        ].join('')
      )
      .appendTo('#notes')
      .append( ' (<a href="#context-' + (index+1) + '">context</a>)' );
  });
});
```

Обратите внимание, что значение атрибута `href` ссылается на значение атрибута `id` соответствующей ссылки в тексте. На рис. 5.8 можно видеть сноски с добавленными к ним ссылками.

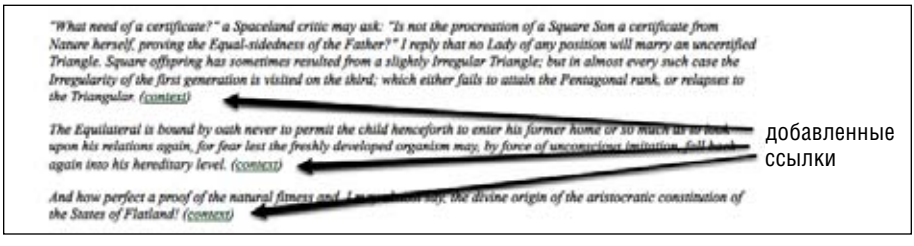


***Рис. 5.8.** Сноски с добавленными ссылками*

Однако в сносках по-прежнему отсутствуют их порядковые номера. Хотя мы и поместили сноски внутрь элемента `<ol>`, каждую из них нужно обернуть в отдельный элемент `<li>`.

Обертывание элементов

Основным методом библиотеки jQuery, используемым для обертывания одних элементов другими, является метод с подходящим названием `.wrap()`. С учетом требования обернуть каждую сноску `$(this)` тегами `<li></li>` окончательная реализация оформления сносок будет выглядеть, как показано ниже:

```
$(document).ready(function() {
  $('<ol id="notes"></ol>').insertAfter('div.chapter');
  $('span.footnote').each(function(index) {
    $(this)
      .before(
        ['<a href="#foot-note-',
          index+1,
          '" id="context-',
          index+1,
          '" class="context">',
          '<sup>' + (index+1) + '</sup>',
          '</a>'
        ].join('')
      )
      .appendTo('#notes')
      .append( ' (<a href="#context-' + (index+1) +'">context</a>)' )
      .wrap('<li id="foot-note-' + (index+1) + '"></li>');
  });
});
```

Теперь атрибуту `id` каждого элемента присваивается значение атрибута `href` соответствующего маркера в тексте. В результате мы получаем на-

бор пронумерованных сносок со ссылками на текст в них, показанный на рис. 5.9.

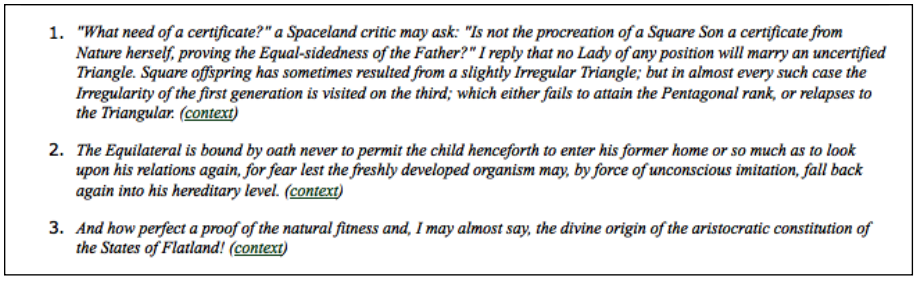

1. *"What need of a certificate?" a Spaceland critic may ask: "Is not the procreation of a Square Son a certificate from Nature herself, proving the Equal-sidedness of the Father?" I reply that no Lady of any position will marry an uncertified Triangle. Square offspring has sometimes resulted from a slightly Irregular Triangle; but in almost every such case the Irregularity of the first generation is visited on the third; which either fails to attain the Pentagonal rank, or relapses to the Triangular. (context)*
2. *The Equilateral is bound by oath never to permit the child henceforth to enter his former home or so much as to look upon his relations again, for fear lest the freshly developed organism may, by force of unconscious imitation, fall back again into his hereditary level. (context)*
3. *And how perfect a proof of the natural fitness and, I may almost say, the divine origin of the aristocratic constitution of the States of Flatland! (context)*

Рис. 5.9. *Набор пронумерованных сносок со ссылками на текст*

Конечно, порядковые номера перед сносками можно было бы вставлять точно так же, как это делалось в параграфах, но наличие осмысленной разметки, динамически создаваемой средствами JavaScript, приносит гораздо более глубокое удовлетворение.

Примечание

В библиотеке jQuery существуют и другие методы, предназначенные для обертывания элементов: `.wrapAll()` и `.wrapInner()`. За дополнительной информацией обращайтесь по адресам: *http://docs.jquery.com/Manipulation/wrapAll* и *http://docs.jquery.com/Manipulation/wrapInner*.

Копирование элементов

До сих пор в этой главе мы вставляли в документ вновь созданные элементы, перемещали элементы из одного места в другое и обертывали существующие элементы новыми. Однако иногда бывает необходимо копировать элементы. Например, меню навигации, присутствующее в заголовке страницы, может потребоваться скопировать в нижнюю часть страницы. В действительности всякий раз, когда предоставляется возможность скопировать элементы для визуального увеличения страницы, очень удобно применять сценарий JavaScript. В конце концов, зачем дважды создавать элементы и удваивать вероятность появления ошибок, когда их можно создать всего один раз и всю тяжелую работу переложить на плечи библиотеки jQuery?

Для копирования элементов библиотека jQuery предоставляет метод `.clone()`; он принимает произвольный набор элементов и создает его копию для последующего использования. Как и при создании новых элементов, рассмотреном выше, скопированные элементы не появятся в странице, пока мы не применим к ним один из методов вставки. Например, следующая строка создает копию первого параграфа, находящегося внутри элемента `<div class="chapter">`:

```
$('div.chapter p:eq(0)').clone();
```

После выполнения этой строки содержимое страницы еще не изменится, как видно из рис. 5.10.

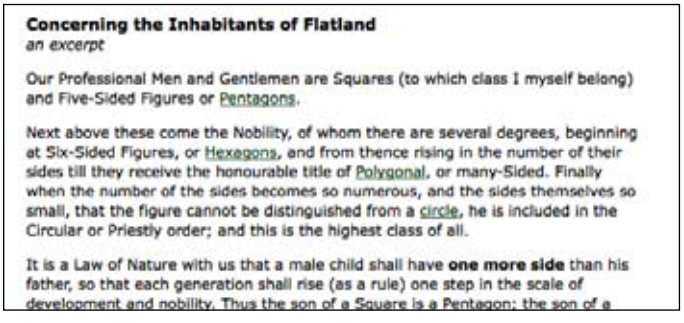


Рис. 5.10. *Скопированные элементы не появляются в странице*

Продолжим пример и заставим скопированный параграф появиться перед элементом `<div class="chapter">`:

```
$('div.chapter p:eq(0)').clone().insertBefore('div.chapter');
```

Теперь первый параграф отображается в странице дважды, а поскольку первый из них находится за пределами элемента `<div class="chapter">`, к нему не применяется оформление, ассоциированное с элементом `div` (особенно это заметно при сравнении их ширин), как показано на рис. 5.11.

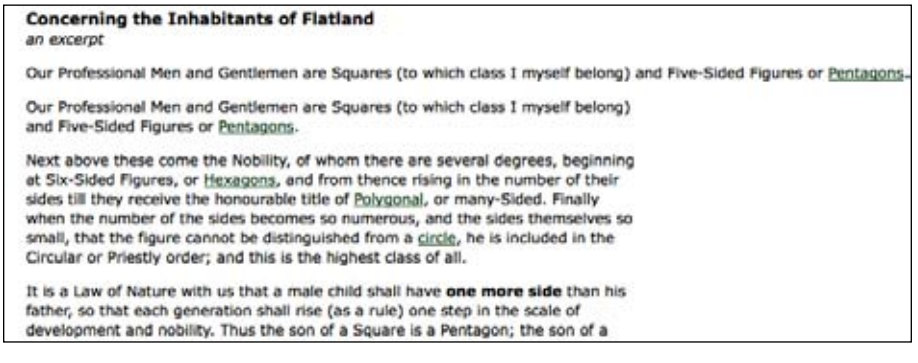


Рис. 5.11. *Скопированный параграф оформлен иначе*

Цепочка из метода `.clone()` и метода добавления аналогична операции копирования и вставки, знакомой большинству людей.

Копирование с обработчиками событий

По умолчанию метод `.clone()` не копирует **обработчики событий**, которые могут быть подключены к соответствующему элементу или его потомкам. Однако он может принимать единственный аргумент со значением логического типа. Если в этом аргументе передается значение `true`, вместе с элементами копируются и обработчики событий, подключенные к ним: `.clone(true)`. Это позволяет избежать необходимости повторно подключать обработчики событий вручную, как описывалось в главе 3.

Копирование с целью создания врезок

На многих веб-сайтах, как и в печатных изданиях, для выделения небольших фрагментов текста и с целью привлечения внимания читателя используются **врезки**. Мы легко можем реализовать их с помощью метода `.clone()`. Для начала посмотрим на изменения в третьем параграфе текста из нашего примера:

```
<p>
  <span class="pull-quote">It is a Law of Nature
  <span class="drop">with us</span> that a male child shall
  have <strong>one more side</strong> than his father</span>,
  so that each generation shall rise (as a rule) one step in
  the scale of development and nobility. Thus the son of a
  Square is a Pentagon; the son of a Pentagon, a Hexagon; and
  so on.
</p>
```

Обратите внимание, что параграф начинается с элемента `<span class="pull-quote">`. Элементы с этим классом мы и будем копировать. После копирования текста внутри элемента `<span>` он будет вставляться в другое место, поэтому нам необходимо изменить его оформление для отделения от остальной части текста.

Стили CSS

Для оформления внешнего вида врезок нам необходимо добавить класс `pulled` к скопированному элементу `<span>` и присвоить этому классу в таблице стилей следующие свойства:

```
.pulled {
  background: #e5e5e5;
  position: absolute;
  width: 145px;
  top: -20px;
  right: -180px;
  padding: 12px 5px 12px 10px;
  font: italic 1.4em "Times New Roman", Times, serif;
}
```

Теперь текст врезки будет отображаться другим шрифтом на светло-сером фоне с дополнительными отступами. Но самое важное, что врезка будет иметь абсолютное позиционирование и располагаться на 20 пикселов выше и на 20 пикселов правее ближайшего предка в дереве DOM (имеющего абсолютное или относительное позиционирование). Если для предка не определен режим абсолютного или относительного позиционирования (отличного от `static`), врезка будет позиционироваться относительно элемента `<body>` документа. По этой причине в программном коде, использующем методы библиотеки jQuery, нам необходимо присвоить родительскому элементу, включающему копируемый текст врезки, свойство стиля `position:relative`.

Позиционирование по верхней границе кажется достаточно очевидным, но может быть не совсем понятно, как блок врезки позиционируется на 20 пикселов правее своего предка. Сначала мы получаем полную ширину блока врезки, которая складывается из значения свойства `width` и величины отступов слева и справа, то есть: 145 px + 5 px + 10 px, или 160 px. Затем мы выбираем значение свойства `right` класса врезки. При значении 0 в этом свойстве правый край врезки выравнивался бы по правому краю родительского элемента. Следовательно, чтобы поместить левый край врезки на 20 пикселов правее родительского элемента, необходимо сместить ее правый край в отрицательном направлении на расстояние, на 20 пикселов превышающее ее полную ширину, то есть на –180 px.

Программный код

Теперь можно подключить к работе библиотеку jQuery. Начнем с селектора, который отберет все элементы `<span class="pull-quote">`, и присоединим к нему метод `.each()`, чтобы иметь возможность выполнять несколько операций в процессе обхода элементов:

```
$(document).ready(function() {
  $('span.pull-quote').each(function(index) {
    //...
  });
});
```

Затем для каждой врезки найдем родительский параграф и присвоим ему новое значение свойства CSS `position`:

```
$(document).ready(function() {
  $('span.pull-quote').each(function(index) {
    var $parentParagraph = $(this).parent('p');
    $parentParagraph.css('position', 'relative');
  });
});
```

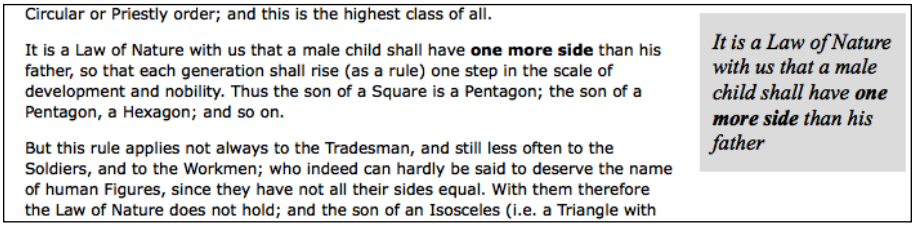
Здесь, чтобы повысить производительность и удобочитаемость, мы снова сохраняем полученный набор элементов в переменной.

Теперь, когда установлены значения необходимых свойств CSS, все готово к созданию врезки. Сейчас можно создать копию каждого элемента , добавить к ней класс pulled и вставить ее в начало параграфа:

```
$(document).ready(function() {
  $('span.pull-quote').each(function(index) {
    var $parentParagraph = $(this).parent('p');
    $parentParagraph.css('position', 'relative');
    $(this).clone()
      .addClass('pulled')
      .prependTo($parentParagraph);
  });
});
```

Так как врезка имеет абсолютное позиционирование, тот факт, что она находится внутри параграфа, не имеет большого значения. Пока врезка находится внутри параграфа, она будет позиционироваться относительно верхней и правой границы этого параграфа в соответствии с правилами CSS. Однако, если бы мы применили к свойству position врезки значение float, ее нахождение внутри параграфа сказалось бы на его расположении по вертикали.

Теперь параграф вместе с врезкой выглядит так, как показано на рис. 5.12.


Circular or Priestly order; and this is the highest class of all.

It is a Law of Nature with us that a male child shall have **one more side** than his father, so that each generation shall rise (as a rule) one step in the scale of development and nobility. Thus the son of a Square is a Pentagon; the son of a Pentagon, a Hexagon; and so on.

But this rule applies not always to the Tradesman, and still less often to the Soldiers, and to the Workmen; who indeed can hardly be said to deserve the name of human Figures, since they have not all their sides equal. With them therefore the Law of Nature does not hold; and the son of an Isosceles (i.e. a Triangle with

*It is a Law of Nature with us that a male child shall have **one more side** than his father*

Рис. 5.12. *Вид параграфа с врезкой*

Для начала неплохо, но во врезках, как правило, не сохраняется форматирование текста, как это получилось с фрагментом one more side. Нам необходимо из текста, включенного в элемент , удалить , , <a href> и другие линейные теги. Кроме того, было бы неплохо иметь возможность немного изменять текст врезки: отбрасывать некоторые слова и замещать их многоточием. Для этого в текстовом примере мы обернули несколько слов тегом : with us.

Сначала мы заменим выделенные слова многоточием, а потом заместим разметку HTML врезки простым текстом:

```
$(document).ready(function() {
  $('span.pull-quote').each(function(index) {
```

```
    var $parentParagraph = $(this).parent('p');
    $parentParagraph.css('position', 'relative');
    var $clonedCopy = $(this).clone();
    $clonedCopy
      .addClass('pulled')
      .find('span.drop')
        .html('&hellip;')
      .end()
      .prependTo($parentParagraph);
    var clonedText = $clonedCopy.text();
    $clonedCopy.html(clonedText);
  });
});
```

На этот раз процесс копирования начинается с сохранения копии в переменной. Переменная необходима просто из-за отсутствия возможности выполнить все необходимые операции в одной цепочке. Обратите также внимание, что после того, как мы отыскали элемент `<span class="drop">` и заменили его разметку HTML многоточием (`&hellip;`), мы используем метод `.end()`, чтобы отклонить результаты последнего запроса, `.find('span.drop')`. Благодаря этому мы получаем возможность вставить в начало параграфа всю копию целиком, а не только многоточие.

В конце мы создаем еще одну переменную, `clonedText`, куда записываем простое текстовое содержимое копии, и этим текстовым содержимым заменяем разметку HTML копии. Теперь врезка выглядит, как показано на рис. 5.13.

Как видите, в один из последующих параграфов был добавлен еще один элемент `<span class="pull-quote">`, чтобы убедиться, что наш программный код может обрабатывать несколько таких элементов на странице.

Украшение врезок

Теперь врезки работают, как и следовало ожидать, дочерние элементы разметки удаляются, а вместо отбрасываемого текста добавляются многоточия.

Тем не менее одна из наших целей состоит в том, чтобы придать врезкам привлекательный внешний вид, поэтому мы могли бы добавить к ним закругленные углы и тени. Однако переменная высота врезок порождает проблему, так как из-за нее возникает необходимость добавлять два фоновых изображения к одному элементу, что невозможно ни в одном современном броузере, за исключением последних сборок броузера Safari.

Для преодоления этого ограничения мы можем завернуть каждую врезку в другой элемент `<div>`:

```
$(document).ready(function() {
  $('span.pull-quote').each(function(index) {
```

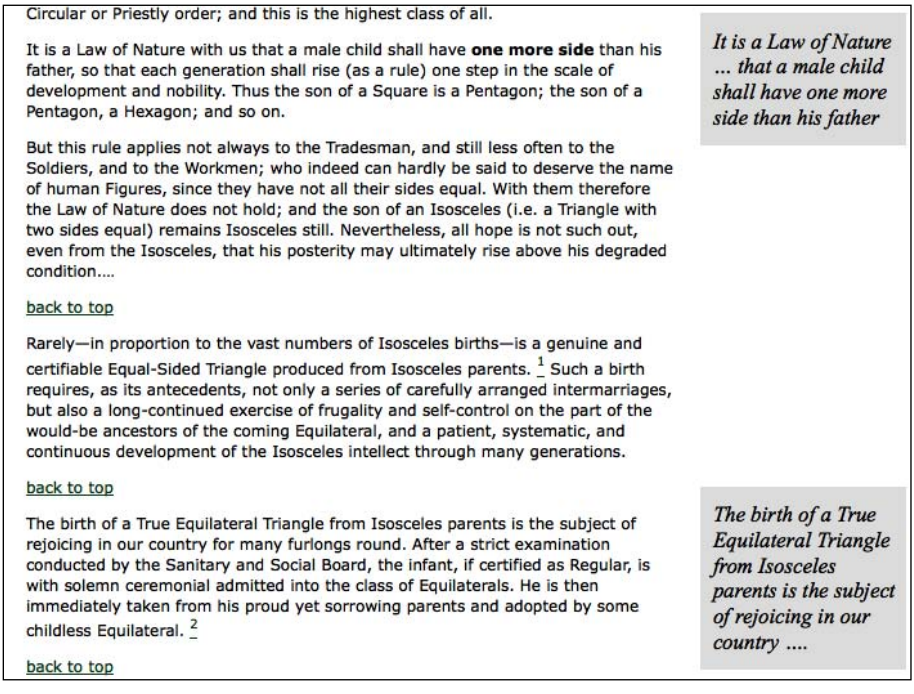


Рис. 5.13. *Вид врезки, в которой часть текста заменена многоточием*

```
    var $parentParagraph = $(this).parent('p');
    $parentParagraph.css('position', 'relative');
    var $clonedCopy = $(this).clone();
    $clonedCopy
      .addClass('pulled')
      .find('span.drop')
        .html('&hellip;')
      .end()
      .prependTo($parentParagraph)
      .wrap('<div class="pulled-wrapper"></div>');
    var clonedText = $clonedCopy.text();
    $clonedCopy.html(clonedText);
  });
});
```

Кроме того, мы должны изменить таблицы стилей CSS, чтобы обеспечить оформление нового элемента `<div>` и добавить два фоновых изображения:

```
.pulled-wrapper {
  background: url(pq-top.jpg) no-repeat left top;
  position: absolute;
```

```
    width: 160px;
    right: -180px;
    padding-top: 18px;
  }
  .pulled {
    background: url(pq-bottom.jpg) no-repeat left bottom;
    position: relative;
    display: block;
    width: 140px;
    padding: 0 10px 24px 10px;
    font: italic 1.4em "Times New Roman", Times, serif;
  }
```

Здесь некоторые правила, которые ранее применялись к элементам `<span class="pulled">`, теперь применяются к элементам `<div class="pulled-wrapper">`. Изменения в свойствах `width` и `padding` учитывают использование фоновых изображений для оформления границ, а в классе `.pulled` изменены значения свойств `position` и `display`, чтобы обеспечить корректное отображение во всех броузерах.

В итоге приукрашенные врезки выглядят так, как показано на рис. 5.14.

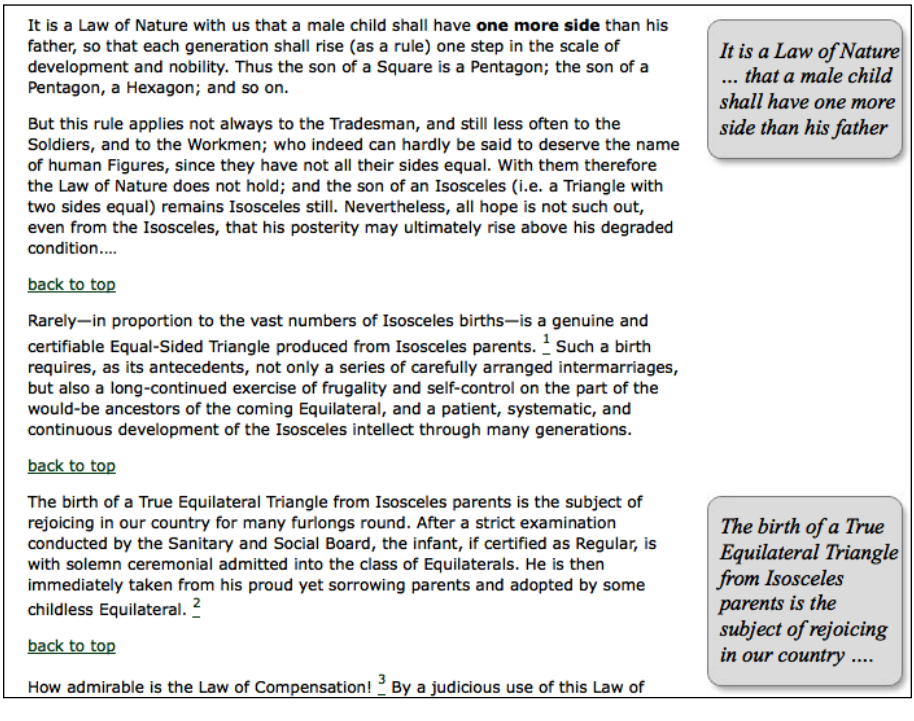


Рис. 5.14. *Окончательный вид врезок*

Коротко о методах манипулирования деревом DOM

Многочисленные методы манипулирования деревом DOM, присутствующие в jQuery, позволяют решать самые разнообразные задачи. Следующее краткое описание может служить напоминанием о том, какие методы имеются в наличии и какие задачи с их помощью можно решать.

1. Чтобы **создать** новые элементы из разметки HTML, следует использовать фабричную функцию `$()`.
2. Чтобы **вставить** новые элементы **внутрь** каждого соответствующего элемента, следует использовать:
   - `.append()`
   - `.appendTo()`
   - `.prepend()`
   - `.prependTo()`
3. Чтобы **вставить** новые элементы **рядом** с каждым соответствующим элементом, следует использовать:
   - `.after()`
   - `.insertAfter()`
   - `.before()`
   - `.insertBefore()`
4. Чтобы **обернуть** каждый соответствующий элемент новым элементом, следует использовать:
   - `.wrap()`
   - `.wrapAll()`
   - `.wrapInner()`
5. Чтобы **заменить** каждый соответствующий элемент новым элементом или текстом, следует использовать:
   - `.html()`
   - `.text()`
   - `.replaceAll()`
   - `.replaceWith()`
6. Чтобы **удалить** элементы, находящиеся **внутри** соответствующих элементов, следует использовать:
   - `.empty()`
7. Чтобы **убрать** из документа все соответствующие элементы и их потомков без их фактического удаления, следует использовать:
   - `.remove()`

В заключение

В этой главе мы научились создавать, копировать, пересобирать и приукрашивать элементы содержимого с помощью методов манипулирования деревом DOM из библиотеки jQuery. Мы применили эти методы к одной веб-странице, придав нескольким параграфам стилизованный литературный вид со сносками, ссылками и врезками.

Учебный раздел книги почти закончен, но прежде чем перейти к исследованию более сложных примеров, мы совершим путешествие к серверу с помощью методов поддержки технологии AJAX, входящих в состав библиотеки jQuery.

6

AJAX

В последние годы стало общепринятым предосудительное отношение к сайтам, которые основаны на применении нестандартных технологий. Одним из самых известных словечек, используемых для описания новых веб-приложений, стало **AJAX-Powered** (управляется технологиями AJAX). Этот термин употребляется для обозначения самых разных вещей, так как охватывает целую группу взаимосвязанных технологий.

Формально слово **AJAX** является аббревиатурой от **Asynchronous JavaScript and XML** (асинхронный JavaScript и XML). В число технологий, включенных в решение AJAX, входят:

- *JavaScript* для организации взаимодействий с пользователем и обработки событий, происходящих в броузерах
- Объект **XMLHttpRequest**, который позволяет выполнять запросы к серверу без прерывания выполнения других задач в броузере
- Файлы в формате *XML* или в других подобных форматах, таких как HTML или JSON, на стороне сервера
- И снова *JavaScript* для интерпретации данных, полученных от сервера, и их отображения на странице

Технологии AJAX были провозглашены средством спасения Всемирной паутины, позволяющим превратить статические веб-страницы в интерактивные веб-приложения. Было создано множество платформ, призванных помочь разработчикам, использующим эти технологии, решить проблему несовместимости реализаций объекта `XMLHttpRequest` в разных броузерах. И библиотека jQuery не исключение.

В этой главе мы попытаемся разобраться, действительно ли AJAX позволяет творить чудеса.

Загрузка данных по требованию

Если отбросить всю эту рекламную шумиху, AJAX – это всего лишь средство загрузки данных с **сервера** в веб-броузер, или **клиент**, не вызывающее видимого обновления страницы. Эти данные могут принимать самые разные формы, и при получении их они могут обрабатываться массой разных способов. Мы убедимся в этом, решая одну и ту же простую задачу разными путями.

Мы создадим страницу, которая будет отображать словарные статьи, сгруппированные по первому символу статьи. Ниже приводится разметка HTML, определяющая область содержимого страницы:

```
<div id="dictionary">
</div>
```

Это не ошибка! Изначально страница не содержит никакой информации. Мы будем использовать различные методы поддержки технологии AJAX, входящие в состав библиотеки jQuery, чтобы заполнить этот элемент `<div>` словарными статьями.

Нам необходим механизм, запускающий процесс загрузки данных, поэтому мы добавим несколько ссылок, к которым будем подключать **обработчики событий**:

```
<div class="letters">
  <div class="letter" id="letter-a">
    <h3><a href="#">A</a></h3>
  </div>
  <div class="letter" id="letter-b">
    <h3><a href="#">B</a></h3>
  </div>
  <div class="letter" id="letter-c">
    <h3><a href="#">C</a></h3>
  </div>
  <div class="letter" id="letter-d">
    <h3><a href="#">D</a></h3>
  </div>
</div>
```

Примечание

Как обычно, действующая реализация должна предусматривать возможность **прогрессивного улучшения**, чтобы обеспечить работоспособность страницы в броузерах с отключенной поддержкой JavaScript. Здесь, чтобы упростить пример, мы не предусматриваем никаких действий для ссылок, пока с помощью jQuery к ним не будут подключены обработчики событий.

Добавив несколько правил CSS, мы получили страницу, которая выглядит, как показано на рис. 6.1.

***Рис. 6.1.** Начальная страница после добавления некоторых правил CSS*

Теперь можно сосредоточиться на наполнении страницы содержимым.

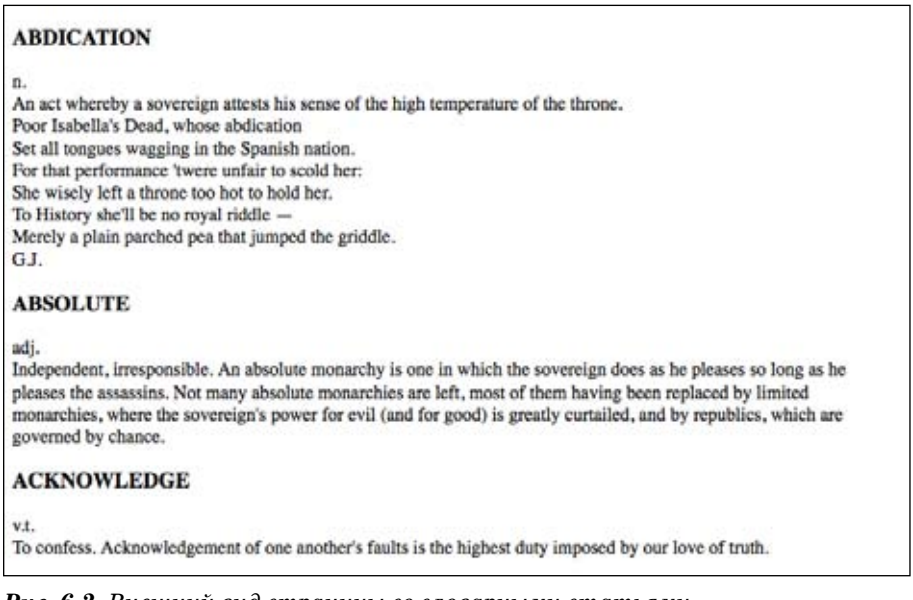
Добавление разметки HTML

Приложения с поддержкой технологии AJAX зачастую просто запрашивают получение фрагментов разметки HTML. Этот прием, который иногда называется **AHAH (Asynchronous HTTP and HTML – асинхронный HTTP и HTML)**, чрезвычайно просто реализуется с помощью jQuery. Во-первых, нам необходима некоторая разметка HTML для добавления в страницу, которую мы поместим в файл с именем `a.html`, находящийся в одном каталоге с нашим документом. Ниже приводится начало этого файла HTML:

```
<div class="entry">
  <h3 class="term">ABDICATION</h3>
  <div class="part">n.</div>
  <div class="definition">
    An act whereby a sovereign attests his sense of the high
                              temperature of the throne.
    <div class="quote">
      <div class="quote-line">Poor Isabella's Dead, whose
                              abdication</div>
      <div class="quote-line">Set all tongues wagging in the
                              Spanish nation.</div>
      <div class="quote-line">For that performance 'twere
                              unfair to scold her:</div>
      <div class="quote-line">She wisely left a throne too
                              hot to hold her.</div>
      <div class="quote-line">To History she'll be no royal
                              riddle &mdash;</div>
      <div class="quote-line">Merely a plain parched pea that
                              jumped the griddle.</div>
      <div class="quote-author">G.J.</div>
    </div>
  </div>
</div>
```

```
<div class="entry">
  <h3 class="term">ABSOLUTE</h3>
  <div class="part">adj.</div>
  <div class="definition">
    Independent, irresponsible. An absolute monarchy is one
    in which the sovereign does as he pleases so long as he
    pleases the assassins. Not many absolute monarchies are
    left, most of them having been replaced by limited
    monarchies, where the sovereign's power for evil (and for
    good) is greatly curtailed, and by republics, which are
    governed by chance.
  </div>
</div>
```

Страница содержит множество статей с такой же структурой разметки HTML. Будучи самостоятельной страницей, она выглядит очень просто, как показано на рис. 6.2.



ABDICATION

n.
An act whereby a sovereign attests his sense of the high temperature of the throne.
Poor Isabella's Dead, whose abdication
Set all tongues wagging in the Spanish nation.
For that performance 'twere unfair to scold her:
She wisely left a throne too hot to hold her.
To History she'll be no royal riddle —
Merely a plain parched pea that jumped the griddle.
G.J.

ABSOLUTE

adj.
Independent, irresponsible. An absolute monarchy is one in which the sovereign does as he pleases so long as he pleases the assassins. Not many absolute monarchies are left, most of them having been replaced by limited monarchies, where the sovereign's power for evil (and for good) is greatly curtailed, and by republics, which are governed by chance.

ACKNOWLEDGE

v.t.
To confess. Acknowledgement of one another's faults is the highest duty imposed by our love of truth.

Рис. 6.2. *Внешний вид страницы со словарными статьями*

Обратите внимание, что файл `a.html` не является настоящим документом HTML – в нем отсутствуют теги `<html>`, `<head>` и `<body>`, которые являются обязательными. Обычно мы называем такие файлы **отрывками** или **фрагментами**, их основное назначение – хранить разметку HTML, которая будет вставляться в другой документ HTML, которым мы сейчас и займемся:

```
$(document).ready(function() {
  $('#letter-a a').click(function() {
    $('#dictionary').load('a.html');
    return false;
  });
});
```

Метод `.load()` выполняет всю тяжелую работу! С помощью обычного селектора jQuery мы указали место, куда будет вставляться отрывок, и затем передали методу адрес URL загружаемого файла в виде аргумента. Теперь, когда на ссылке будет выполнен первый щелчок, файл будет загружен, а его содержимое вставлено в элемент `<div id="dictionary">`. Броузер отобразит новую разметку HTML после ее добавления в документ, как показано на рис. 6.3.

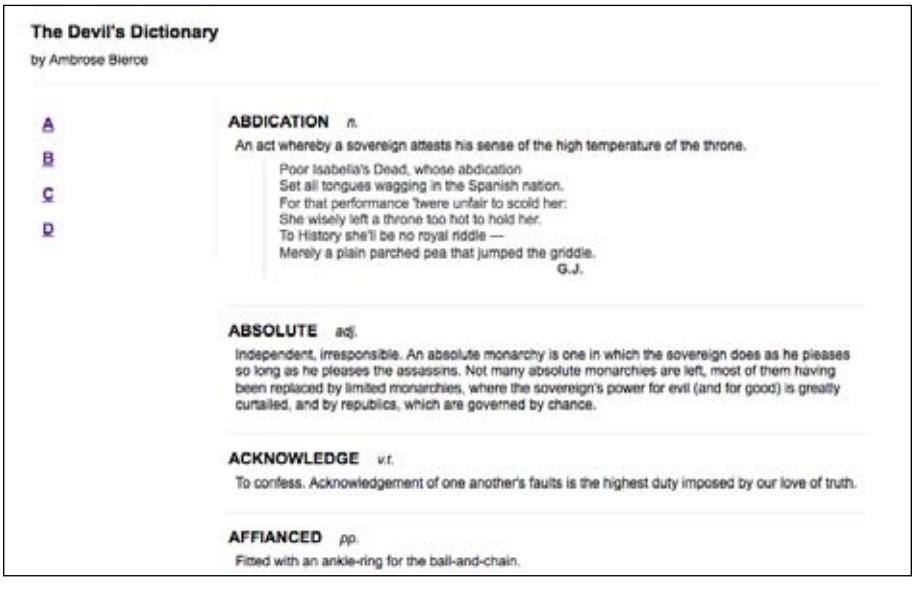


Рис. 6.3. *Вид страницы после добавления нового фрагмента разметки HTML*

Обратите внимание, что прежде простая разметка HTML приобрела оформление. Это обусловлено правилами CSS в основном документе – когда в документ вставляется новый фрагмент HTML, к его тегам применяются правила оформления, существующие в основном документе.

При опробовании этого примера вы наверняка увидите, что словарные определения будут появляться практически мгновенно, сразу же после щелчка на ссылке. В этом заключается опасность работы с локальными приложениями – в такой ситуации очень сложно учесть задержки, возникающие при передаче документов через сеть. Предположим,

что мы добавили диалог, сообщающий о завершении загрузки определений:

```
$(document).ready(function() {
  $('#letter-a a').click(function() {
    $('#dictionary').load('a.html');
    alert('Loaded!');
    return false;
  });
});
```

Из структуры этого программного кода можно было бы предположить, что диалог будет отображаться только после того, как загрузка данных будет завершена. Программный код JavaScript обычно выполняется **синхронно**, действия происходят в строгой последовательности друг за другом.

Однако если опробовать этот программный код на действующем вебсервере, из-за задержек, возникающих в сети, диалог наверняка появится прежде, чем загрузка будет завершена. Это происходит потому, что все запросы AJAX по умолчанию выполняются **асинхронно**. В противном случае нам пришлось бы назвать эту технологию SJAX, что едва ли вызвало бы точно такую же шумиху! В асинхронном режиме загрузки, сразу после того как будет отправлен запрос HTTP на получение фрагмента разметки HTML, сценарий немедленно продолжит свою работу. Спустя некоторое время броузер примет ответ от сервера и обработает его. В большинстве случаев именно такой порядок работы является желаемым – было бы недружелюбным по отношению к пользователю блокировать весь броузер, ожидая получения данных.

Для случаев, когда действие должно быть отложено до завершения загрузки данных, библиотека jQuery предоставляет возможность определить **функцию обратного вызова**, реализующую эти действия. Пример такой функции будет представлен ниже.

Работа с объектами JavaScript

Запрашивать уже готовый фрагмент разметки HTML очень удобно, но иногда бывает необходимо, чтобы сценарий выполнял предварительную обработку данных, прежде чем отобразить их. В этом случае нам необходимо организовать получение данных в виде структуры, обход которой можно будет выполнить с помощью JavaScript.

Используя селекторы jQuery, мы могли бы реализовать обход получаемой разметки HTML и выполнить требуемые операции, но для этого она сначала должна быть вставлена в документ. Для обработки данных в формате, который ближе к языку JavaScript, может даже потребоваться меньше программного кода.

Извлечение объектов JavaScript

Зачастую нам встречаются **объекты JavaScript**, которые являются всего лишь множеством **пар ключ-значение** и могут быть определены с помощью фигурных скобок ({}). С другой стороны, массивы JavaScript могут определяться в ходе выполнения с помощью квадратных скобок ([]). Комбинируя эти две концепции, мы легко можем выразить самые сложные структуры данных.

Для обозначения этого простого синтаксиса Дугласом Крокфордом (Douglas Crockford) был придуман термин **JSON (JavaScript Object Notation – форма записи объектов JavaScript)**. Эта форма записи может рассматриваться как более компактная альтернатива более расточительному формату XML:

```
{
  "key": "value",
  "key 2": [
    "array",
    "of",
    "items"
  ]
}
```

Примечание

Дополнительную информацию о потенциальных преимуществах формата JSON и его реализациях во многих языках программирования можно найти по адресу: *http://json.org/*.

Представить свои данные в этом формате можно самыми разными способами. Некоторые словарные статьи в формате JSON мы поместим в файл с именем `b.json`, начало которого приводится ниже:

```
[
  {
    "term": "BACCHUS",
    "part": "n.",
    "definition": "A convenient deity invented by the...",
    "quote": [
      "Is public worship, then, a sin,",
      "That for devotions paid to Bacchus",
      "The lictors dare to run us in,",
      "And resolutely thump and whack us?"
    ],
    "author": "Jorace"
  },
  {
    "term": "BACKBITE",
    "part": "v.t.",
    "definition": "To speak of a man as you find him when..."
  },
```

```
{
  "term": "BEARD",
  "part": "n.",
  "definition": "The hair that is commonly cut off by..."
},
```

Для извлечения этих данных мы будем использовать метод `$.getJSON()`, который извлекает файл и обрабатывает его, возвращая вызывающему программному коду сформированный объект JavaScript.

Глобальные функции jQuery

До этого момента мы использовали только методы объекта jQuery, создаваемого фабричной функцией `$()`. Селекторы позволяли нам определять множество требуемых узлов дерева DOM, а методы давали возможность воздействовать на них тем или иным способом. Однако функция `$.getJSON()` – это совсем другое дело. Здесь нет элемента DOM, к которому логично было применить эту функцию – возвращаемый объект должен передаваться сценарию, а не добавляться в страницу. По этой причине функция `getJSON()` определена как метод **глобального объекта jQuery** (единственный объект с именем `jQuery`, или `$`, определяемый библиотекой jQuery), а не как метод конкретного **экземпляра объекта jQuery** (объекты, которые создаются с помощью функции `$()`).

Если бы в JavaScript имелись классы, похожие на классы в других объектно-ориентированных языках программирования, мы бы назвали `$.getJSON()` **методом класса**. В дальнейшем обсуждении методы такого типа мы будем назвать **глобальными функциями**. В действительности – это функции, использующие **пространство имен** jQuery, чтобы избежать конфликтов с именами других функций.

При использовании этой функции ей передается имя файла, так же как и прежде:

```
$(document).ready(function() {
  $('#letter-b a').click(function() {
    $.getJSON('b.json');
    return false;
  });
});
```

Этот программный код не производит никаких видимых эффектов при щелчке мышью на ссылке. Функция загружает файл, но мы еще не сказали JavaScript, что делать с полученными данными. Для этого нам потребуется задействовать **функцию обратного вызова**.

Функция `$.getJSON()` принимает во втором аргументе функцию, которая будет вызвана по завершении загрузки. Как уже упоминалось выше, вызовы AJAX выполняются **асинхронно**, а функция обратного вызова является способом отложить выполнение операций до момента, пока не будут получены данные. Функция обратного вызова также принима-

ет аргумент, в котором ей передаются полученные данные. Поэтому мы можем написать такой программный код:

```
$(document).ready(function() {
  $('#letter-b a').click(function() {
    $.getJSON('b.json', function(data) {
    });
    return false;
  });
});
```

Здесь в качестве функции обратного вызова мы использовали **анонимную функцию**, что часто используется для сокращения объема программного кода при работе с библиотекой jQuery. С тем же успехом в качестве функции обратного вызова можно было бы использовать именованную функцию.

Внутри этой функции мы можем использовать переменную `data` для обхода структуры с данными. Нам потребуется выполнить итерации по элементам массива и для каждого из них создать разметку HTML. Это можно было бы реализовать с помощью стандартного цикла `for`, но вместо этого мы познакомимся с другой глобальной функцией jQuery – `$.each()`. В главе 5 мы уже встречались с родственным ей методом `.each()`. Но в отличие от него, эта функция работает не с объектом jQuery, а принимает массив или отображение в первом аргументе и функцию обратного вызова – во втором. На каждой итерации производится обращение к функции обратного вызова, которой в виде двух аргументов передаются текущий **порядковый номер итерации** и текущий элемент массива или отображения.

```
$(document).ready(function() {
  $('#letter-b a').click(function() {
    $.getJSON('b.json', function(data) {
      $('#dictionary').empty();
      $.each(data, function(entryIndex, entry) {
        var html = '<div class="entry">';
        html += '<h3 class="term">' + entry['term'] + '</h3>';
        html += '<div class="part">' + entry['part'] + '</div>';
        html += '<div class="definition">';
        html += entry['definition'];
        html += '</div>';
        html += '</div>';
        $('#dictionary').append(html);
      });
    });
    return false;
  });
});
```

Перед началом цикла выполняется очистка содержимого элемента `<div id="dictionary">`, чтобы мы могли наполнить его вновь созданной размет-

кой HTML. Затем с помощью функции `$.each()` поочередно извлекается каждый элемент массива и на основе его содержимого создается структура HTML. В заключение полученная разметка HTML вставляется в дерево DOM посредством добавления ее в элемент `<div>`.

Примечание

Такой подход предполагает, что полученные данные можно без опаски вставлять в разметку HTML, то есть, они не содержат специальных символов, таких как `<`.

Все, что осталось сделать, – это предусмотреть обработку статей с цитатами, которая выполняется с помощью другого цикла `$.each()`:

```
$(document).ready(function() {
  $('#letter-b a').click(function() {
    $.getJSON('b.json', function(data) {
      $('#dictionary').empty();
      $.each(data, function(entryIndex, entry) {
        var html = '<div class="entry">';
        html += '<h3 class="term">' + entry['term'] + '</h3>';
        html += '<div class="part">' + entry['part'] + '</div>';
        html += '<div class="definition">';
        html += entry['definition'];
        if (entry['quote']) {
          html += '<div class="quote">';
          $.each(entry['quote'], function(lineIndex, line) {
            html += '<div class="quote-line">' + line + '</div>';
          });
          if (entry['author']) {
            html += '<div class="quote-author">' + entry['author'] + '</div>';
          }
          html += '</div>';
        }
        html += '</div>';
        html += '</div>';
        $('#dictionary').append(html);
      });
    });
    return false;
  });
});
```

Добавив этот программный код, можно попробовать щелкнуть на ссылке B и увидеть результаты, как показано на рис. 6.4.

Примечание

Формат JSON обеспечивает компактность представления данных, но он не прощает ошибок. Каждая скобка, кавычка и запятая должны находиться на своем месте, иначе файл не будет загружен. В большинстве броузеров вы даже не получите сообщение об ошибке – сценарий просто будет терпеть неудачу.

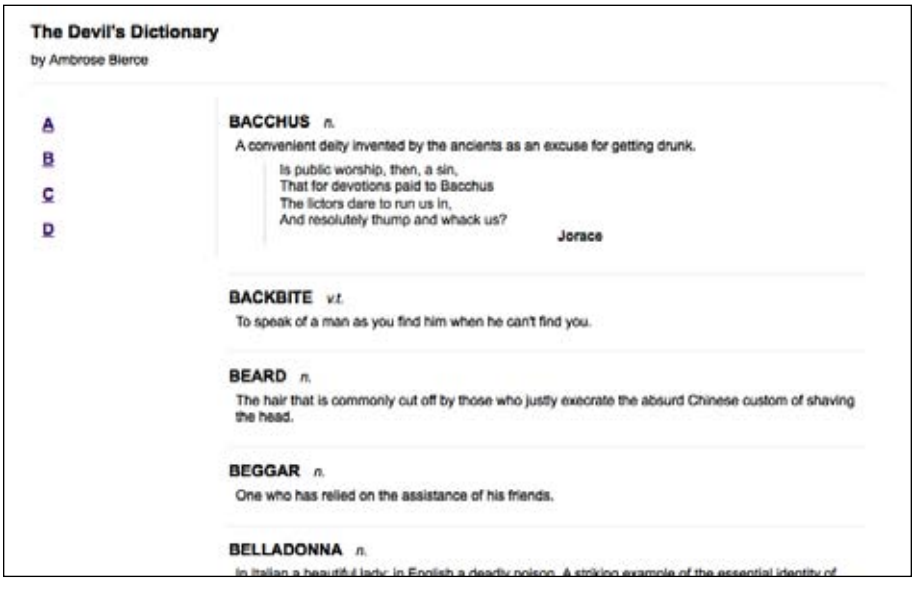


Рис. 6.4. *Вид страницы после загрузки данных в формате JSON*

Запуск сценария

Иногда требуется, чтобы не весь программный код JavaScript загружался вместе со страницей. Мы можем не знать, какие сценарии потребуются, пока пользователь не выполнит какие-либо действия. В такой ситуации можно «на лету» генерировать теги `<script>`, по мере необходимости, однако существует более элегантный способ внедрения дополнительного программного кода, который заключается в том, чтобы с помощью jQuery напрямую загружать файл `.js`.

Загрузка сценария выполняется так же просто, как и загрузка фрагмента HTML. Для этого можно воспользоваться глобальной функцией `$.getScript()`, которая, как и родственные ей функции, принимает адрес URL файла сценария:

```
$(document).ready(function() {
  $('#letter-c a').click(function() {
    $.getScript('c.js');
    return false;
  });
});
```

В нашем последнем примере нам требовалось обработать полученные данные, чтобы можно было с пользой использовать загруженный файл. В случае же с файлом сценария его обработка выполняется автоматически – сценарий просто запускается.

Сценарии, полученные таким способом, выполняются в **глобальном контексте** текущей страницы. Это означает, что они имеют доступ ко всем глобальным функциям и переменным, включая и библиотеку jQuery. Благодаря этому мы можем сымитировать работу примера, использующего данные в формате JSON, для подготовки и добавления в страницу разметки HTML во время выполнения сценария и поместить этот программный код в отдельный файл `c.js`:

```
var entries = [
{
  "term": "CALAMITY",
  "part": "n.",
  "definition": "A more than commonly plain and..."
},
{
  "term": "CANNIBAL",
  "part": "n.",
  "definition": "A gastronome of the old school who..."
},
{
  "term": "CHILDHOOD",
  "part": "n.",
  "definition": "The period of human life intermediate..."
},
{
  "term": "CLARIONET",
  "part": "n.",
  "definition": "An instrument of torture operated by..."
},
{
  "term": "COMFORT",
  "part": "n.",
  "definition": "A state of mind produced by..."
},
{
  "term": "CORSAIR",
  "part": "n.",
  "definition": "A politician of the seas."
}
];

var html = '';

$.each(entries, function() {
  html += '<div class="entry">';
  html += '<h3 class="term">' + this['term'] + '</h3>';
  html += '<div class="part">' + this['part'] + '</div>';
  html += '<div class="definition">' + this['definition'] + '</div>';
  html += '</div>';
});
$('#dictionary').html(html);
```

Теперь щелчок на ссылке C будет давать желаемый результат, как показано на рис. 6.5.

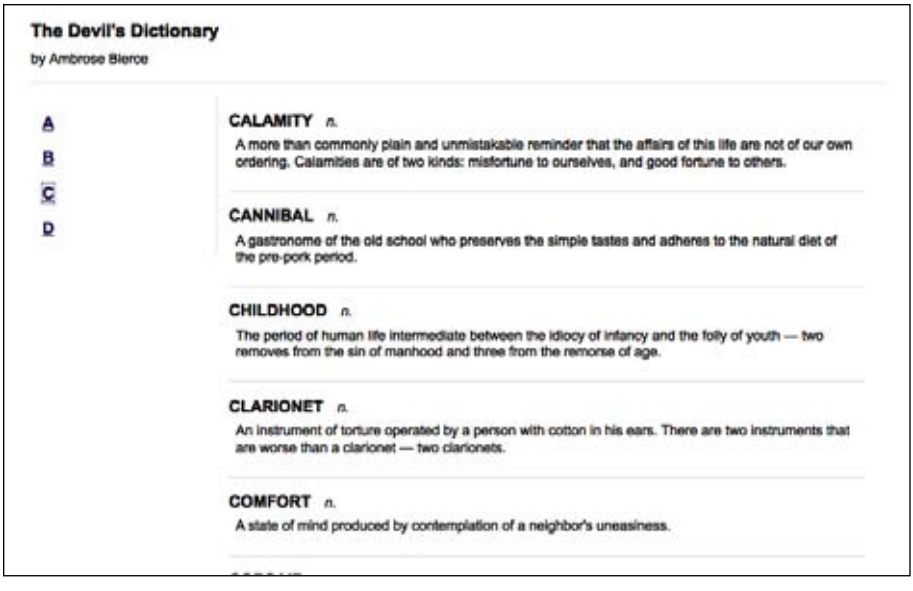


***Рис. 6.5.** Вид страницы после загрузки данных с дополнительным сценарием*

Загрузка документа XML

Название XML является частью аббревиатуры AJAX, но мы еще не рассматривали загрузку данных в формате XML. Эта возможность реализуется просто и очень похожа на методику работы с форматом JSON. Для начала нам потребуется XML-файл `d.xml`, содержащий некоторые данные, которые требуется отобразить:

```
<?xml version="1.0" encoding="UTF-8"?>
<entries>
  <entry term="DEFAME" part="v.t.">
    <definition>
      To lie about another. To tell the truth about another.
    </definition>
  </entry>
  <entry term="DEFENCELESS" part="adj.">
    <definition>
      Unable to attack.
    </definition>
  </entry>
  <entry term="DELUSION" part="n.">
    <definition>
      The father of a most respectable family, comprising
      Enthusiasm, Affection, Self-denial, Faith, Hope,
```

```
      Charity and many other goodly sons and daughters.
    </definition>
    <quote author="Mumfrey Mappel">
      <line>All hail, Delusion! Were it not for thee</line>
      <line>The world turned topsy-turvy we should see;</line>
      <line>For Vice, respectable with cleanly fancies,</line>
      <line>Would fly abandoned Virtue's gross advances.</line>
    </quote>
  </entry>
  <entry term="DIE" part="n.">
    <definition>
      The singular of "dice." We seldom hear the word,
      because there is a prohibitory proverb, "Never say
      die." At long intervals, however, some one says: "The
      die is cast," which is not true, for it is cut. The
      word is found in an immortal couplet by that eminent
      poet and domestic economist, Senator Depew:
    </definition>
    <quote>
      <line>A cube of cheese no larger than a die</line>
      <line>May bait the trap to catch a nibbling mie.</line>
    </quote>
  </entry>
</entries>
```

Конечно, эти данные можно выразить множеством способов, и некоторые из них более близко имитируют структуру, выбранную для форматов HTML и JSON, использовавшихся выше. Однако здесь мы постарались продемонстрировать некоторые особенности XML, которые обеспечивают более высокую удобочитаемость этого формата для человека, например использование **атрибутов** для определения терминов и разделов вместо отдельных **тегов**.

Начало функции уже знакомо нам:

```
$(document).ready(function() {
  $('#letter-d a').click(function() {
    $.get('d.xml', function(data) {

    });
    return false;
  });
});
```

На этот раз основную работу будет выполнять функция `$.get()`. Вообще говоря, эта функция просто извлекает файл из заданного адреса URL и передает функции обратного вызова простой текст. Однако если из ответа известно, что он имеет формат XML, благодаря тому, что сервер вместе с ответом возвращает его **тип MIME**, функция обратного вызова получит дерево DOM XML.

К счастью, как мы уже видели, библиотека jQuery обладает мощными средствами обхода дерева DOM. При обработке документов XML мы можем использовать обычные методы `.find()`, `.filter()` и другие методы обхода дерева, как если бы это был документ HTML:

```
$(document).ready(function() {
  $('#letter-d a').click(function() {
    $.get('d.xml', function(data) {
      $('#dictionary').empty();
      $(data).find('entry').each(function() {
        var $entry = $(this);
        var html = '<div class="entry">';
        html += '<h3 class="term">' + $entry.attr('term') + '</h3>';
        html += '<div class="part">' + $entry.attr('part') + '</div>';
        html += '<div class="definition">';
        html += $entry.find('definition').text();
        var $quote = $entry.find('quote');
        if ($quote.length) {
          html += '<div class="quote">';
          $quote.find('line').each(function() {
            html += '<div class="quote-line">' + $(this).text() + '</div>';
          });
          if ($quote.attr('author')) {
            html += '<div class="quote-author">'
              + $quote.attr('author') + '</div>';
          }
          html += '</div>';
        }
        html += '</div>';
        html += '</div>';
        $('#dictionary').append($(html));
      });
    });
    return false;
  });
});
```

Этот программный код воспроизводит требуемый эффект, когда выполняется щелчок на ссылке D, как показано на рис. 6.6.

Этот новый способ использования уже знакомых нам методов обхода дерева DOM подчеркивает гибкость поддержки селекторов CSS в библиотеке jQuery. Синтаксис стилей CSS обычно используется для улучшения оформления страниц HTML, поэтому для определения селекторов в стандартных файлах `.css` используются имена тегов HTML, такие как `div` или `body`, чтобы указать местоположение искомого содержимого. Однако при работе с библиотекой jQuery наравне со стандартными именами тегов HTML допускается использовать произвольные имена тегов XML, такие как `entry` и `definition`.

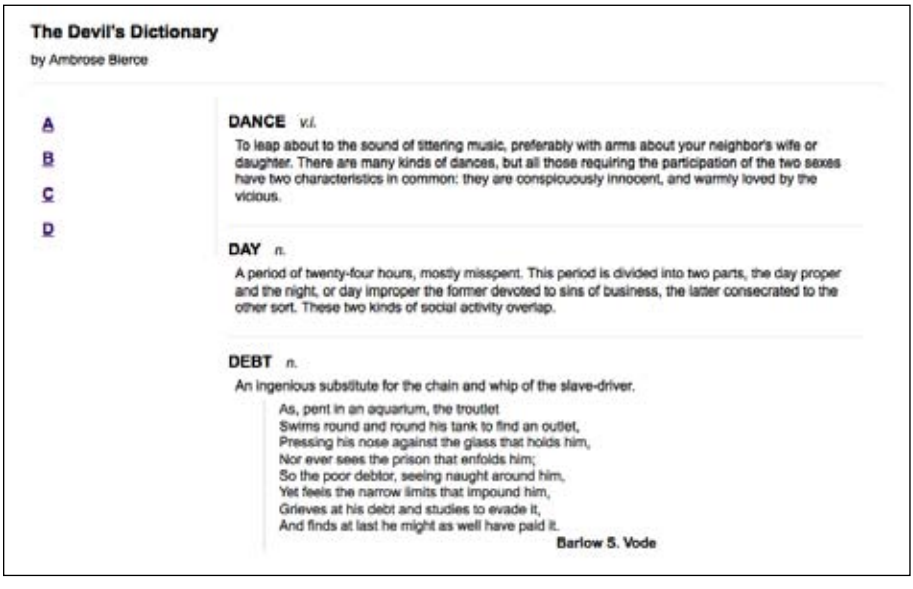


Рис. 6.6. *Вид страницы после загрузки данных в формате XML*

Улучшенный механизм селекторов, реализованный в jQuery, упрощает поиск элементов документа XML даже в самых сложных ситуациях. Например, предположим, что нам требуется отобразить только те статьи, в которых имеются цитаты, для которых в свою очередь указаны авторы. Для этого мы можем организовать выборку только тех статей, которые имеют вложенные элементы `<quote>`, заменив селектор `entry` на `entry:has(quote)`. Затем можно добавить дополнительное ограничение и отобрать только те статьи, в которых элемент `<quote>` имеет атрибут `author`, указав селектор `entry:has(quote[author])`. Теперь строка с начальным селектором будет выглядеть так:

```
$(data).find('entry:has(quote[author])').each(function() {
```

Этот новый селектор соответствующим образом ограничивает перечень возвращаемых словарных статей, как показано на рис. 6.7.

Выбор формата данных

Мы рассмотрели четыре формата для внешних данных, каждый из которых обрабатывается функциями поддержки технологии AJAX из библиотеки jQuery. Мы также убедились, что все четыре формата позволяют решить поставленную задачу загрузки информации в существующую страницу по требованию пользователя, но не раньше. Но как тогда решить, какой из них использовать в своих приложениях?

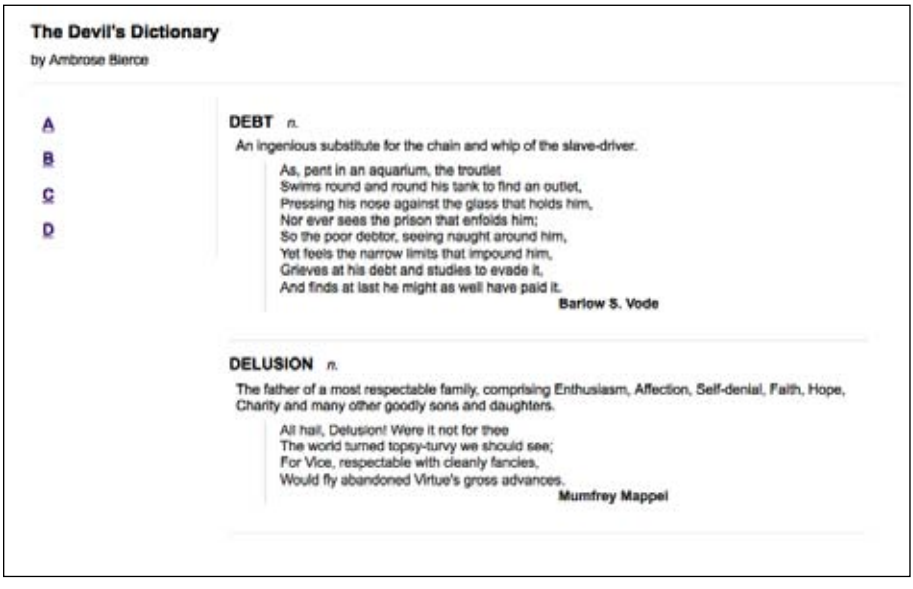


Рис. 6.7. *Вид страницы с ограниченным набором словарных статей*

Фрагменты HTML требуют очень небольшого объема программного кода для его обработки. Внешние данные могут загружаться и вставляться в страницу одним простым методом, который даже не требует функции обратного вызова. Для решения простой задачи добавления новой разметки HTML в существующую страницу нет необходимости обращаться к методам обхода элементов дерева DOM. С другой стороны, данные необязательно могут быть структурированы так, чтобы их можно было повторно использовать в других приложениях. Внешний файл слишком тесно связан с контейнером, куда он вставляется.

Файлы JSON структурированы и легко могут использоваться повторно. Они компактны, и легко читаются. Чтобы извлечь информацию из такого файла и отобразить ее на странице, необходимо выполнить обход структуры данных, но это можно реализовать с помощью стандартных средств JavaScript. Разбор файлов может производиться единственным вызовом функции `eval()` языка JavaScript, поэтому чтение файла в формате JSON выполняется чрезвычайно быстро. Однако любое обращение к функции `eval()` представляет собой угрозу безопасности. Ошибки в файле JSON могут вызывать завершение работы сценария без вывода сообщений об ошибках или побочные эффекты, поэтому данные должны подготавливаться с особой тщательностью.

Файлы JavaScript обеспечивают непревзойденную гибкость, но в действительности они не являются механизмом хранения данных. Эти файлы тесно связаны с языком программирования, поэтому они не могут использоваться для передачи той же самой информации системам, отличным от JavaScript. Однако использование файлов JavaScript

позволяет вынести редко используемые операции во внешние файлы, уменьшить объем программного кода и загружать требуемую реализацию только по мере необходимости.

Документы XML обеспечивают наилучшую переносимость. Так как XML стал *общепринятым языком* обмена данными с веб-службами, предоставление данных в этом формате обеспечивает возможность повторного их использования в разных местах. Примерами могут служить такие сайты, как Flickr (*http://flickr.com/*), del.icio.us (*http://del.icio.us/*) и Upcoming (*http://upcoming.org/*), которые экспортируют данные в формате XML, что позволяет использовать их во многих **гибридных приложениях**. Однако формат XML является довольно расточительным, а скорость работы с ним может быть ниже, чем с другими форматами.

Учитывая все эти характеристики, обычно проще организовать передачу данных в виде фрагментов HTML при условии, что они не будут использоваться другими приложениями. В случаях, когда данные могут использоваться и другими приложениями, часто хорошим выбором является формат JSON благодаря высокой скорости обработки и небольшому размеру. Когда об удаленных приложениях ничего не известно, формат XML даст наибольшую гарантию функциональной совместимости с ними.

Но прежде чем делать какой-либо выбор, следует проверить, доступны ли данные. Если они уже доступны, есть вероятность, что они поставляются в одном из описанных форматов, и тогда может оказаться так, что решение уже будет принято за нас.

Передача данных на сервер

До настоящего момента все наше внимание в примерах было сконцентрировано на получении **статических** файлов с данными со стороны сервера. Однако технология AJAX проявляет всю полноту своих возможностей, только когда сервер может **динамически** формировать данные, исходя из информации, получаемой от броузера. Библиотека jQuery помогает в решении и этой задачи – все методы, рассматривавшиеся до сих пор, могут отправлять данные на сервер, превращая канал связи в улицу с двусторонним движением.

Примечание

Поскольку для демонстрации этих приемов необходима возможность взаимодействия с веб-сервером, мы впервые будем использовать программный код, выполняющийся на сервере. Данные примеры написаны на языке сценариев **PHP**, свободно доступном и получившем широкое распространение. Мы не будем рассматривать вопросы, связанные с настройкой веб-сервера и PHP; необходимую помощь можно получить на веб-сайтах проектов Apache (*http://apache.org/*) и PHP (*http://php.net/*) или в компании, которая предоставляет услуги хостинга для вашего веб-сервера.

Выполнение запроса GET

Чтобы продемонстрировать порядок взаимодействия между клиентом и сервером, мы напишем сценарий, который будет возвращать по одной словарной статье на каждый запрос. Выбор статьи зависит от параметров, передаваемых броузером. Наш сценарий будет хранить свои данные во внутренней структуре, такой как показано ниже:

```
<?php
$entries = array(
  'EAVESDROP' => array(
    'part' => 'v.i.',
    'definition' => 'Secretly to overhear a catalogue of the
      crimes and vices of another or yourself.',
    'quote' => array(
      'A lady with one of her ears applied',
      'To an open keyhole heard, inside,',
      'Two female gossips in converse free &mdash;',
      'The subject engaging them was she.',
      '"I think," said one, "and my husband thinks',
      'That she\'s a prying, inquisitive minx!"',
      'As soon as no more of it she could hear',
      'The lady, indignant, removed her ear.',
      '"I will not stay," she said, with a pout,',
      '"To hear my character lied about!"',
    ),
    'author' => 'Gopete Sherany',
  ),
  'EDIBLE' => array(
    'part' => 'adj.',
    'definition' => 'Good to eat, and wholesome to digest, as
      a worm to a toad, a toad to a snake, a snake to a pig,
      a pig to a man, and a man to a worm.',
  ),
  'EDUCATION' => array(
    'part' => 'n.',
    'definition' => 'That which discloses to the wise and
      disguises from the foolish their lack of
      understanding.',
  ),
);
?>
```

Окончательная версия этого примера может хранить информацию в базе данных и загружать ее по мере необходимости. Так как в нашем случае данные являются частью сценария, программный код, извлекающий их, выглядит очень просто. Сценарий проверяет полученные данные и конструирует фрагмент HTML для отображения:

```
<?php
$term = strtoupper($_REQUEST['term']);
```

```
if (isset($entries[$term])) {
  $entry = $entries[$term];

  $html = '<div class="entry">';
  $html .= '<h3 class="term">';
  $html .= $term;
  $html .= '</h3>';
  $html .= '<div class="part">';
  $html .= $entry['part'];
  $html .= '</div>';
  $html .= '<div class="definition">';
  $html .= $entry['definition'];
  if (isset($entry['quote'])) {
    $html .= '<div class="quote">';
    foreach ($entry['quote'] as $line) {
      $html .= '<div class="quote-line">'. $line .'</div>';
    }
    if (isset($entry['author'])) {
      $html .= '<div class="quote-author">'. $entry['author'] .'</div>';
    }
    $html .= '</div>';
  }
  $html .= '</div>';

  $html .= '</div>';
  print($html);
}
?>
```

Теперь, если послать запрос этому сценарию, который мы назвали `e.php`**, он вернет фрагмент HTML, соответствующий термину, отправленному на сервер в виде параметра запроса GET. Например, при попытке обратиться к сценарию, как** `e.php?term=eavesdrop`**, мы получим обратно фрагмент, как показано на рис. 6.8.**

EAVESDROP

v.i.
Secretly to overhear a catalogue of the crimes and vices of another or yourself.
A lady with one of her ears applied
To an open keyhole heard, inside,
Two female gossips in converse free —
The subject engaging them was she.
"I think," said one, "and my husband thinks
That she's a prying, inquisitive minx!"
As soon as no more of it she could hear
The lady, indignant, removed her ear.
"I will not stay," she said, with a pout,
"To hear my character lied about!"
Gopete Sherany

Рис. 6.8. *Фрагмент HTML, соответствующий единственному термину*

Опять же обратите внимание на отсутствие форматирования, это обусловлено тем, что к данному фрагменту не применялись стили CSS, как и в примере с фрагментом HTML, что приводился выше.

Так как наша цель состоит в том, чтобы показать, как передавать данные на сервер, мы будем использовать другой способ запроса, отличный от предыдущего, который был основан на использовании отдельных ссылок. Вместо этого мы создадим список ссылок, по одной для каждого термина, а щелчок на любой из них будет вызывать загрузку соответствующего определения. Разметка HTML, которую мы добавили для этой цели, приводится ниже:

```
<div class="letter" id="letter-e">
  <h3>E</h3>
  <ul>
    <li><a href="e.php?term=Eavesdrop">Eavesdrop</a></li>
    <li><a href="e.php?term=Edible">Edible</a></li>
    <li><a href="e.php?term=Education">Education</a></li>
    <li><a href="e.php?term=Eloquence">Eloquence</a></li>
    <li><a href="e.php?term=Elysium">Elysium</a></li>
    <li><a href="e.php?term=Emancipation">Emancipation</a></li>
    <li><a href="e.php?term=Emotion">Emotion</a></li>
    <li><a href="e.php?term=Envelope">Envelope</a></li>
    <li><a href="e.php?term=Envy">Envy</a></li>
    <li><a href="e.php?term=Epitaph">Epitaph</a></li>
    <li><a href="e.php?term=Evangelist">Evangelist</a></li>
  </ul>
</div>
```

Теперь необходимо написать программный код JavaScript, который будет вызывать сценарий PHP с корректными параметрами. Для извлечения данных мы могли бы использовать обычный механизм `.load()`, добавив строку запроса прямо в адрес URL, например `e.php?term=eavesdrop`. Однако вместо этого мы можем создавать строку запроса средствами библиотеки jQuery, опираясь на отображение с данными, которое будет передаваться функции `$.get()`:

```
$(document).ready(function() {
  $('#letter-e a').click(function() {
    $.get('e.php', {'term': $(this).text()}, function(data) {
      $('#dictionary').html(data);
    });
    return false;
  });
});
```

Теперь, когда мы уже видели другие функции поддержки AJAX, предоставляемые библиотекой jQuery, обращение к этой функции выглядит знакомым. Единственное отличие заключается во втором аргументе, который позволяет нам передавать функции отображения ключей и значений для включения их в строку запроса. В данном случае клю-

чом всегда будет слово `term`, а значением – текст ссылки. Теперь щелчок на первой ссылке в списке заставит появиться соответствующее ей определение, как показано на рис. 6.9.

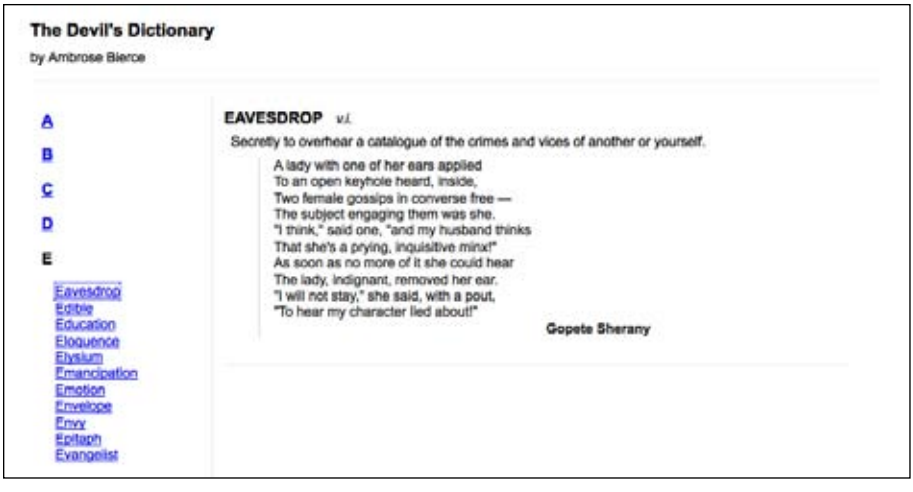


Рис. 6.9. *Вид страницы после загрузки фрагмента HTML, соответствующего указанному термину*

Для каждой ссылки в этом списке определен прямой адрес доступа к соответствующему определению, хотя мы и не используем эти адреса в программном коде. Таким способом обеспечивается альтернативный способ навигации по информации для тех, у кого поддержка JavaScript в броузере отключена или недоступна (разновидность приема **прогрессивного улучшения**). Чтобы предотвратить возможность перехода по ссылкам в результате щелчка, как это предусмотрено по умолчанию, обработчик события возвращает значение `false`.

Выполнение запроса POST

Запросы HTTP, использующие метод **POST**, почти идентичны запросам, использующим метод GET. Одно из наиболее заметных отличий заключается в том, что при использовании метода GET параметры помещаются в строку запроса, которая является частью строки адреса URL, тогда как при использовании метода POST этого не происходит. Однако при работе с методами поддержки технологии AJAX даже это отличие не заметно для обычного пользователя. Вообще говоря, единственной причиной, определяющей выбор того или иного метода, является соответствие правилам, устанавливаемым серверным программным кодом, или необходимость передавать большие объемы данных – метод GET имеет более строгие ограничения. Наш сценарий на языке PHP одинаково хорошо справляется с любым из этих методов, благодаря этому мы

легко можем перейти от метода GET к методу POST, просто изменив вызов функции jQuery:

```
$(document).ready(function() {
  $('#letter-e a').click(function() {
    $.post('e.php', {'term': $(this).text()}, function(data) {
      $('#dictionary').html(data);
    });
    return false;
  });
});
```

Аргументы функции остались теми же самыми, но сам запрос теперь будет выполняться методом POST. Можно еще больше упростить сценарий, использовав метод `.load()`, который по умолчанию использует метод POST, когда ему передается отображение параметров:

```
$(document).ready(function() {
  $('#letter-e a').click(function() {
    $('#dictionary').load('e.php', {'term': $(this).text()});
    return false;
  });
});
```

Эта упрощенная версия действует точно так же, когда пользователь щелкнет на ссылке, как показано на рис. 6.10.

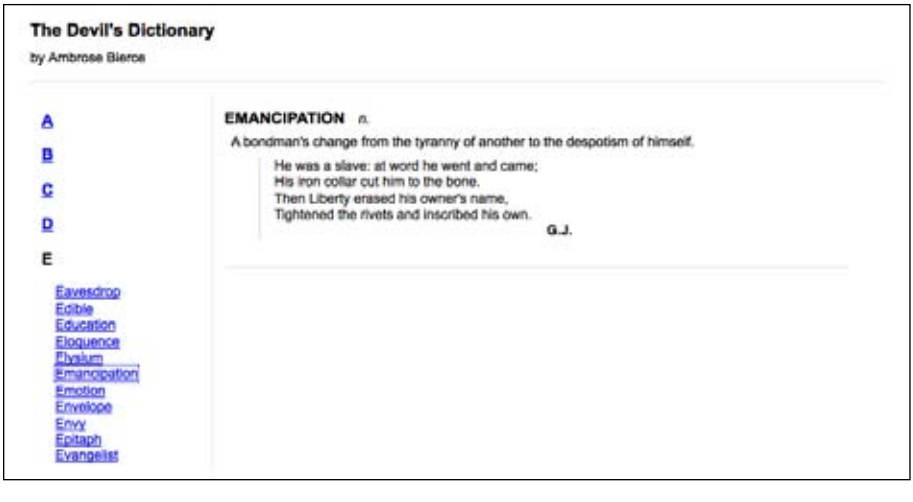


Рис. 6.10. *Вид страницы после выполнения запроса методом POST*

Сериализация формы

Передача данных на сервер часто связана с заполнением **форм** пользователем. Вместо того чтобы полагаться на обычный механизм отправ-

ки формы, который выполняет загрузку ответа сервера в окно броузера как отдельную страницу, мы можем использовать средства поддержки технологии AJAX, входящие в библиотеку jQuery, чтобы отправлять данные формы асинхронно и помещать возвращаемые данные внутрь текущей страницы.

Чтобы продемонстрировать такую возможность, создадим простую форму:

```
<div class="letter" id="letter-f">
  <h3>F</h3>
  <form>
    <input type="text" name="term" value="" id="term" />
    <input type="submit" name="search" value="search" id="search" />
  </form>
</div>
```

На этот раз сценарий PHP будет возвращать несколько словарных статей, содержащих искомый термин, как подстроку в заголовке словарной статьи. Данные будут иметь ту же структуру, что и прежде, но логика их выборки будет немного иной:

```
foreach ($entries as $term => $entry) {
  if (strpos($term, strtoupper($_REQUEST['term']))
      !== FALSE) {
    $html = '<div class="entry">';

    $html .= '<h3 class="term">';
    $html .= $term;
    $html .= '</h3>';

    $html .= '<div class="part">';
    $html .= $entry['part'];
    $html .= '</div>';

    $html .= '<div class="definition">';
    $html .= $entry['definition'];
    if (isset($entry['quote'])) {
      foreach ($entry['quote'] as $line) {
        $html .= '<div class="quote-line">'. $line .'</div>';
      }
      if (isset($entry['author'])) {
        $html .= '<div class="quote-author">'.
          $entry['author'] .'</div>';
      }
    }
    $html .= '</div>';
    $html .= '</div>';
    print($html);
  }
}
```

Функция `strpos()` проверяет наличие в слове искомой подстроки. Теперь мы можем реализовать реакцию на событие отправки формы и подготовить параметры запроса, получив данные из дерева DOM:

```
$(document).ready(function() {
  $('#letter-f form').submit(function() {
    $('#dictionary').load('f.php',
      {'term': $('input[name="term"]').val()});
    return false;
  });
});
```

Эта реализация дает желаемый эффект, но поиск полей ввода по имени и добавление их значений в отображение выглядит довольно громоздко. В частности, такая реализация плохо масштабируется по мере усложнения формы. К счастью, библиотека jQuery предлагает упрощенную реализацию этой часто встречающейся идиомы. Метод `.serialize()` воздействует на объект jQuery и преобразует соответствующие элементы DOM в строку запроса, которая может передаваться в составе запроса AJAX. Мы можем сделать наш обработчик события отправки формы более универсальным, как показано ниже:

```
$(document).ready(function() {
  $('#letter-f form').submit(function() {
    $.get('f.php', $(this).serialize(), function(data) {
      $('#dictionary').html(data);
    });
    return false;
  });
});
```

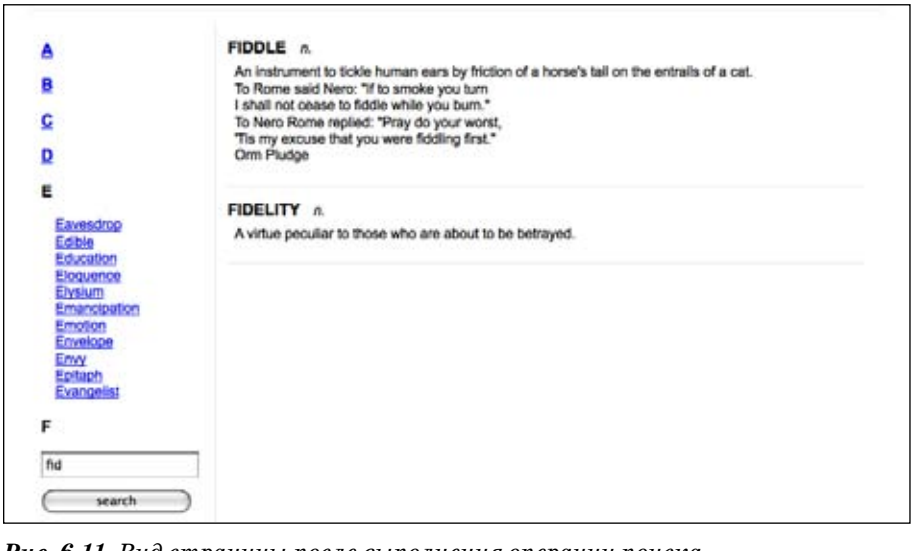


Рис. 6.11. *Вид страницы после выполнения операции поиска*

Теперь тот же самый сценарий сможет отправлять данные формы даже при увеличении количества полей в форме. После выполнения поиска на странице отображаются найденные словарные статьи, как показано на рис. 6.11.

Слежение за ходом выполнения запроса

До настоящего момента нам было достаточно просто вызвать метод поддержки AJAX и терпеливо ожидать получения ответа. Однако иногда бывает желательно знать немного больше о том, как протекает процесс выполнения запроса HTTP. На этот случай в библиотеке jQuery имеется несколько функций, используемых для регистрации **функций обратного вызова**, которые в свою очередь будут вызываться для обработки различных событий, связанных с технологией AJAX.

Двумя примерами таких функций являются методы `.ajaxStart()` и `.ajaxStop()`, которые могут присоединяться к любому объекту jQuery. Когда производится вызов метода поддержки AJAX, но при этом фактическая передача данных еще не была осуществлена, запускается функция обратного вызова, зарегистрированная с помощью метода `.ajaxStart()`. И наоборот, когда завершается выполнение последнего активного запроса, запускается функция обратного вызова, зарегистрированная с помощью метода `.ajaxStop()`. Все функции слежения являются **глобальными**, в том смысле, что они вызываются при любых взаимодействиях AJAX независимо от того, какой программный код инициировал их.

Мы можем использовать эти методы для предоставления пользователю обратной связи в случае медленных сетевых соединений. Разметка HTML страницы может содержать для этого соответствующее сообщение о загрузке:

```
<div id="loading">
  Loading...
</div>
```

Данное сообщение – всего лишь фрагмент произвольной разметки HTML. Оно, например, может содержать анимированное изображение в формате GIF, имитирующее **ход выполнения операции**. В данном примере мы добавили в таблицу CSS несколько простых стилей, чтобы при отображении сообщения страница выглядела, как показано на рис. 6.12.

Однако, следуя духу **прогрессивного улучшения**, мы не будем добавлять эту разметку HTML в страницу. Она обретает смысл, только когда в броузере включена поддержка JavaScript, поэтому мы будем вставлять ее с помощью jQuery:

```
$(document).ready(function() {
  $('<div id="loading">Loading...</div>')
```

```
    .insertBefore('#dictionary')
});
```

В нашем файле CSS для этого элемента `<div>` определено правило `display: none;`, поэтому изначально это сообщение скрыто. Чтобы отобразить его в нужный момент времени, мы просто зарегистрируем функцию обратного вызова с помощью метода `.ajaxStart()`:

```
$(document).ready(function() {
  $('<div id="loading">Loading...</div>')
    .insertBefore('#dictionary')
    .ajaxStart(function() {
      $(this).show();
    });
});
```

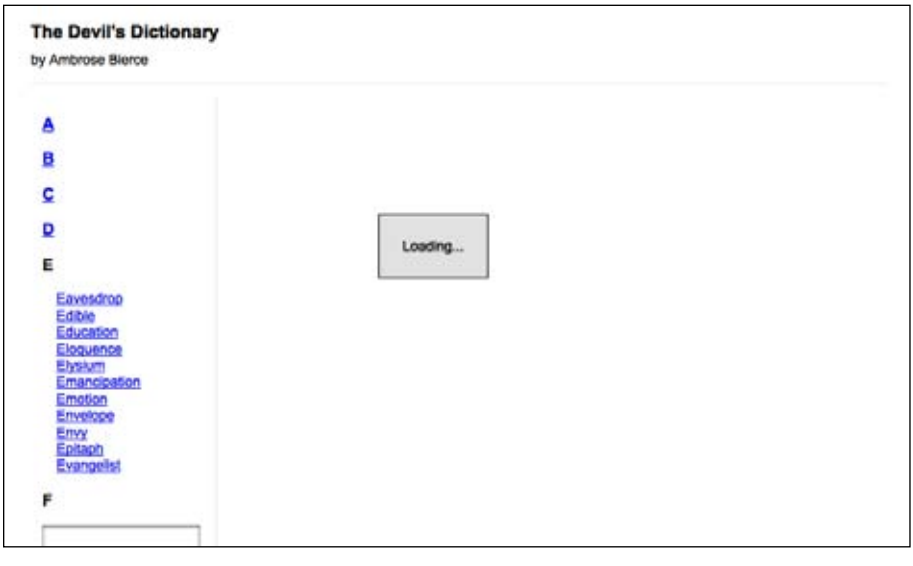


Рис. 6.12. *Вид сообщения о выполнении загрузки*

Прямо в эту же **цепочку** мы можем добавить реализацию сокрытия сообщения:

```
$(document).ready(function() {
  $('<div id="loading">Loading...</div>')
    .insertBefore('#dictionary')
    .ajaxStart(function() {
      $(this).show();
    }).ajaxStop(function() {
      $(this).hide();
    });
});
```

Вот и все! Мы реализовали вывод сообщения на время загрузки.

Еще раз отметим, что эти методы никак не связаны с конкретными способами запуска взаимодействий AJAX. Метод `.load()`, подключенный к ссылке A и метод `.getJSON()`, подключенный к ссылке B, оба будут вызывать выполнение этих действий.

В данном случае такое глобальное поведение является желательным. Однако, на тот случай, когда требуется более высокая избирательность, в нашем распоряжении имеется несколько возможностей. Некоторые из методов контроля, такие как `.ajaxError()`, передают ссылку на свою функцию обратного вызова объекту `XMLHttpRequest`. Это может использоваться для различения запросов и реализации различных эффектов. Еще большая избирательность может быть достигнута за счет использования низкоуровневой функции `$.ajax()`, которая будет рассматриваться немного ниже.

Однако наиболее типичный способ взаимодействия с запросом заключается в реализации функции обратного вызова, вызываемой в случае успешного его завершения, которую мы уже рассмотрели. Мы использовали эту функцию в некоторых предыдущих примерах для обработки данных, получаемых от сервера, и заполнения страницы результатами. Конечно же, она может использоваться и для предоставления обратной связи. Вернемся к нашему примеру, где используется метод `.load()`:

```
$(document).ready(function() {
  $('#letter-a a').click(function() {
    $('#dictionary').load('a.html');
    return false;
  });
});
```

Мы можем его немного улучшить, добавив эффект проявления, чтобы новое содержимое появлялось на странице не мгновенно, а постепенно. Метод `.load()` может принимать функцию обратного вызова, которая будет запускаться в момент завершения запроса:

```
$(document).ready(function() {
  $('#letter-a a').click(function() {
    $('#dictionary').hide().load('a.html', function() {
      $(this).fadeIn();
    });
    return false;
  });
});
```

Сначала мы скрыли требуемый элемент и инициировали загрузку. По завершении загрузки с помощью функции обратного вызова мы отображаем элемент с новыми данными, постепенно проявляя его.

AJAX и события

Предположим, что для каждого термина необходимо обеспечить возможность управления отображением определения, следующего за ним, – щелчок мышью на термине должен приводить к отображению или сокрытию связанного с ним определения. Используя приемы, с которыми мы уже познакомились, добиться этого будет совсем несложно:

```
$(document).ready(function() {
  $('.term').click(function() {
    $(this).siblings('.definition').slideToggle();
  });
});
```

Когда пользователь щелкнет на термине, это программный код отыщет братские элементы с классом `definition` и свернет их или развернет.

Похоже, что все сделано правильно, но при такой реализации щелчок мышью не будет давать ожидаемого эффекта. К сожалению, в момент подключения обработчика события `click` к терминам сами термины еще не включены в состав документа. Даже если бы нам удалось подключить к этим элементам обработчики события `click`, после щелчка на другой ссылке, вызывающей загрузку нового списка терминов, эти обработчики окажутся отключенными.

Эта проблема характерна для областей страницы, заполняемых средствами AJAX. Распространенное решение этой проблемы заключается в **повторном подключении** обработчиков всякий раз, когда происходит обновление этой области страницы. Однако такое решение может оказаться громоздким, так как программный код, выполняющий подключение обработчиков, необходимо вызывать всякий раз, когда изменяется структура дерева DOM страницы.

Более предпочтительная альтернатива была представлена в главе 3: мы можем использовать прием **делегирования событий**, привязывая фактический обработчик события к элементу-предку, который никогда не изменяется. В данном случае мы подключим обработчик события `click` к элементу `document` с помощью метода `.live()` и будем перехватывать события, как показано ниже:

```
$(document).ready(function() {
  $('.term').live('click', function() {
    $(this).siblings('.definition').slideToggle();
  });
});
```

Метод `.live()` предписывает броузеру отслеживать все события `click`, которые могут возникать в любом месте страницы. При этом обработчик события будет вызываться, только если элемент соответствует селектору `.term`. Теперь сворачивание и разворачивание описания любого

термина будет обеспечено даже в случае, когда словарная статья добавляется более поздней операцией.

Ограничения безопасности

При всей своей значимости для разработки динамических веб-приложений объект `XMLHttpRequest` (технология, лежащая в основе реализации поддержки AJAX в библиотеке jQuery) подчиняется строгим ограничениям. Чтобы предотвратить возможность атак типа **межсайтовый скриптинг**, этот объект не дает возможность запрашивать документы с сервера, отличного от того, где была получена оригинальная страница.

В большинстве случаев это правильно. Например, некоторые считают реализацию интерпретации данных в формате JSON с помощью функции eval() небезопасной. Если в файле с данными будет присутствовать злонамеренный программный код, он будет выполнен при вызове функции `eval()`. Однако так как файл с данными должен находиться на том же сервере, что и веб-страница, вероятность внедрения программного кода в файл данных практически равна вероятности внедрения этого кода в саму страницу. Это означает, что при загрузке файлов в формате JSON из доверенных источников использование функции `eval()` не представляет серьезной угрозы безопасности.

Тем не менее, часто бывает целесообразно загружать данные из сторонних источников. Чтобы обеспечить такую возможность и обойти ограничения безопасности, имеется несколько способов.

Один из методов заключается в том, чтобы сервер сам загружал данные из удаленного источника и передавал их клиенту по запросу. Это очень мощный прием, так как в случае необходимости на сервере можно организовать предварительную обработку данных. Например, мы могли бы загружать из разных источников файлы XML, содержащие ленты новостей RSS, собирать их в единую ленту на сервере и предоставлять этот новый файл клиенту по запросу.

Чтобы загрузить данные из удаленного источника *без* привлечения сервера, нам придется действовать исподтишка. Распространенный способ загрузки посторонних файлов JavaScript заключается во внедрении тегов `<script>` по мере необходимости. Это легко реализовать благодаря возможности с помощью библиотеки jQuery вставлять новые элементы в дерево DOM:

```
$(document.createElement('script'))
  .attr('src', 'http://example.com/example.js')
  .appendTo('head');
```

Фактически метода `$.getScript()` будет автоматически использовать описанную методику при обнаружении имени удаленного сервера в сво-

ем аргументе с адресом URL, поэтому все, что необходимо, делается без нашего участия.

Броузер выполнит загруженный сценарий, но он не имеет механизма, который позволил бы получить результаты работы этого сценария. По этой причине, чтобы использовать данный прием, необходимо содействие удаленного сервера. Загруженный сценарий должен предпринять некоторое действие, например определить значение глобальной переменной, чтобы оказать воздействие на окружение. Службы, предоставляющие сценарии, которые запускаются таким способом, кроме того предоставляют описание прикладного интерфейса (API), посредством которого обеспечивается взаимодействие с удаленным сценарием.

Другая возможность состоит в применении тега `<iframe>` для загрузки удаленных данных. Этот элемент позволяет указывать в качестве источника данных любой адрес URL, даже если он не совпадает с адресом сервера, откуда была получена оригинальная страница. С помощью этого тега легко можно загружать и отображать данные на странице. Однако для обеспечения возможности манипулирования этими данными, как правило, требуется такое же содействие удаленного сервера, как и в случае с использованием тега `<script>`, – сценарии внутри тега `<iframe>` должны явно передавать данные в объекты родительского документа.

Использование формата JSONP для удаленных данных

Идея использования тегов `<script>` для получения файлов JavaScript из удаленных источников также может быть распространена на получение файлов в формате JSON. Однако для этого нужно немного изменить файл JSON на сервере. Для этого имеется несколько механизмов, один из которых поддерживается библиотекой jQuery: **JSON with Padding (JSON с дополнением)**, или **JSONP**.

Файл формата JSONP содержит в себе стандартный файл в формате JSON, окруженный круглыми скобками, которым предшествует произвольная текстовая строка. Эта строка, или «padding» (дополнение), определяется клиентом, запрашивающим данные. Благодаря круглым скобкам клиент может либо организовать вызов функции, либо установить значение переменной, в зависимости от того, какой вид имеет строка дополнения.

Реализация приема JSONP на языке PHP выглядит очень просто:

```
<?php
  print($_GET['callback'] .'('. $data .')');
?>
```

Где `$data` – это переменная, хранящая строковое представление содержимого файла JSON. Когда этот сценарий вызывается, он вставляет перед данными параметр `callback` из строки запроса и возвращает получившийся файл обратно клиенту.

Для демонстрации этого приема нам достаточно будет лишь немного изменить предыдущий пример использования файлов JSON, чтобы обеспечить возможность получения этих данных из удаленного источника. Чтобы добиться желаемого эффекта, в функции `$.getJSON()` используется специальный символ-заполнитель `?`.

```
$(document).ready(function() {
  var url = 'http://examples.learningjquery.com/jsonp/g.php';
  $('#letter-g a').click(function() {
    $.getJSON(url + '?callback=?', function(data) {
      $('#dictionary').empty();
      $.each(data, function(entryIndex, entry) {
        var html = '<div class="entry">';
        html += '<h3 class="term">' + entry['term'] + '</h3>';
        html += '<div class="part">' + entry['part'] + '</div>';
        html += '<div class="definition">';
        html += entry['definition'];
        if (entry['quote']) {
          html += '<div class="quote">';
          $.each(entry['quote'], function(lineIndex, line) {
            html += '<div class="quote-line">' + line + '</div>';
          });
          if (entry['author']) {
            html += '<div class="quote-author">'
              + entry['author'] + '</div>';
          }
          html += '</div>';
        }
        html += '</div>';
        html += '</div>';
        $('#dictionary').append(html);
      });
    });
    return false;
  });
});
```

В обычных условиях нам не было бы позволено получить файл JSON от удаленного сервера (в данном случае – `examples.learningjquery.com`). Однако так как запрашиваемый файл содержит данные в формате JSONP, мы можем получить данные, добавив в конец адреса URL строку запроса, используя символ-заполнитель `?` для значения аргумента `callback`. При выполнении запроса jQuery заменит символ `?`, обработает результат и передаст его функции обратного вызова, как если бы был выполнен запрос на получение локального файла JSON.

Обратите внимание: в этом случае справедливыми остаются те же самые предостережения, что и прежде, – все, что сервер решит вернуть броузеру, будет выполнено на компьютере пользователя. Прием, основанный на использовании формата JSONP, должен использоваться только для получения данных из *доверенных* источников.

Дополнительные возможности

Арсенал методов AJAX в библиотеке jQuery обладает широкими возможностями. Мы рассмотрели некоторые из имеющихся методов, но лишь коснулись самой поверхности. Несмотря на то, что круг возможностей намного шире, чем можно было бы охватить здесь, тем не менее, мы коротко рассмотрим некоторые из наиболее важных способов реализации взаимодействий с применением технологии AJAX.

Низкоуровневый метод AJAX

Мы рассмотрели несколько методов, инициирующих обмен данными. Однако внутри библиотека jQuery отображает все эти методы на различные варианты вызова глобальной функции `$.ajax()`. Вместо реализации какого-то определенного типа взаимодействий AJAX эта функция принимает массив параметров, определяющих ее поведение.

Наш первый пример выполнял загрузку фрагмента HTML с помощью вызова `$('#dictionary').load('a.html')`. То же самое действие может быть реализовано с помощью функции `$.ajax()`, как показано ниже:

```
$.ajax({
  url: 'a.html',
  type: 'GET',
  dataType: 'html',
  success: function(data) {
    $('#dictionary').html(data);
  }
});
```

Нам потребовалось явно определить метод запроса, тип возвращаемых данных и функцию обработки этих данных. Очевидно, что это не самый эффективный пример использования усилий программиста, однако, за счет дополнительных усилий обретается большая гибкость. В число дополнительных возможностей, которые приобретаются при использовании низкоуровневой функции `$.ajax()`, входят :

- Предотвращение кэширования броузером ответов сервера. Это может оказаться полезным, когда сервер генерирует данные динамически.
- Регистрация отдельных функций обратного вызова для случаев успешного выполнения запроса, возникновения ошибки или для всех случаев сразу.
- Подавление глобальных обработчиков (например, тех, что были зарегистрированы с помощью функции `$.ajaxStart()`), которые обычно вызываются всеми методами поддержки AJAX.
- Передача имени пользователя и пароля для аутентификации на удаленном сервере.

За подробным описанием этих и других возможностей обращайтесь к руководству «jQuery Reference Guide» или к справочнику по прикладному программному интерфейсу (*http://docs.jquery.com/Ajax/jQuery.ajax*).

Изменение значений параметров по умолчанию

Функция `$.ajaxSetup()` позволяет изменять значения по умолчанию для всех параметров, используемых методами поддержки AJAX. Она принимает отображение параметров, идентичное тому, что передается функции `$.ajax()`, и обеспечивает использование этих значений при выполнении всех последующих запросов AJAX, если они не будут переопределены.

```
$.ajaxSetup({
  url: 'a.html',
  type: 'POST',
  dataType: 'html'
});
$.ajax({
  type: 'GET',
  success: function(data) {
    $('#dictionary').html(data);
  }
});
```

Этот программный код выполняет те же операции, что и предыдущий пример использования функции `$.ajax()`. Обратите внимание, что адрес URL запроса определяется вызовом функции `$.ajaxSetup()` как значение по умолчанию, поэтому его можно не указывать при вызове функции `$.ajax()`. Параметр `type` напротив имеет значение по умолчанию POST, тем не менее, существует возможность переопределить его при вызове функции `$.ajax()` и заменить на GET.

Загрузка частей страницы HTML

Первый и самый простой пример использования технологии AJAX, который мы рассматривали, запрашивал фрагмент разметки HTML и вставлял его в страницу. Однако иногда на сервере уже имеется необходимый нам код HTML, но он заключен в страницу HTML, которая нам не нужна. Когда на стороне сервера нет возможности реализовать передачу данных в желаемом нами формате, библиотека jQuery может оказать нам помощь на стороне клиента.

Рассмотрим случай, напоминающий наш первый пример, но в отличие от него документ, содержащий словарные статьи, представляет собой законченную страницу HTML, как показано ниже:

```
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
```

```
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8"/>
    <title>The Devil's Dictionary: H</title>

    <link rel="stylesheet" href="dictionary.css"
                           type="text/css" media="screen" />
  </head>
  <body>
    <div id="container">
      <div id="header">
        <h2>The Devil's Dictionary: H</h2>
        <div class="author">by Ambrose Bierce</div>
      </div>

      <div id="dictionary">
        <div class="entry">
          <h3 class="term">HABEAS CORPUS</h3>
          <div class="part">n.</div>
          <div class="definition">
            A writ by which a man may be taken out of jail
            when confined for the wrong crime.
          </div>
        </div>

        <div class="entry">
          <h3 class="term">HABIT</h3>
          <div class="part">n.</div>
          <div class="definition">
            A shackle for the free.
          </div>
        </div>
      </div>
    </div>
  </body>
</html>
```

Мы можем загрузить страницу целиком, используя программный код, написанный ранее:

```
$(document).ready(function() {
  $('#letter-h a').click(function() {
    $('#dictionary').load('h.html');
    return false;
  });
});
```

Однако это породит странный эффект из-за наличия частей страницы HTML, которые нам не нужны, как показано на рис. 6.13.

Чтобы удалить эти лишние части, мы можем задействовать новую для нас особенность метода `.load()`. Вместе с адресом URL загружаемого документа можно указать селектор jQuery. При наличии селектора он ис-

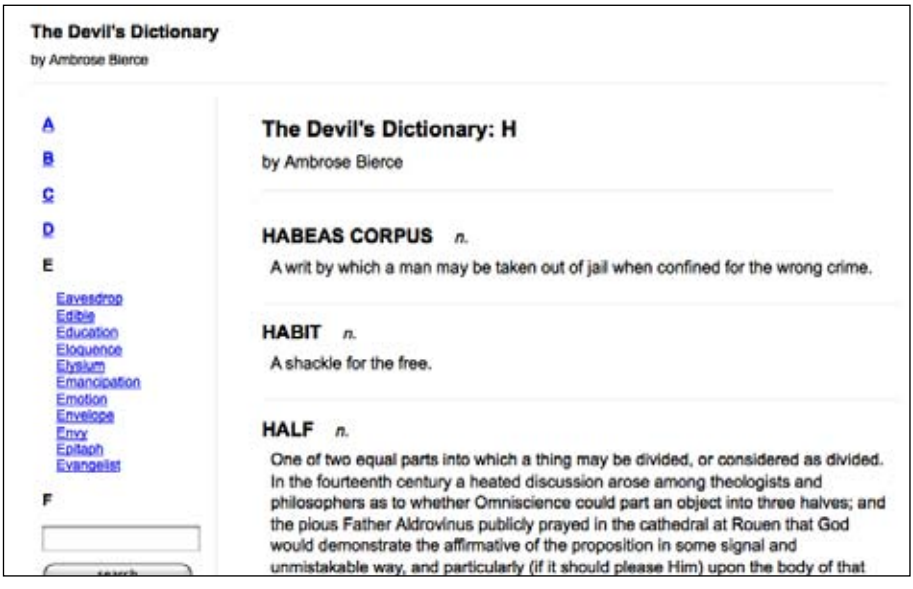


Рис. 6.13. *Появление лишних фрагментов HTML в странице*

пользуется для выделения частей загруженного документа. Благодаря этому в страницу будут вставлены только те части, которые соответствуют селектору. В данном случае мы применим этот прием, чтобы извлечь из документа и вставить только словарные статьи:

```
$(document).ready(function() {
  $('#letter-h a').click(function() {
    $('#dictionary').load('h.html .entry');
    return false;
  });
});
```

Теперь ненужные части документа не попадут в страницу, как показано на рис. 6.14.

В заключение

Мы узнали, что методы поддержки технологии AJAX, предоставляемые библиотекой jQuery, могут помочь нам загрузить с сервера данные в различных форматах, устраняя необходимость обновлять всю страницу. Мы можем по мере необходимости загружать и выполнять сценарии и отправлять данные на сервер.

Мы также узнали, как обходить типичные проблемы, вызванные асинхронной природой загрузки, например, как сохранить привязку обработчиков событий после выполнения загрузки и как загружать данные из сторонних источников.

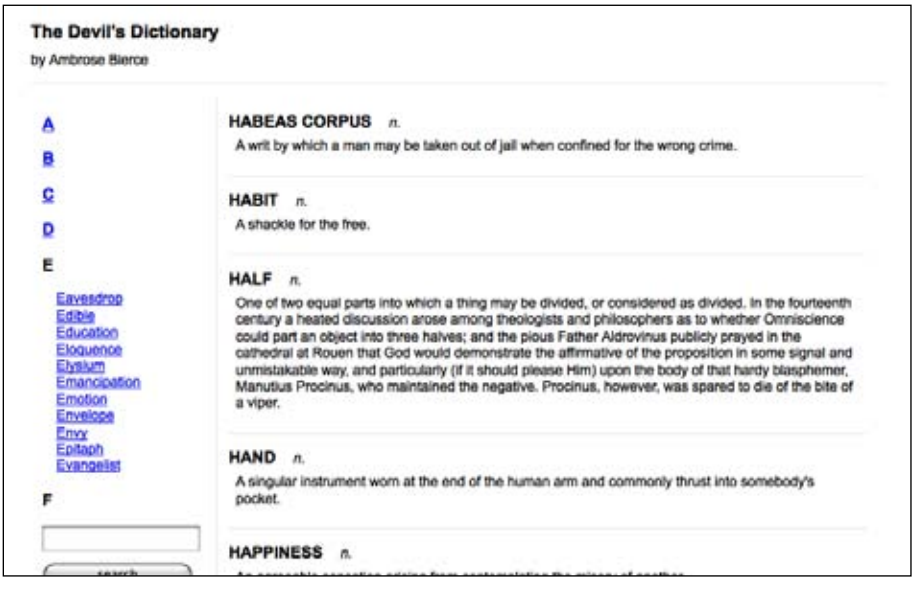


Рис. 6.14. *Вид страницы после удаления ненужных фрагментов*

На этом заканчивается учебный раздел книги. Теперь у нас на вооружении имеются основные инструменты, предлагаемые библиотекой jQuery: селекторы, события, эффекты, методы манипулирования деревом DOM и выполнения асинхронных запросов к серверу. Но это еще не все, чем может помочь нам библиотека jQuery. В последующих главах мы рассмотрим некоторые из большого числа возможностей, поставляемых с помощью модулей расширения библиотеки. Но сначала исследуем несколько комбинаций из уже известных приемов, которые позволят дополнить наши веб-страницы новыми и интересными возможностями.

7

Работа с таблицами

В первых шести главах мы исследовали библиотеку jQuery на серии учебных материалов, основное внимание в которых уделялось отдельным компонентам jQuery, и использовали примеры для демонстрации этих компонентов в действии. В главах с 7-й по 9-ю мы подойдем с другого конца – сначала мы будем рассматривать примеры реальных проблем, а затем знакомиться с методами их решения с помощью jQuery.

Здесь в качестве примера веб-сайта мы возьмем книжный интернет-магазин, однако приемы, обсуждаемые в этих главах, могут применяться к разнообразным сайтам, от блогов до каталогов документов, от бизнес-сайтов до сайтов в корпоративных сетях. В главах 7 и 8 все свое внимание мы уделим двум основным компонентам большинства сайтов – **таблицам** и **формам**, а в главе 9 исследуем пару способов улучшения визуального представления информации с применением эффектов **прокрутки** и **перемещения**.

Благодаря ускоряющемуся развитию веб-стандартов в последние несколько лет табличная верстка стала все больше уступать место верстке, основанной на применении CSS. Несмотря на то, что в 90-х годах прошлого века таблицы часто применялись как вынужденная мера для реализации многоколоночных и других сложных макетов размещения элементов, они никогда не предназначались для использования таким образом. Напротив, технология CSS специально разрабатывалась для решения подобных задач представления информации.

Однако здесь не стоит разворачивать дискуссию о роли таблиц. Заметим лишь, что в этой главе мы будем использовать библиотеку jQuery с целью применения приемов, увеличивающих удобочитаемость, удобство в использовании и привлекательность контейнеров, семантически предназначенных для представления **табличных данных**. Для желающих поближе познакомиться с взглядами на семантику таблиц HTML можно порекомендовать статью «Bring on the Tables» в блоге Родже-

ра Йоханссона (Roger Johansson), по адресу: *http://www.456bereastreet.com/archive/200410/bring_on_the_tables/*.[1]

Реализацию некоторых из приемов, которые мы будем применять в данной главе при работе с таблицами, можно найти в модулях расширения, таких как TableSorter Кристиана Баха (Christian Bach). Дополнительную информацию можно найти в каталоге модулей расширения jQuery по адресу: *http://plugins.jquery.com/*.

В этой главе мы рассмотрим реализацию:

- Сортировки
- Разбиения на страницы
- Подсветки строк
- Всплывающих подсказок
- Свертывания и развертывания строк
- Фильтрации

Сортировка и разбивка на страницы

Сортировка и **разбивка на страницы** являются двумя наиболее типичными операциями, выполняемыми над табличными данными. Трудно переоценить практическую ценность возможности переупорядочивать информацию в больших таблицах. К сожалению, эти полезные операции могут оказаться одними из самых сложных в реализации.

Сначала мы рассмотрим, что необходимо для реализации сортировки таблиц, дающей пользователю возможность упорядочивать данные по его желанию.

Сортировка на стороне сервера

Типичное решение проблемы сортировки заключается в реализации ее на стороне сервера. Табличные данные часто извлекаются из **базы данных**, что для программного кода означает возможность запросить их с требуемым порядком сортировки (используя, например, предложение `ORDER BY` языка **SQL**). Если у нас имеется доступ к программному коду на стороне сервера, проще всего начать с установки наиболее разумного порядка сортировки по умолчанию.

Однако сортировка обретает свою значимость, когда пользователь может определять свой порядок сортировки. Наиболее часто для данной цели заголовки столбцов таблицы (`<th>`), допускающих сортировку, преобразуются в ссылки. Эти ссылки могут ссылаться на текущую страни-

[1] Перевод статьи на русский язык можно найти по адресу: *http://habrahabr.ru/blogs/webdev/31233/*. – *Прим. перев.*

цу, но при этом включать **строку запроса**, определяющую столбец, по которому должна быть выполнена сортировка:

```
<table id="my-data">
  <thead>
    <tr>
      <th class="name">
        <a href="index.php?sort=name">Name</a>
      </th>
      <th class="date">
        <a href="index.php?sort=date">Date</a>
      </th>
    </tr>
  </thead>
  <tbody>
    ...
  </tbody>
</table>
```

Сервер может реагировать на параметр в строке запроса, возвращая содержимое базы данных с иным порядком сортировки.

Предотвращение обновления всей страницы

Такой способ сортировки достаточно прост в реализации, но в случае его использования каждая операция сортировки будет приводить к полному обновлению страницы. Как мы уже знаем, библиотека jQuery позволяет избавиться от необходимости обновления страницы с помощью методов поддержки технологии AJAX. Если заголовки столбцов таблицы оформлены в виде ссылок, как было показано выше, в страницу можно добавить программный код, использующий jQuery, чтобы превратить эти ссылки в запросы AJAX:

```
$(document).ready(function() {
  $('#my-data th a').click(function() {
    $('#my-data tbody').load($(this).attr('href'));
    return false;
  });
});
```

Теперь после щелчка на ссылке jQuery будет отправлять серверу запрос AJAX на получение той же страницы. Когда для запроса страницы используются методы поддержки AJAX из библиотеки jQuery, они устанавливают HTTP-заголовок `X-Requested-With` в объекте `XMLHttpRequest`, чтобы сервер мог определить, что выполняется запрос AJAX. В ответ на запрос, в котором присутствует параметр, программный код на стороне сервера может предусматривать отправку только информационного наполнения элемента `<tbody>` без окружающей его разметки HTML. При такой реализации можно просто вставлять полученный ответ на место существующего элемента `<tbody>`.

Это пример использования принципа прогрессивного улучшения. Такая страница прекрасно работает и без наличия поддержки JavaScript в броузере, так как ссылки, позволяющие выполнять сортировку, никуда не исчезают. Однако при наличии поддержки JavaScript запросы производятся с применением технологии AJAX, что позволяет выполнять сортировку без полного обновления страницы.

Сортировка с помощью JavaScript

Однако иногда мы либо не можем ждать ответа сервера, либо не имеем доступа к программному коду на стороне сервера. В таких случаях сортировку можно полностью реализовать в броузере, на стороне клиента, с помощью сценария на языке JavaScript.

Например, предположим, что у нас имеется таблица (рис. 7.1), в которой перечислены следующие данные – названия книг, имена авторов, даты выпуска и цены:

```
<table class="sortable">
  <thead>
    <tr>
      <th></th>
      <th>Title</th>
      <th>Author(s)</th>
      <th>Publish&amp;nbsp;Date</th>
      <th>Price</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <td><img src="../images/covers/small/1847192386.png"
        width="49" height="61" alt="Building Websites with
        Joomla! 1.5 Beta 1" />
      </td>
      <td>Building Websites with Joomla! 1.5 Beta 1</td>
      <td>Hagen Graf</td>
      <td>Feb 2007</td>
      <td>$40.49</td>
    </tr>
    <tr>
      <td><img src="../images/covers/small/1904811620.png"
        width="49" height="61" alt="Learning Mambo:
        A Step-by-Step Tutorial to Building Your Website" />
      </td>
      <td>Learning Mambo: A Step-by-Step Tutorial to Building Your Website
      </td>
      <td>Douglas Paterson</td>
      <td>Dec 2006</td>
      <td>$40.49</td>
    </tr>
```

```
    </tbody>
  </table>
```

	Title	Author(s)	Publish Date	Price
	Building Websites with Joomla! 1.5 Beta 1	Hagen Graf	Feb 2007	$40.49
	Learning Mambo: A Step-by-Step Tutorial to Building Your Website	Douglas Paterson	Dec 2006	$40.49
	Moodle E-Learning Course Development	William Rice	May 2006	$35.99
	AJAX and PHP: Building Responsive Web Applications	Cristian Darie, Mihai Bucica, Filip Chereches-Toşa, Bogdan Brinzarea	Mar 2006	$31.49
	OpenVPN: Building and Integrating Virtual Private Networks	Markus Feilner	May 2006	$53.99

Рис. 7.1. *Таблица с перечнем книг*

Мы можем превратить заголовки таблицы в кнопки, щелчок на которых будет вызывать сортировку данных по соответствующим столбцам. Рассмотрим несколько способов реализации такой сортировки.

Теги группировки строк

Обратите внимание, что для разделения данных на **группы строк** мы применяли теги `<thead>` и `<tbody>`. Многие авторы разметки HTML опускают эти подразумеваемые теги, хотя они могут дать нам возможность использовать более удобные селекторы CSS. Например, предположим, что в данной таблице нам потребовалось выделить цветом чередующиеся четные/нечетные строки, причем только в теле таблицы:

```
$(document).ready(function() {
  $('table.sortable tbody tr:odd').addClass('odd');
  $('table.sortable tbody tr:even').addClass('even');
});
```

Такой код приведет к выделению цветом строк таблицы, но оставит нетронутым заголовок, как показано на рис. 7.2.

С помощью тегов группировки строк мы легко сможем собирать строки с данными и выполнять операции над ними, не оказывая влияния на заголовок.

	Title	Author(s)	Publish Date	Price
	Building Websites with Joomla! 1.5 Beta 1	Hagen Graf	Feb 2007	$40.49
	Learning Mambo: A Step-by-Step Tutorial to Building Your Website	Douglas Paterson	Dec 2006	$40.49
	Moodle E-Learning Course Development	William Rice	May 2006	$35.99
	AJAX and PHP: Building Responsive Web Applications	Cristian Darie, Mihai Bucica, Filip Chereches-Tosa, Bogdan Brinzarea	Mar 2006	$31.49
	OpenVPN: Building and Integrating Virtual Private Networks	Markus Feilner	May 2006	$53.99

Рис. 7.2. *Выделение строк таблицы цветом*

Простая сортировка в алфавитном порядке

Теперь реализуем сортировку таблицы по столбцу Title (Название). Чтобы иметь возможность выбирать ячейку в заголовке таблицы, нам необходимо определить для нее класс:

```
<thead>
  <tr>
    <th></th>
    <th class="sort-alpha">Title</th>
    <th>Author(s)</th>
    <th>Publish Date</th>
    <th>Price</th>
  </tr>
</thead>
```

Использование JavaScript для сортировки массивов

Для выполнения фактической сортировки можно использовать встроенный в JavaScript метод `.sort()`. Он выполняет сортировку массива без создания его копии и может принимать **функцию сравнения** в качестве аргумента. Эта функция сравнивает два элемента массива и должна возвращать положительное или отрицательное число в зависимости от того, какой элемент массива должен следовать раньше в отсортированном массиве.

Для примера возьмем простой массив чисел:

```
var arr = [52, 97, 3, 62, 10, 63, 64, 1, 9, 3, 4];
```

Мы можем отсортировать этот массив, вызвав метод `arr.sort()`. В результате элементы массива будут упорядочены следующим образом:

```
[1, 10, 3, 3, 4, 52, 62, 63, 64, 9, 97]
```

Как видим, по умолчанию элементы массива сортируются в **лексикографическом (**алфавитном**)** порядке. Однако в данном случае предпочтительнее было бы отсортировать элементы в **числовом** порядке. Для этого мы можем передать методу `.sort()` функцию сравнения:

```
arr.sort(function(a,b) {
  if (a < b)
    return -1;
  if (a > b)
    return 1;
  return 0;
});
```

Эта функция возвращает отрицательное число, если первым должен следовать элемент `a`, положительное число, если первым должен следовать элемент `b`, и ноль, если порядок следования элементов не имеет значения. С помощью такой функции сравнения метод `.sort()` сможет расположить элементы в требуемом порядке:

```
[1, 3, 3, 4, 9, 10, 52, 62, 63, 64, 97]
```

Использование функции сравнения для сортировки строк таблицы

Наша первая процедура сортировки выглядит, как показано ниже:

```
$(document).ready(function() {
  $('table.sortable').each(function() {
    var $table = $(this);
    $('th', $table).each(function(column) {
      var $header = $(this);
      if ($header.is('.sort-alpha')) {
        $header.addClass('clickable').hover(function() {
          $header.addClass('hover');
        }, function() {
          $header.removeClass('hover');
        }).click(function() {
          var rows = $table.find('tbody > tr').get();
          rows.sort(function(a, b) {
            var keyA = $(a).children('td').eq(column).text()
              .toUpperCase();
            var keyB = $(b).children('td').eq(column).text()
              .toUpperCase();
            if (keyA < keyB) return -1;
            if (keyA > keyB) return 1;
            return 0;
          });
          $.each(rows, function(index, row) {
```

```
                $table.children('tbody').append(row);
              });
            });
          }
        });
      });
    });
```

Первое, на что здесь следует обратить внимание, – это использование метода `.each()`, который делает итерации **явными**. Мы могли бы подключить обработчик события `click` ко всем заголовкам, содержащим класс `sort-alpha`, простым вызовом `$('table.sortable th.sort-alpha').click()`, но это не позволило бы нам легко получить важную информацию: **индекс столбца**, на заголовке которого был выполнен щелчок. Благодаря тому, что метод `.each()` передает индекс итерации своей функции обратного вызова, мы сможем использовать его позднее для поиска соответствующей ячейки в каждой строке данных.

Отыскав ячейку в заголовке, мы извлекаем массив всех строк с данными. Это прекрасный пример того, насколько удобно с помощью метода `.get()` выполнять преобразование объекта jQuery в массив узлов дерева DOM; несмотря на то что объекты jQuery во многом напоминают массивы, в них отсутствуют какие-либо свойственные массивам методы, такие как `.sort()`.

Теперь, когда у нас имеется массив узлов DOM, мы можем отсортировать его, но для этого необходимо написать подходящую функцию сравнения. Нам требуется отсортировать строки в соответствии с текстом, содержащимся в ячейках таблицы, поэтому именно эту информацию и будет рассматривать функция сравнения. Нам известно, какие ячейки необходимо сравнить, потому что мы сохранили индекс столбца в объемлющем вызове метода `.each()`. Так как сравнение строк в JavaScript выполняется **с учетом регистра символов**, а нам желательно сделать сортировку **нечувствительной к регистру символов**, мы приводим все символы к верхнему регистру. Чтобы избежать лишних вычислений, мы сохраняем значения ключей в переменных, сравниваем их и возвращаем положительное или отрицательное число, как оговаривалось выше.

Выполнив сортировку массива, мы обходим строки в цикле и вставляем их обратно в таблицу. Метод `.append()` не создает копии узлов, поэтому он не копирует их, а *перемещает*. Теперь наша таблица отсортирована.

Этот пример демонстрирует использование приема **постепенной деградации**, противоположного приему прогрессивного улучшения. В отличие от решений на основе применения технологии AJAX, обсуждавшихся выше, данное решение не может функционировать без поддержки JavaScript; здесь предполагается, что у нас нет доступа к программному коду на стороне сервера. Так как для обеспечения возможности сортировки нам необходимо наличие поддержки JavaScript, мы производим добавление класса `clickable` только в программном коде, благодаря

чему интерфейс будет указывать на возможность сортировки (за счет использования фонового изображения), только если сценарий имеет возможность работать. В случае отсутствия поддержки JavaScript в броузере функциональность страницы **деградирует:** страница по-прежнему будет выполнять свои функции, но уже без возможности сортировки.

В ходе сортировки произошла перестановка строк, вследствие чего нарушилось выделение цветом чередующихся строк, как показано на рис. 7.3.

	Title	Author(s)	Publish Date	Price
	Advanced Microsoft Content Management Server Development	Angus Logan, Stefan Goßner, Lim Mei Ying, Andrew Connell	Nov 2005	$53.99
	AJAX and PHP: Building Responsive Web Applications	Cristian Darie, Mihai Bucica, Filip Chereches-Tosa, Bogdan Brinzarea	Mar 2006	$31.49
	Alfresco Enterprise Content Management Implementation	Munwar Shariff	Jan 2007	$53.99
	BPEL Cookbook: Best Practices for SOA-based integration and composite applications development	Jerry Thomas, Doug Todd, Harish Gaur, Lawrence Pravin, Arun Poduval, The Hoa Nguyen, Yves Coene, Jeremy Bolie, Stany Blanvalet, Markus Zirn, Matjaz Juric, Sean Carey, Michael Cardella, Kevin Geminiuc, Praveen Ramachandran	Jul 2006	$40.49

Рис. 7.3. *Чередование окраски строк нарушилось*

После сортировки нам требуется вновь произвести выделение цветом чередующихся строк. Для этого можно вынести программный код, выполняющий окрашивание строк таблицы, в отдельную функцию, чтобы ее можно было вызывать по мере необходимости:

```
$(document).ready(function() {
  var alternateRowColors = function($table) {
    $('tbody tr:odd', $table)
      .removeClass('even').addClass('odd');
    $('tbody tr:even', $table)
      .removeClass('odd').addClass('even');
  };
  $('table.sortable').each(function() {
    var $table = $(this);
    alternateRowColors($table);
    $('th', $table).each(function(column) {
      var $header = $(this);
      if ($header.is('.sort-alpha')) {
        $header.addClass('clickable').hover(function() {
          $header.addClass('hover');
```

```
        }, function() {
          $header.removeClass('hover');
        }).click(function() {
          var rows = $table.find('tbody > tr').get();
          rows.sort(function(a, b) {
            var keyA = $(a).children('td').eq(column).text()
                           .toUpperCase();
            var keyB = $(b).children('td').eq(column).text()
                           .toUpperCase();
            if (keyA < keyB) return -1;
            if (keyA > keyB) return 1;
            return 0;
          });
          $.each(rows, function(index, row) {
            $table.children('tbody').append(row);
          });
          alternateRowColors($table);
        });
      }
    });
  });
});
```

Такая реализация повторно выполняет окрашивание строк таблицы, решая нашу проблему, как показано на рис. 7.4.

	Title	Author(s)	Publish Date	Price
	Advanced Microsoft Content Management Server Development	Angus Logan, Stefan Goßner, Lim Mei Ying, Andrew Connell	Nov 2005	$53.99
	AJAX and PHP: Building Responsive Web Applications	Cristian Darie, Mihai Bucica, Filip Chereches-Tosa, Bogdan Brinzarea	Mar 2006	$31.49
	Alfresco Enterprise Content Management Implementation	Munwar Shariff	Jan 2007	$53.99
	BPEL Cookbook: Best Practices for SOA-based Integration and composite applications development	Jerry Thomas, Doug Todd, Harish Gaur, Lawrence Pravin, Arun Poduval, The Hoa Nguyen, Yves Coene, Jeremy Bolie, Stany Blanvalet, Markus Zirn, Matjaz Juric, Sean Carey, Michael Cardella, Kevin Geminiuc, Praveen Ramachandran	Jul 2006	$40.49

Рис. 7.4. *Вид таблицы после повторного окрашивания строк*

Возможности модулей расширения

Написанная нами функция `alternateRowColors()` является отличным кандидатом, чтобы стать **модулем расширения** к библиотеке jQuery. Факти-

чески любую операцию, которая должна применяться к множеству элементов DOM, можно легко превратить в модуль расширения. Для этого достаточно лишь немного изменить существующую функцию:

```
jQuery.fn.alternateRowColors = function() {
  $('tbody tr:odd', this)
    .removeClass('even').addClass('odd');
  $('tbody tr:even', this)
    .removeClass('odd').addClass('even');
  return this;
};
```

Мы внесли в функцию три важных изменения.

- Мы определили ее уже не как самостоятельную функцию, а как новое свойство `jQuery.fn`. Тем самым мы зарегистрировали функцию как метод модуля расширения.
- Взамен аргумента `$table` мы использовали ключевое слово `this`. Внутри метода модуля расширения оно будет ссылаться на объект jQuery, над которым выполняется операция.
- Наконец, функция возвращает ссылку `this`. Передача объекта jQuery в качестве возвращаемого значения обеспечивает возможность включения нашего нового метода в **цепочку**.

Примечание

Дополнительную информацию о создании модулей расширения для библиотеки jQuery можно найти в главе 11. В ней мы рассмотрим вопросы создания модулей расширения, пригодных для общего использования в отличие от данного небольшого примера, применимого только в нашем собственном программном коде.

Определив новый модуль расширения, мы можем использовать более естественную для jQuery форму вызова метода `$table.alternateRowColors()` вместо `alternateRowColors($table)`.

Проблемы производительности

Наша реализация работает, но очень медленно. Проблема заключена в функции сравнения, которая выполняет слишком большой объем работы. В ходе сортировки эта функция вызывается много раз, и, значит, каждое мгновение, которое она тратит на обработку, будет увеличено многократно.

В действительности стандарты не определяют **алгоритм сортировки**, который должен использоваться в JavaScript. Он может быть простым, как алгоритм **пузырьковой сортировки** (сложность которого в наихудшей ситуации составляет $\Theta(n^2)$), или более сложным, как алгоритм **быстрой сортировки** (сложность которого в среднем составляет $\Theta(n \log n)$). В любом случае можно с уверенностью сказать, что удвоение числа эле-

ментов в массиве увеличит количество вызовов функции сравнения более чем в два раза.

Исправить ситуацию с низкой скоростью работы нашей функции можно с помощью предварительного вычисления значений сравниваемых ключей. Начнем с текущей медленной версии нашей функции сортировки:

```
rows.sort(function(a, b) {
  var keyA = $(a).children('td').eq(column).text().toUpperCase();
  var keyB = $(b).children('td').eq(column).text().toUpperCase();
  if (keyA < keyB) return -1;
  if (keyA > keyB) return 1;
  return 0;
});
```

Мы можем организовать вычисление ключей в отдельном цикле:

```
$.each(rows, function(index, row) {
  row.sortKey = $(row).children('td').eq(column).text().toUpperCase();
});
rows.sort(function(a, b) {
  if (a.sortKey < b.sortKey) return -1;
  if (a.sortKey > b.sortKey) return 1;
  return 0;
});
$.each(rows, function(index, row) {
  $table.children('tbody').append(row);
  row.sortKey = null;
});
```

В данном цикле мы выполняем все дорогостоящие операции и сохраняем результаты в новом свойстве `.sortKey`. Такого рода свойства, присоединяемые к элементу DOM, но не являющиеся обычными атрибутами DOM, называются **расширенными (expando)**. Это удобное место для хранения ключа, так как нам необходимо иметь по одному такому свойству для каждого элемента строки таблицы. Теперь внутри функции сравнения можно использовать данный атрибут, благодаря чему скорость сортировки увеличится.

Примечание

По окончании мы устанавливаем расширенное свойство в значение `null`, самостоятельно освобождая память. В данном случае так делать не совсем обязательно, но это хорошая привычка, чтобы взять ее на вооружение, так как разбросанные повсюду расширенные свойства могут стать причиной **утечки памяти**. Подробнее об этом рассказывается в Приложении С.

Библиотека jQuery предоставляет альтернативный использованию расширенных свойств механизм хранения данных, который мы можем применять. Метод `.data()` записывает или возвращает произвольную

информацию, ассоциированную с элементами страницы, а метод .removeData() уничтожает информацию, сохранявшуюся ранее:

```
$.each(rows, function(index, row) {
  $(row).data('sortKey', $(row).children('td')
    .eq(column).text().toUpperCase());
});
rows.sort(function(a, b) {
  if ($(a).data('sortKey') < $(b).data('sortKey')) return -1;
  if ($(a).data('sortKey') > $(b).data('sortKey')) return 1;
  return 0;
});
$.each(rows, function(index, row) {
  $table.children('tbody').append(row);
  $(row).removeData('sortKey');
});
```

Иногда бывает удобнее применять метод .data() вместо расширенных свойств, поскольку нам чаще приходится работать с объектами jQuery, а не с узлами дерева DOM. Кроме того, это позволяет избежать потенциальных проблем, связанных с утечками памяти в Internet Explorer. Однако далее в нашем примере мы продолжим использовать расширенные свойства, чтобы попрактиковаться в переходе от операций над узлами дерева DOM к операциям над объектами jQuery.

Особенности ключей сортировки

Теперь нам необходимо реализовать тот же вид сортировки для столбца Author(s) (Автор(ы)).Мы можем выполнить сортировку по столбцу Author(s) с помощью существующего программного кода простым добавлением класса sort-alpha к ячейке в заголовке таблицы. В идеале сортировка должна выполняться по фамилиям авторов, а не по именам. Так как некоторые книги могут быть написаны коллективами авторов, а для некоторых авторов могут указываться отчества или инициалы, нам необходимо определить, какую часть текста ячейки следует использовать в качестве ключа сортировки. Мы можем обозначить эту часть текста, обернув ее тегом:

```
<tr>
  <td><img src="../images/covers/small/1847192386.png"
    width="49" height="61" alt="Building Websites with Joomla! 1.5 Beta 1" />
  </td>
  <td>Building Websites with Joomla! 1.5 Beta 1</td>
  <td>Hagen <span class="sort-key">Graf</span></td>
  <td>Feb 2007</td>
  <td>$40.49</td>
</tr>
<tr>
  <td><img src="../images/covers/small/1904811620.png"
```

```
      width="49" height="61" alt="Learning Mambo:
      A Step-by-Step Tutorial to Building Your Website" />
    </td>
    <td>Learning Mambo: A Step-by-Step Tutorial to Building Your Website
    </td>
    <td>Douglas <span class="sort-key">Paterson</span></td>
    <td>Dec 2006</td>
    <td>$40.49</td>
  </tr>
```

Теперь следует изменить реализацию сортировки так, чтобы принять во внимание этот тег и не нарушить при этом существующий порядок сортировки по столбцу Title. Добавив перед отмеченным ключом сортировки ключ, который был вычислен ранее, мы можем отсортировать данные в таблице сначала по фамилиям авторов, если они обозначены в разметке, или в противном случае по всему тексту в ячейке:

```
$.each(rows, function(index, row) {
  var $cell = $(row).children('td').eq(column);
  row.sortKey = $cell.find('.sort-key').text().toUpperCase()
    + ' ' + $cell.text().toUpperCase();
});
```

Сейчас для сортировки по столбцу Author(s) используется предоставленный ключ, благодаря чему она выполняется по фамилиям авторов, как показано на рис. 7.5.

	Title	Author(s)	Publish Date	Price
	Programming Windows Workflow Foundation: Practical WF Techniques and Examples using XAML and C#	K. Scott Allen	Dec 2006	$40.49
	Building Websites with XOOPS : A step-by-step tutorial	Steve Atwal	Oct 2006	$26.99
	Learn OpenOffice.org Spreadsheet Macro Programming: OOoBasic and Calc automation	Dr. Mark Alexander Bain	Dec 2006	$35.99
	UML 2.0 in Action: A project-based tutorial	Philippe Baumann, Henriette Baumann, Patrick Grassle	Sep 2005	$31.49

Рис. 7.5. *Таблица, отсортированная по фамилиям авторов*

Если в соседних ячейках указаны одинаковые фамилии авторов, для разрешения конфликта сортировка выполняется по всей строке.

Сортировка данных других типов

Наши пользователи должны иметь возможность выполнять сортировку не только по столбцам Title и Author(s), но и по столбцам Publish Date (Дата выпуска) и Price (Цена). Благодаря тому что мы модернизировали функцию сравнения, она в состоянии работать с данными любых типов; но нам еще необходимо реализовать вычисление ключей сортировки для других типов данных. Например, в случае сортировки по цене необходимо удалить ведущий символ $ и преобразовать оставшуюся часть текста в число, чтобы значения ячеек сравнивались как числа:

```
var key = parseFloat($cell.text().replace(/^[^\d.]*/, ''));
row.sortKey = isNaN(key) ? 0 : key;
```

Используемое здесь регулярное выражение удаляет любые ведущие символы, не являющиеся цифрами или десятичной точкой, и результат передается функции `parseFloat()`. После этого необходимо проверить результат, возвращаемый функцией `parseFloat()`, поскольку в случае невозможности извлечь число из текста, она возвращает значение `NaN` (**not a number** – нечисло), что может вызвать ошибку в методе `.sort()`.

Для ячеек, в которых хранятся даты, можно задействовать JavaScript объект `Date`:

```
row.sortKey = Date.parse('1 ' + $cell.text());
```

Даты в таблице содержат только месяц и год, а метод `Date.parse()` требует, чтобы дата была указана полностью, поэтому мы добавляем в начало строки `1`. Таким образом мы дополняем месяц и год числом месяца, благодаря чему полученная комбинация может быть преобразована в значение типа **timestamp**, которое может использоваться для сортировки нашей обычной функцией сравнения.

Мы можем оформить эти выражения в виде отдельных функций и вызывать подходящую, опираясь на класс в заголовке таблицы:

```
jQuery.fn.alternateRowColors = function() {
  $('tbody tr:odd', this)
    .removeClass('even').addClass('odd');
  $('tbody tr:even', this)
    .removeClass('odd').addClass('even');
  return this;
};

$(document).ready(function() {
  $('table.sortable').each(function() {
    var $table = $(this);
    $table.alternateRowColors();
    $('th', $table).each(function(column) {
      var $header = $(this);
      var findSortKey;
```

```
      if ($header.is('.sort-alpha')) {
        findSortKey = function($cell) {
          return $cell.find('.sort-key')
            .text().toUpperCase()
              + ' ' + $cell.text().toUpperCase();
        };
      }
      else if ($header.is('.sort-numeric')) {
        findSortKey = function($cell) {
          var key = $cell.text().replace(/^[^\d.]*/, '');
          key = parseFloat(key);
          return isNaN(key) ? 0 : key;
        };
      }
      else if ($header.is('.sort-date')) {
        findSortKey = function($cell) {
          return Date.parse('1 ' + $cell.text());
        };
      }
      if (findSortKey) {
        $header.addClass('clickable').hover(function() {
          $header.addClass('hover');
        }, function() {
          $header.removeClass('hover');
        }).click(function() {
          var rows = $table.find('tbody > tr').get();
          $.each(rows, function(index, row) {
            var $cell = $(row).children('td').eq(column);
            row.sortKey = findSortKey($cell);
          });
          rows.sort(function(a, b) {
            if (a.sortKey < b.sortKey) return -1;
            if (a.sortKey > b.sortKey) return 1;
            return 0;
          });
          $.each(rows, function(index, row) {
            $table.children('tbody').append(row);
            row.sortKey = null;
          });
          $table.alternateRowColors();
        });
      }
    });
  });
});
```

Переменная `findSortKey` дополнительно работает как функция для вычисления значений ключей, а флаг помогает определить, каким классом сортировки отмечен заголовок столбца. Теперь мы можем сортировать данные в таблице по дате или по цене, как показано на рис. 7.6 и 7.7.

	Title	Author(s)	Publish Date	Price
	Building Websites with the ASP.NET Community Starter Kit	Cristian Darie, K. Scott Allen	May 2004	$40.49
	Building Websites with Plone	Cameron Cooper	Nov 2004	$44.99
	Windows Server 2003 Active Directory Design and Implementation: Creating, Migrating, and Merging Networks	John Savill	Jan 2005	$53.99
	SSL VPN: Understanding, evaluating and planning secure, web-based remote access	Tim Speed, Joseph Steinberg	Mar 2005	$44.99

Рис. 7.6. *Таблица, отсортированная по датам публикаций книг*

	Title	Author(s)	Publish Date	Price
	User Training for Busy Programmers	William Rice	Jun 2005	$11.69
	Creating your MySQL Database: Practical Design Tips and Techniques	Marc Delisle	Nov 2006	$17.99
	Pluggable Authentication Modules: The Definitive Guide to PAM for Linux SysAdmins and C Developers	Kenneth Geisshirt	Jan 2007	$17.99
	Invision Power Board 2: A User Guide	David Mytton	Jul 2005	$22.49

Рис. 7.7. *Таблица, отсортированная по ценам книг*

Выделение столбца

Полезным расширением пользовательского интерфейса может быть возможность напоминания пользователю о последней выполненной операции. Выделение столбца, по которому была выполнена сортировка, может помочь пользователю сосредоточить внимание на той части таблицы, которая наиболее вероятно имеет отношение к делу. Благодаря тому что мы уже определили, как выделять ячейки в столбцах таблицы, все сводится к применению к ним соответствующего класса CSS:

```
$table.find('td').removeClass('sorted')
  .filter(':nth-child(' + (column + 1) + ')')
  .addClass('sorted');
```

Данный фрагмент сначала удаляет класс `sorted` из всех ячеек, а затем добавляет его только к ячейкам столбца, по которому только что была выполнена сортировка. Обратите внимание, что индекс столбца, определенный ранее, мы увеличили на 1; это обусловлено тем, что селектор `:nth-child()` начинает отсчет элементов не с нуля, а с единицы. Добавив этот программный код, мы обеспечим выделение столбца после любой операции сортировки, как показано на рис. 7.8.

	Title	Author(s)	Publish Date	Price
	Programming Windows Workflow Foundation: Practical WF Techniques and Examples using XAML and C#	K. Scott Allen	Dec 2006	$40.49
	Building Websites with XOOPS : A step-by-step tutorial	Steve Atwal	Oct 2006	$26.99
	Learn OpenOffice.org Spreadsheet Macro Programming: OOoBasic and Calc automation	Dr. Mark Alexander Bain	Dec 2006	$35.99
	UML 2.0 in Action: A project-based tutorial	Philippe Baumann, Henriette Baumann, Patrick Grassle	Sep 2005	$31.49

***Рис. 7.8.** Выделение столбца после любой операции сортировки*

Изменение направления сортировки

Последнее наше улучшение позволит выполнять сортировку в порядке **возрастания** или **убывания**. Пусть теперь необходимо, чтобы в результате выполнения пользователем щелчка на столбце, по которому уже была выполнена сортировка, текущий порядок сортировки менялся на обратный.

Чтобы обратить порядок сортировки, достаточно просто изменить знак значения, возвращаемого функцией сравнения. Это легко реализовать с помощью простой переменной:

```
if (a.sortKey < b.sortKey) return -sortDirection;
if (a.sortKey > b.sortKey) return sortDirection;
```

Если значение переменной `sortDirection` равно 1, тогда сортировка будет выполняться в том же порядке, что и прежде. Если оно равно -1, сортировка будет выполняться в обратном порядке. Для сохранения информации о текущем порядке сортировки можно использовать классы:

```
jQuery.fn.alternateRowColors = function() {
  $('tbody tr:odd', this)
    .removeClass('even').addClass('odd');
  $('tbody tr:even', this)
    .removeClass('odd').addClass('even');
  return this;
};

$(document).ready(function() {
  $('table.sortable').each(function() {
    var $table = $(this);
    $table.alternateRowColors();
    $('th', $table).each(function(column) {
      var $header = $(this);
      var findSortKey;
      if ($header.is('.sort-alpha')) {
        findSortKey = function($cell) {
          return $cell.find('.sort-key').text().toUpperCase()
            + ' ' + $cell.text().toUpperCase();
        };
      }
      else if ($header.is('.sort-numeric')) {
        findSortKey = function($cell) {
          var key = $cell.text().replace(/^[^\d.]*/, '');
          key = parseFloat(key);
          return isNaN(key) ? 0 : key;
        };
      }
      else if ($header.is('.sort-date')) {
        findSortKey = function($cell) {
          return Date.parse('1 ' + $cell.text());
        };
      }
      if (findSortKey) {
        $header.addClass('clickable').hover(function() {
          $header.addClass('hover');
        }, function() {
          $header.removeClass('hover');
        }).click(function() {
          var sortDirection = 1;
          if ($header.is('.sorted-asc')) {
            sortDirection = -1;
          }
          var rows = $table.find('tbody > tr').get();
          $.each(rows, function(index, row) {
            var $cell = $(row).children('td').eq(column);
            row.sortKey = findSortKey($cell);
          });
          rows.sort(function(a, b) {
            if (a.sortKey < b.sortKey) return -sortDirection;
            if (a.sortKey > b.sortKey) return sortDirection;
```

```
            return 0;
          });
          $.each(rows, function(index, row) {
            $table.children('tbody').append(row);
            row.sortKey = null;
          });
          $table.find('th').removeClass('sorted-asc')
            .removeClass('sorted-desc');
          if (sortDirection == 1) {
            $header.addClass('sorted-asc');
          }
          else {
            $header.addClass('sorted-desc');
          }
          $table.find('td').removeClass('sorted')
            .filter(':nth-child(' + (column + 1) + ')')
            .addClass('sorted');
          $table.alternateRowColors();
        });
      }
    });
  });
});
```

Благодаря тому, что порядок сортировки запоминается с помощью классов, мы получаем также дополнительную возможность изменять оформление заголовков столбцов, чтобы показать текущий порядок сортировки (рис. 7.9).

	Title	Author(s)	Publish Date	Price
	Microsoft AJAX C# Essentials: Building Responsive ASP.NET 2.0 Applications	Cristian Darie, Bogdan Brinzarea	Mar 2007	$31.99
	MediaWiki Administrators' Tutorial Guide	Mizanur Rahman	Mar 2007	$31.99
	CherryPy Essentials: Rapid Python Web Application Development	Sylvain Hellegouarch	Mar 2007	$31.99
	Visual SourceSafe 2005 Software Configuration Management in Practice	Alexandru Serban	Feb 2007	$44.99

Рис. 7.9. *Заголовки столбцов указывают на порядок сортировки*

Разбивка на страницы на стороне сервера

Сортировка – отличный способ облегчить поиск данных в большом объеме информации. Мы можем также помочь пользователю сосредоточить внимание на небольшой части данных, организовав **разбивку на страницы**.

Как и сортировка, разбивка на страницы часто выполняется на стороне сервера. Если данные для отображения хранятся в базе данных, можно легко организовать выемку данных по частям, используя предложение `LIMIT` в MySQL, `ROWNUM` – в Oracle или эквивалентные им способы в других механизмах баз данных.

Как и в первом примере сортировки, разбивка на страницы может быть инициирована посредством передачи на сервер информации в строке запроса, скажем, `index.php?page=52`. И снова, как и прежде, эту задачу можно выполнить, либо полностью загружая страницу, либо вставляя блоки данных в таблицу с помощью методов поддержки AJAX. Данная стратегия обеспечивает независимость от броузера и позволяет обрабатывать очень большие объемы данных.

Совместное использование сортировки и разбивки на страницы

Если данных достаточно много, чтобы сортировка приносила определенные выгоды, то, скорее всего, их будет достаточно много, чтобы реализовать функцию разбивки на страницы. Нет ничего необычного в желании объединить эти две функции, используемые в представлении данных. Но так как они обе оказывают влияние на набор данных, представленных на странице, важно рассмотреть их взаимодействие в процессе реализации.

И сортировка, и разбивка на страницы могут быть реализованы как на стороне сервера, так и в веб-броузере. Однако мы должны постоянно учитывать взаимовлияние этих двух функций, в противном случае можно получить интерфейс с обескураживающим поведением. Предположим, например, что у нас имеется таблица с восьмью строками и двумя столбцами, первоначально отсортированная по первому столбцу. В результате сортировки по второму столбцу многие строки могут поменять свое место в таблице, как показано на рис. 7.10.

Посмотрим, что произойдет, когда к таблице будет добавлена разбивка на страницы. Предположим, что сервер возвращает только первые четыре строки и броузер пытается отсортировать данные. Если разбивка на страницы выполняется на стороне сервера, а сортировка – в броузере, процедуре сортировки будет доступен не весь объем данных, что приведет к получению ошибочных результатов, как показано на рис. 7.11.

Из JavaScript можно манипулировать только теми данными, которые представлены в странице. Во избежание проблем необходимо реализовать обе функции либо на стороне сервера (запрашивая у сервера кор-

ректный набор данных при выполнении каждой операции выбора страницы или сортировки), либо в броузере (когда все необходимые данные будут постоянно доступны для JavaScript) и добиться таким образом того, чтобы первыми отображались действительно первые строки в наборе данных, как показано на рис. 7.12.

A	4
B	5
C	2
D	7
E	1
F	8
G	3
H	6

До

E	1
C	2
G	3
A	4
B	5
H	6
D	7
F	8

После

Рис. 7.10. *В результате сортировки некоторые строки могут изменить свое положение в таблице*

A	4
B	5
C	2
D	7

До

C	2
A	4
B	5
D	7

После

Рис. 7.11. *Получение ошибочных результатов, когда разбивка на страницы выполняется без учета сортировки*

A	4
B	5
C	2
D	7

До

E	1
C	2
G	3
A	4

После

Рис. 7.12. *После устранения проблемы, обусловленной взаимным влиянием функций сортировки и разбивки на страницы*

Разбивка на страницы с помощью JavaScript

Рассмотрим, как с помощью JavaScript можно реализовать разбивку на страницы для таблицы, для которой в броузере уже реализована воз-

можность сортировки. Сначала сосредоточимся на отображении определенной страницы с данными, игнорируя пока возможность взаимодействия с пользователем.

```
$(document).ready(function() {
  $('table.paginated').each(function() {
    var currentPage = 0;
    var numPerPage = 10;
    var $table = $(this);
    $table.find('tbody tr').hide()
      .slice(currentPage * numPerPage,
        (currentPage + 1) * numPerPage)
      .show();
  });
});
```

Этот программный код отображает первую страницу – десять строк данных.

Здесь мы снова опираемся на наличие элемента `<tbody>`, отделяющего данные от заголовков, – мы не хотим, чтобы исчезали заголовки или нижние колонтитулы при переходе на вторую страницу. В процессе выбора строк, содержащих данные, при переходе на новую страницу мы сначала скрываем все строки, затем выбираем строки для текущей страницы и после этого отображаем выбранные строки. Метод `.slice()`, используемый здесь, действует как метод массивов с тем же именем; он ограничивает число отбираемых элементов согласно заданным граничным позициям.

Большая часть ошибок в решении рассматриваемой задачи допускается при написании программного кода, формулирующего выражения для использования в вызове фильтра `.slice()`. Нам требуется найти порядковые номера строк в начале и в конце текущей страницы. Номер первой строки получается простым умножением номера текущей страницы на количество строк в каждой странице. Умножение числа строк на число, на единицу большее номера текущей страницы, дает порядковый номер первой строки следующей страницы; метод `.slice()` извлекает строки с порядковыми номерами до значения второго аргумента, но не включая его.

Отображение элемента выбора страницы

Чтобы добавить возможность взаимодействия с пользователем, мы должны поместить рядом с таблицей **элемент выбора страниц**: множество ссылок для навигации по страницам с данными. Для этого мы могли бы просто вставить ссылки на страницы прямо в разметку HTML, но это привело бы к нарушению принципа **прогрессивного улучшения**, которого мы стараемся придерживаться. Поэтому мы добавим ссылки с помощью JavaScript, чтобы неработающие ссылки не вводили в заблуждение пользователей, у которых отсутствует поддержка JavaScript.

Чтобы отобразить ссылки, необходимо определить число страниц и создать соответствующее число элементов DOM:

```
var numRows = $table.find('tbody tr').length;
var numPages = Math.ceil(numRows / numPerPage);

var $pager = $('<div class="pager"></div>');
for (var page = 0; page < numPages; page++) {
  $('<span class="page-number">' + (page + 1) + '</span>')
    .appendTo($pager).addClass('clickable');
}
$pager.insertBefore($table);
```

Число страниц можно определить, разделив число строк данных на число элементов, отображаемых в одной странице. Так как при делении может получиться нецелое число, мы должны округлить результат вверх до ближайшего целого числа с помощью `Math.ceil()`, чтобы гарантировать доступность последней неполной страницы. Получив требуемое число, можно создать кнопки, по одной для каждой страницы, и поместить новый элемент выбора страниц над таблицей, как показано на рис. 7.13.

***Рис. 7.13.** Элемент выбора страниц над таблицей*

Обработка кнопок выбора страниц

Данные кнопки должны вызывать изменение переменной `currentPage` и запускать процедуру перелистывания страниц. На первый взгляд кажется, что мы можем реализовать это, записывая в переменную `currentPage` значение `page`, равное текущему номеру итерации, в ходе которой создается очередная кнопка:

```
$(document).ready(function() {
  $('table.paginated').each(function() {
    var currentPage = 0;
    var numPerPage = 10;
    var $table = $(this);
    var repaginate = function() {
      $table.find('tbody tr').hide()
        .slice(currentPage * numPerPage,
           (currentPage + 1) * numPerPage)
        .show();
    };
    var numRows = $table.find('tbody tr').length;
    var numPages = Math.ceil(numRows / numPerPage);
```

```
    var $pager = $('<div class="pager"></div>');
    for (var page = 0; page < numPages; page++) {
      $('<span class="page-number"></span>').text(page + 1)
        .click(function() {
          currentPage = page;
          repaginate();
        }).appendTo($pager).addClass('clickable');
    }
    $pager.insertBefore($table);
  });
});
```

Эта реализация работает в том смысле, что она вызывает новую функцию `repaginate()`, когда производится загрузка веб-страницы и когда выполняется щелчок на любой из ссылок. Однако в результате щелчка на любой из ссылок мы будем получать таблицу, в которой отсутствуют строки с данными, как показано на рис. 7.14.

Рис. 7.14. *После щелчка на любой из ссылок отображается пустая таблица*

Проблема заключается в том, что при определении обработчика `click` создается **замыкание**. Обработчик события `click` ссылается на переменную `page`, которая определена *за пределами* функции. Когда на следующей итерации происходит изменение значения переменной, это оказывает воздействие и на обработчики события `click`, определенные ранее. В результате для элемента выбора из 7 страниц каждая кнопка будет ссылаться на страницу с номером 8 (последнее значение переменной `page`, получаемое по завершении цикла).

Примечание

Дополнительную информацию о том, как работают замыкания, можно найти в приложении C.

Для устранения этой проблемы мы воспользуемся одной из расширенных особенностей методов обработки событий в библиотеке jQuery. Во время подключения обработчика события мы можем добавить набор **собственных данных, связанных с событием**, которые будут доступны обработчику в момент вызова. Благодаря наличию этой возможности в нашем портфеле мы можем записать такой код:

```
$('<span class="page-number"></span>').text(page + 1)
  .bind('click', {newPage: page}, function(event) {
    currentPage = event.data['newPage'];
    repaginate();
  }).appendTo($pager).addClass('clickable');
```

Номер новой страницы передается обработчику через свойство `data` события. Вследствие этого номер страницы уходит от опасности попадания в замыкание, и его значение фиксируется в момент, когда выполняется подключение обработчика. Теперь ссылки в элементе выбора страниц работают правильно, вызывая появление разных страниц, как показано на рис. 7.15.

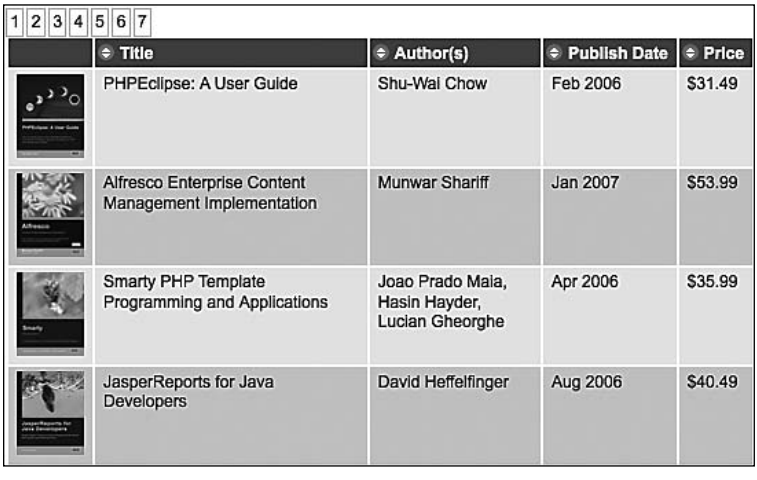


Рис. 7.15. *После устранения проблемы с замыканиями выбор страниц происходит, как ожидалось*

Выделение номера текущей страницы

Наш элемент выбора страниц можно сделать более дружественным по отношению к пользователю, если выделять в нем номер текущей страницы. Для этого достаточно изменять в кнопках классы при каждом щелчке:

```
$(document).ready(function() {
  $('table.paginated').each(function() {
    var currentPage = 0;
    var numPerPage = 10;
    var $table = $(this);
    var repaginate = function() {
      $table.find('tbody tr').hide()
        .slice(currentPage * numPerPage,
          (currentPage + 1) * numPerPage)
        .show();
    };
    var numRows = $table.find('tbody tr').length;
    var numPages = Math.ceil(numRows / numPerPage);
    var $pager = $('<div class="pager"></div>');
    for (var page = 0; page < numPages; page++) {
      $('<span class="page-number"></span>').text(page + 1)
```

```
        .bind('click', {newPage: page}, function(event) {
          currentPage = event.data['newPage'];
          repaginate();
          $(this).addClass('active')
            .siblings().removeClass('active');
        }).appendTo($pager).addClass('clickable');
    }
    $pager.insertBefore($table)
      .find('span.page-number:first').addClass('active');
  });
});
```

Теперь у нас имеется индикатор текущего состояния элемента выбора страниц, как показано на рис. 7.16.

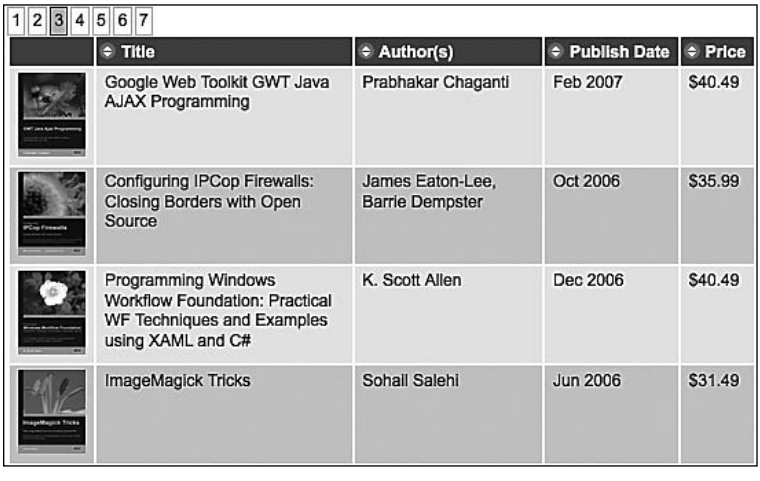


Рис. 7.16. *Элемент выбора страниц указывает, какая страница является текущей*

Разбивка на страницы с сортировкой

Мы начали данное обсуждение с замечания, что функции сортировки и разбивки на страницы должны взаимодействовать между собой во избежание обескураживающих результатов. Теперь, когда у нас имеется работающий элемент выбора страниц, необходимо, чтобы операция сортировки осуществлялась с учетом номера текущей страницы.

Для выполнения такой задачи достаточно вызвать функцию `repaginate()` после завершения операции сортировки. Однако **область видимости** функции делает это проблематичным. Мы не вправе вызвать функцию `repaginate()` из процедуры сортировки, потому что она определена внутри другого обработчика `$(document).ready()`. Можно было бы просто объединить эти два фрагмента программного кода, но давайте поступим немного хитрее. Мы можем **ослабить** связь между реализациями так, чтобы функция сортировки вызывала `repaginate`, только если она

существует, и игнорировала ее в противном случае. Для этого мы будем использовать обработчик **искусственного события**.

В предыдущих наших дискуссиях, касающихся обработки событий, мы ограничивались событиями, которые возбуждаются веб-броузером, такими как `click` и `mouseup`. Но методы `.bind()` и `.trigger()` не ограничиваются ими; в качестве имени события можно использовать любую строку. Благодаря данной возможности мы можем определить новое событие с именем `repaginate`, которое будет применяться взамен функции:

```
$table.bind('repaginate', function() {
  $table.find('tbody tr').hide()
    .slice(currentPage * numPerPage,
      (currentPage + 1) * numPerPage)
    .show();
});
```

Теперь мы можем заменить вызов функции `repaginate()` на:

```
$table.trigger('repaginate');
```

Точно такой же вызов можно выполнять и в процедуре сортировки. Он не будет оказывать никакого воздействия, если в таблице отсутствует элемент выбора страниц, поэтому мы можем объединять эти две возможности по своему желанию.

Окончательная версия

Ниже приводится окончательная версия реализации функций сортировки и разбивки на страницы:

```
jQuery.fn.alternateRowColors = function() {
  $('tbody tr:odd', this)
    .removeClass('even').addClass('odd');
  $('tbody tr:even', this)
    .removeClass('odd').addClass('even');
  return this;
};
$(document).ready(function() {
  $('table.sortable').each(function() {
    var $table = $(this);
    $table.alternateRowColors();
    $('th', $table).each(function(column) {
      var $header = $(this);
      var findSortKey;
      if ($header.is('.sort-alpha')) {
        findSortKey = function($cell) {
          return $cell.find('.sort-key').text().toUpperCase()
            + ' ' + $cell.text().toUpperCase();
        };
      }
      else if ($header.is('.sort-numeric')) {
        findSortKey = function($cell) {
```

```
        var key = $cell.text().replace(/^[^\d.]*/, '');
        key = parseFloat(key);
        return isNaN(key) ? 0 : key;
      };
    }
    else if ($header.is('.sort-date')) {
      findSortKey = function($cell) {
        return Date.parse('1 ' + $cell.text());
      };
    }
    if (findSortKey) {
      $header.addClass('clickable').hover(function() {
        $header.addClass('hover');
      }, function() {
        $header.removeClass('hover');
      }).click(function() {
        var sortDirection = 1;
        if ($header.is('.sorted-asc')) {
          sortDirection = -1;
        }
        var rows = $table.find('tbody > tr').get();
        $.each(rows, function(index, row) {
          var $cell = $(row).children('td').eq(column);
          row.sortKey = findSortKey($cell);
        });
        rows.sort(function(a, b) {
          if (a.sortKey < b.sortKey) return -sortDirection;
          if (a.sortKey > b.sortKey) return sortDirection;
          return 0;
        });
        $.each(rows, function(index, row) {
          $table.children('tbody').append(row);
          row.sortKey = null;
        });
        $table.find('th').removeClass('sorted-asc')
          .removeClass('sorted-desc');
        if (sortDirection == 1) {
          $header.addClass('sorted-asc');
        }
        else {
          $header.addClass('sorted-desc');
        }
        $table.find('td').removeClass('sorted')
          .filter(':nth-child(' + (column + 1) + ')')
          .addClass('sorted');
        $table.alternateRowColors();
        $table.trigger('repaginate');
      });
    }
  });
});
```

```
});
$(document).ready(function() {
  $('table.paginated').each(function() {
    var currentPage = 0;
    var numPerPage = 10;
    var $table = $(this);
    $table.bind('repaginate', function() {
      $table.find('tbody tr').hide()
        .slice(currentPage * numPerPage,
          (currentPage + 1) * numPerPage)
        .show();
    });
    var numRows = $table.find('tbody tr').length;
    var numPages = Math.ceil(numRows / numPerPage);
    var $pager = $('<div class="pager"></div>');
    for (var page = 0; page < numPages; page++) {
      $('<span class="page-number"></span>').text(page + 1)
        .bind('click', {newPage: page}, function(event) {
          currentPage = event.data['newPage'];
          $table.trigger('repaginate');
          $(this).addClass('active')
            .siblings().removeClass('active');
        }).appendTo($pager).addClass('clickable');
    }
    $pager.insertBefore($table)
      .find('span.page-number:first').addClass('active');
  });
});
```

Изменение внешнего вида таблицы

Мы рассмотрели некоторые способы упорядочения строк данных в таблице с целью помочь пользователю быстрее находить необходимую ему информацию. Однако в таблицах часто бывает множество данных, которые требуется отсеять после выполнения сортировки или разбивки на страницы. Мы можем содействовать пользователю в управлении не только порядком и количеством отображаемых строк, но и их внешним видом.

Выделение строк

Один из способов привлечения внимания пользователя к определенным строкам заключается в их **выделении**, с тем чтобы можно было визуально отличить наиболее важные данные. Для исследования некоторых стратегий выделения нам потребуется таблица, с которой мы будем работать. На этот раз мы будем использовать таблицу с заголовками новостей. Она будет немного сложнее предыдущей, поскольку будет содержать строки, играющие роль **подзаголовков**, дополняющих главную строку заголовка. Разметка HTML приводится ниже:

```
<table>
  <thead>
    <tr>
      <th>Date</th>
      <th>Headline</th>
      <th>Author</th>
      <th>Topic</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <th colspan="4">2008</th>
    </tr>
    <tr>
      <td>Sep 28</td>
      <td>jQuery, Microsoft, and Nokia</td>
      <td>John Resig</td>
      <td>third-party</td>
    </tr>
      ...
    <tr>
      <td>Jan 15</td>
      <td>jQuery 1.2.2: 2nd Birthday Present</td>
      <td>John Resig</td>
      <td>release</td>
    </tr>
  </tbody>
  <tbody>
    <tr>
      <th colspan="4">2007</th>
    </tr>
    <tr>
      <td>Dec 8</td>
      <td>jQuery Plugins site updated</td>
      <td>Mike Hostetler</td>
      <td>announcement</td>
    </tr>
      ...
    <tr>
      <td>Jan 11</td>
      <td>Selector Speeds</td>
      <td>John Resig</td>
      <td>source</td>
    </tr>
  </tbody>
    ...
</table>
```

Обратите внимание на наличие нескольких разделов `<tbody>`. Это вполне допустимая разметка HTML, применяемая для группировки наборов строк. В пределах таких групп мы создали разделы подзаголовков,

воспользовавшись элементами <th> для их выделения. После добавления простых правил CSS данная таблица будет выглядеть, как показано на рис. 7.17.

Date	Headline	Author	Topic
2008			
Sep 28	jQuery, Microsoft, and Nokia	John Resig	third-party
Aug 31	jQuery Conference 2008 Agenda	Rey Bango	conference
Aug 29	jQuery.com Site Redesign	John Resig	announcement
Aug 15	Registration Open for jQuery Conference 2008	Karl Swedberg	conference
Jul 14	jQuery UI 1.5.2	Paul Bakaus	release
Jun 26	jQuery UI 1.5.1	Paul Bakaus	release
Jun 26	jQuery Camp 2008 Announced	Rey Bango	conference
Jun 9	jQuery UI v1.5 Released, Focus on Consistent API and Effects	Paul Bakaus	release
Jun 4	jQuery 1.2.6: Events 100% faster	John Resig	release
Mar 7	jQuery UI Worldwide Sprint: March 14-15	Richard Worth	conference
Feb 8	jQuery 1.2.3: AIR, Namespacing, and UI Alpha	John Resig	release
Jan 23	jQuery UI and beyond: The jQuery-Liferay partnership	Paul Bakaus	announcement
Jan 15	jQuery 1.2.2: 2nd Birthday Present	John Resig	release
2007			
Dec 8	jQuery Plugins site updated	Mike Hostetler	announcement
Dec 6	Flot, a new plotting plugin for jQuery	Bradley Sepos	plug-in
Nov 2	Google Using jQuery	Rey Bango	third-party

Рис. 7.17. *Начальный вид таблицы с разделами*

Выделение чередующихся строк

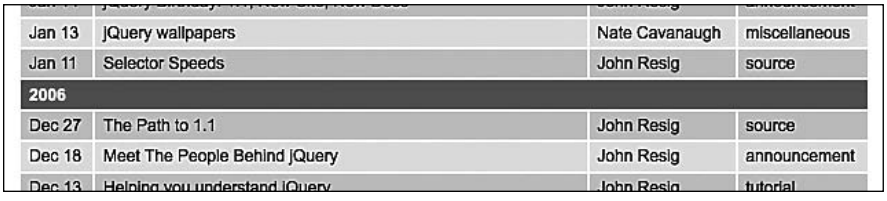
Мы уже встречали один простой пример выделения строк выше в этой главе и еще один – в главе 2. Оформление чередования строк – распространенный способ помочь пользователю зацепиться взглядом за строку при просмотре информации в множестве столбцов.

Как мы уже видели, выделение чередующихся строк реализуется всего парой строк программного кода, добавляющих классы к четным и нечетным строкам:

```
$(document).ready(function() {
  $('table.striped tr:odd).addClass('odd');
  $('table.striped tr:even).addClass('even');
});
```

Эта реализация прекрасно работает с таблицами, имеющими простую структуру. При введении дополнительных строк (таких как строки подзаголовков, используемые в нашей таблице для отображения года) мы не хотим, чтобы они участвовали в таком выделении; шаблона четный-нечетный в данном случае будет недостаточно, поскольку если бы, например, к строке **2006** был добавлен класс even, то к обеим строкам пе-

ред ней и после нее был бы добавлен класс `odd`, что, скорее всего, было бы нежелательно (рис. 7.18).

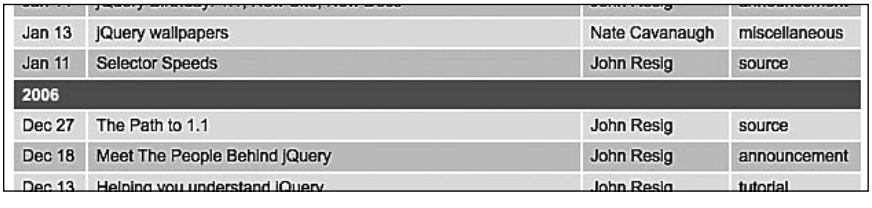


Jan 13	jQuery wallpapers	Nate Cavanaugh	miscellaneous
Jan 11	Selector Speeds	John Resig	source
2006			
Dec 27	The Path to 1.1	John Resig	source
Dec 18	Meet The People Behind jQuery	John Resig	announcement
Dec 13	Helping you understand jQuery	John Resig	tutorial

Рис. 7.18. *Функция чередования окраски строк не учитывает наличие строк заголовков*

Мы можем изменить наш шаблон чередования строк с заголовками новостей так, чтобы отсчет строк начинался заново с каждой строки с номером года (как показано на рис. 7.19), используя для этого псевдокласс `:nth-child()`, с которым мы познакомились в главе 2:

```
$(document).ready(function() {
  $('table.striped tr:nth-child(odd)').addClass('odd');
  $('table.striped tr:nth-child(even)').addClass('even');
});
```



Jan 13	jQuery wallpapers	Nate Cavanaugh	miscellaneous
Jan 11	Selector Speeds	John Resig	source
2006			
Dec 27	The Path to 1.1	John Resig	source
Dec 18	Meet The People Behind jQuery	John Resig	announcement
Dec 13	Helping you understand jQuery	John Resig	tutorial

Рис. 7.19. *Отсчет строк начинается заново с каждой строки с номером года*

Теперь каждая группа строк будет начинаться с нечетной строки, но при этом строки заголовков будут участвовать в подсчете. Чтобы такого не происходило, мы можем исключить строки подзаголовков из рассмотрения, использовав псевдокласс `:has()`; тогда мы получим таблицу, показанную на рис. 7.20:

```
$(document).ready(function() {
  $('table.striped tr:not(:has(th)):odd').addClass('odd');
  $('table.striped tr:not(:has(th)):even').addClass('even');
});
```

Теперь подзаголовки не участвуют в подсчете, однако группы начинаются с четной или нечетной строки в зависимости от того, какой класс применялся к предыдущей строке с данными. Совмещение этих двух реализаций может оказаться непростым делом; один из наиболее прямолинейных способов заключается в выполнении **явных итераций** с помощью метода `.each()`.

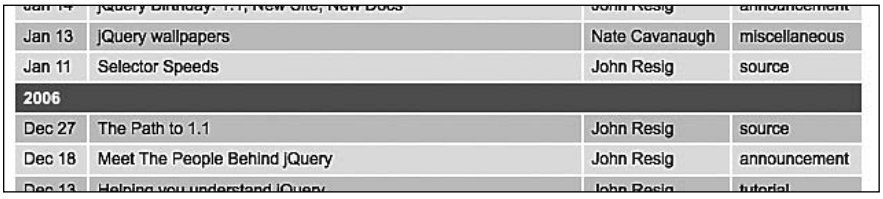


Рис. 7.20. *Отсчет строк начинается заново с первой строки каждого раздела*

```
$(document).ready(function() {
  $('table.striped tbody').each(function() {
    $(this).find('tr:not(:has(th)):odd').addClass('odd');
    $(this).find('tr:not(:has(th)):even').addClass('even');
  });
});
```

Теперь оформление чередования строк в каждой группе выполняется независимо, и строки подзаголовков исключаются из подсчетов; результат показан на рис. 7.21.

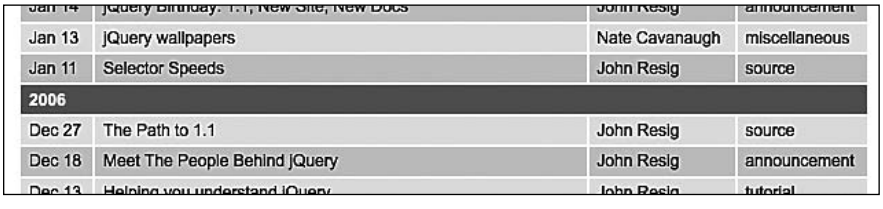


Рис. 7.21. *Независимое чередование окраски строк в каждом разделе*

Дополнительные способы оформления чередования строк

Эти манипуляции с четными и нечетными строками подготовили нас к реализации более сложных приемов. В особенно плотных таблицах даже чередование цвета строк может не помогать зацепиться взглядом за определенную строку, поэтому может оказаться предпочтительнее организовать чередование цвета на большем интервале строк. Например, в нашей таблице с заголовками новостей можно было бы организовать чередование, окрашивая одним цветом сразу по три строки.

В главе 2 был представлен метод `.filter()`, позволяющий очень гибко выбирать элементы страницы. Вспомнив, что метод `.filter()` может принимать не только выражение селектора, но и **функцию фильтрации**, мы можем написать:

```
$(document).ready(function() {
  $('table.striped tbody').each(function() {
    $(this).find('tr:not(:has(th))').filter(function(index) {
      return (index % 6) < 3;
    }).addClass('odd');
  });
});
```

Этот программный код добивается многого, несмотря на небольшой объем, поэтому рассмотрим его по частям.

Во-первых, мы, как и раньше, использовали метод `.each()`, чтобы отделить друг от друга группы строк. Нам необходимо, чтобы окрашивание чередующихся фрагментов, по три строки в каждом, начиналось заново после каждого подзаголовка, так что именно данный прием позволяет нам работать с одним сегментом таблицы за раз. Затем, как и в последнем примере, мы использовали метод `.find()`, чтобы выбрать все строки, не содержащие элементов `<th>` (то есть не являющиеся подзаголовками).

Теперь необходимо выбрать первые три элемента из полученного множества, пропустить следующие три элемента и так далее. Здесь в игру вступает метод `.filter()`. Функция фильтрации принимает аргумент, содержащий порядковый номер элемента внутри соответствующего набора, то есть номер строки в сегменте таблицы, с которой мы работаем. Элемент останется в наборе, если и только если наша функция фильтрации возвратит значение `true`.

Оператор **деления по модулю (%)** позволяет получить всю необходимую нам информацию. Выражение `index % 6` вычисляет остаток от деления номера строки на число 6; если остаток равен 0, 1 или 2, строка помечается классом `odd`, если остаток равен 3, 4 или 5, строка помечается классом `even`.

Программный код, как видно из реализации, отмечает только нечетные (`odd`) группы строк. Чтобы применить класс `even`, мы можем написать другой фильтр, который будет использовать противоположное условие, или подойти к реализации более творчески:

```
$(document).ready(function() {
  $('table.striped tbody').each(function() {
    $(this).find('tr:not(:has(th))').addClass('even')
      .filter(function(index) {
        return (index % 6) < 3;
      }).removeClass('even').addClass('odd');
  });
});
```

Здесь мы сначала применяем класс `even` ко всем строкам, а затем удаляем его из тех строк, к которым применяется класс `odd`. Теперь в нашей таблице цветом выделяются чередующиеся группы, по три строки в каждой, причем выделение чередующихся групп начинается заново в каждом новом сегменте таблицы, как показано на рис. 7.22.

Интерактивное выделение строк

Еще один визуальный эффект, который можно применить к нашей таблице с заголовками новостей, – это выделение строки по желанию пользователя. Здесь в ответ на щелчок мышью на имени автора мы будем выделять все строки, где в ячейке Author указано то же имя автора.

Date	Headline	Author	Topic
2008			
Sep 28	jQuery, Microsoft, and Nokia	John Resig	third-party
Aug 31	jQuery Conference 2008 Agenda	Rey Bango	conference
Aug 29	jQuery.com Site Redesign	John Resig	announcement
Aug 15	Registration Open for jQuery Conference 2008	Karl Swedberg	conference
Jul 14	jQuery UI 1.5.2	Paul Bakaus	release
Jun 26	jQuery UI 1.5.1	Paul Bakaus	release
Jun 26	jQuery Camp 2008 Announced	Rey Bango	conference
Jun 9	jQuery UI v1.5 Released, Focus on Consistent API and Effects	Paul Bakaus	release
Jun 4	jQuery 1.2.6: Events 100% faster	John Resig	release
Mar 7	jQuery UI Worldwide Sprint: March 14-15	Richard Worth	conference
Feb 8	jQuery 1.2.3: AIR, Namespacing, and UI Alpha	John Resig	release
Jan 23	jQuery UI and beyond: The jQuery-Liferay partnership	Paul Bakaus	announcement
Jan 15	jQuery 1.2.2: 2nd Birthday Present	John Resig	release
2007			
Dec 8	jQuery Plugins site updated	Mike Hostetler	announcement
Dec 6	Flot, a new plotting plugin for jQuery	Bradley Sepos	plug-in
Nov 2	Google Using jQuery	Rey Bango	third-party
Sep 17	jQuery UI: Interactions and Widgets	John Resig	announcement
Sep 10	jQuery 1.2: jQuery.extend("Awesome")	John Resig	release
Sep 6	jQueryCamp '07 (Boston)	John Resig	conference
Aug 24	jQuery 1.1.4: Faster, More Tests, Ready for 1.2	John Resig	release
Jul 17	SF jQuery Meetup and Ajax Experience	John Resig	conference

Рис. 7.22. *Чередование окраски групп строк*

Как и в случае выделения чередующихся строк, мы можем изменять внешний вид выделенных строк, добавляя к ним класс CSS:

```
#content tr.highlight {
  background: #ff6;
}
```

Очень важно дать точное **определение** класса `highlight`, чтобы он переназначал цвет фона классов `even` и `odd`.

Теперь нам необходимо выбрать соответствующую ячейку и подключить к ней обработчик события с помощью метода `.click()`:

```
$(document).ready(function() {
  var $authorCells = $('table.striped td:nth-child(3)');
  $authorCells.click(function() {
    // Здесь выполняется выделение строк.
  });
});
```

Обратите внимание, что в выражении селектора мы использовали псевдокласс `:nth-child(n)`, чтобы указать на третий столбец, в котором содержится информация об авторе. Эту константу `3` желательно было бы по-

местить где-то в одном месте в программном коде, чтобы ее проще было изменить, если вдруг позднее структура таблицы изменится. По этой причине, а также из соображений эффективности мы сохраняем результат выбора по селектору в переменной `$authorCells` вместо того, чтобы запускать селектор каждый раз, когда это будет необходимо.

Примечание

Не забывайте, что, в отличие от индексов JavaScript, псевдокласс `:nth-child(n)` начинает нумерацию не с 0, а с 1.

После того как пользователь щелкнет на ячейке в третьем столбце, нам необходимо сравнить текст в ячейке с текстом в ячейках того столбца во всех остальных строках. При совпадении текста необходимо переключить класс `highlight`. Другими словами, если класс в ячейке отсутствует, его необходимо добавить, а если он присутствует, – удалить. Благодаря такой реализации можно щелкнуть на ячейке с именем автора, чтобы убрать выделение строки, если одна или более строк с этим же автором уже была выделена щелчком мыши.

```
$(document).ready(function() {
  var $authorCells = $('table.striped td:nth-child(3)');
  $authorCells.click(function() {
    var authorName = $(this).text();
    $authorCells.each(function(index) {
      if (authorName == $(this).text()) {
        $(this).parent().toggleClass('highlight');
      }
    });
  });
});
```

Наша реализация прекрасно работает, за исключением случая, когда пользователь щелкает сначала на имени одного автора, а потом на имени другого. Вместо переключения выделения строк с одного автора на другого, как можно было бы ожидать, происходит применение класса `highlight` к обеим группам строк. Чтобы избежать этого, в программный код можно добавить инструкцию `else`, выполняющую удаление класса `highlight` во всех строках, в которых имя автора отличается от того, на котором был выполнен щелчок:

```
$(document).ready(function() {
  var $authorCells = $('table.striped td:nth-child(3)');
  $authorCells.click(function() {
    var authorName = $(this).text();
    $authorCells.each(function(index) {
      if (authorName == $(this).text()) {
        $(this).parent().toggleClass('highlight');
      }
      else {
```

```
        $(this).parent().removeClass('highlight');
      }
    });
  });
});
```

Теперь если мы выполним щелчок, например, на ячейке с именем **Rey Bango**, то сможем легко увидеть все новости, принадлежащие этому автору, как показано на рис. 7.23.

Date	Headline	Author	Topic
2008			
Sep 28	jQuery, Microsoft, and Nokia	John Resig	third-party
Aug 31	jQuery Conference 2008 Agenda	Rey Bango	conference
Aug 29	jQuery.com Site Redesign	John Resig	announcement
Aug 15	Registration Open for jQuery Conference 2008	Karl Swedberg	conference
Jul 14	jQuery UI 1.5.2	Paul Bakaus	release
Jun 26	jQuery UI 1.5.1	Paul Bakaus	release
Jun 26	jQuery Camp 2008 Announced	Rey Bango	conference
Jun 9	jQuery UI v1.5 Released, Focus on Consistent API and Effects	Paul Bakaus	release
Jun 4	jQuery 1.2.6: Events 100% faster	John Resig	release
Mar 7	jQuery UI Worldwide Sprint: March 14-15	Richard Worth	conference
Feb 8	jQuery 1.2.3: AIR, Namespacing, and UI Alpha	John Resig	release
Jan 23	jQuery UI and beyond: The jQuery-Liferay partnership	Paul Bakaus	announcement
Jan 15	jQuery 1.2.2: 2nd Birthday Present	John Resig	release
2007			
Dec 8	jQuery Plugins site updated	Mike Hostetler	announcement
Dec 6	Flot, a new plotting plugin for jQuery	Bradley Sepos	plug-in
Nov 2	Google Using jQuery	Rey Bango	third-party
Sep 17	jQuery UI: Interactions and Widgets	John Resig	announcement

Рис. 7.23. *Все новости, принадлежащие автору с именем Rey Bango*

Если затем в любой из строк щелкнуть на имени **John Resig**, выделение будет снято со строк с именем автора Rey Bango и добавлено к строкам с именем John Resig.

Подсказки

Несмотря на всю полезность возможности выделения строк, пользователь может даже не предполагать, что такая возможность существует. Мы можем исправить эту ситуацию, добавив класс `clickable` к ячейке с именем автора, который вызывает изменение формы указателя мыши при наведении его на ячейку:

```
$(document).ready(function() {
  var $authorCells = $('table.striped td:nth-child(3)');
  $authorCells
    .addClass('clickable')
    .click(function() {
```

```
        var authorName = $(this).text();
        $authorCells.each(function(index) {
          if (authorName == $(this).text()) {
            $(this).parent().toggleClass('highlight');
          }
          else {
            $(this).parent().removeClass('highlight');
          }
        });
      });
});
```

Вне всяких сомнений использование класса `clickable` – это шаг в правильном направлении, но пользователь по-прежнему не имеет представления, что произойдет, когда он щелкнет на ячейке. Опыт пользователя подсказывает ему, что щелчок легко может вызвать иную реакцию, например переход на другую страницу. Необходима какая-то дополнительная подсказка о том, что произойдет в случае щелчка.

Подсказки – знакомая особенность многих программ, включая и веб-броузеры. Мы можем обратить внимание пользователя на возможность выделения строк, отображая подсказку при наведении указателя мыши на ячейку в столбце Author. Текст подсказки может описывать, какой эффект будет иметь действие пользователя, например, с помощью сообщения Highlight all articles by Rey Bango (выделить все строки, автором которых является Rey Bango). Этот текст будет содержаться в элементе `<div>`, который можно добавить в элемент `<body>`. Переменная `$tooltip` будет использоваться в сценарии для обращения к вновь созданному элементу:

```
var $tooltip = $('<div id="tooltip"></div>').appendTo('body');
```

При отображении каждой подсказки нам потребуется выполнять три основных операции:

1. Отображать подсказку при наведении указателя мыши на интерактивный элемент,
2. Скрывать ее при выходе указателя мыши за пределы элемента,
3. Изменять местоположение подсказки при перемещении указателя мыши.

Для каждой из этих операций мы напишем функции, а затем подключим их к событиям броузера с помощью jQuery.

Начнем с функции `positionTooltip()`, которая будет вызываться при наведении указателя мыши на любую из ячеек в столбце Author:

```
var positionTooltip = function(event) {
  var tPosX = event.pageX;
  var tPosY = event.pageY + 20;
  $tooltip.css({top: tPosY, left: tPosX});
};
```

Здесь, чтобы определить координаты верхнего левого угла подсказки, мы использовали свойства pageX и pageY объекта event. Когда данная функция будет вызываться в ответ на события мыши, такие как mousemove, свойства event.pageX и event.pageY будут содержать координаты указателя мыши, в результате переменные tPosX и tPosY будут содержать координаты точки на экране, расположенной на 20 пикселов ниже указателя мыши.

Затем нам необходимо написать функцию showTooltip(), которая будет выводить подсказку на экран.

```
var showTooltip = function(event) {
  var authorName = $(this).text();
  $tooltip
    .text('Highlight all articles by ' + authorName)
    .show();
  positionTooltip(event);
};
```

Функция showTooltip() достаточно прямолинейна. В ней производятся заполнение подсказки содержимым посредством создания строки на основе содержимого ячейки (представляющего собой имя автора) и ее отображение.

Затем с помощью функции positionTooltip() подсказка помещается в нужное место на странице. Так как подсказка добавляется в элемент body, чтобы иметь возможность располагать ее в произвольном месте на странице, нам необходимо определить некоторые правила CSS для нее:

```
#tooltip {
  position: absolute;
  z-index: 2;
  background: #efd;
  border: 1px solid #ccc;
  padding: 3px;
}
```

Наконец, напишем простую функцию hideTooltip():

```
var hideTooltip = function() {
  $tooltip.hide();
};
```

Теперь, когда у нас имеются функции отображения, сокрытия и позиционирования подсказки, можно добавить обращения к ним в наш программный код:

```
$(document).ready(function() {
  var $authorCells = $('table.striped td:nth-child(3)');
  var $tooltip = $('<div id="tooltip"></div>').appendTo('body');

  var positionTooltip = function(event) {
    var tPosX = event.pageX;
```

```
    var tPosY = event.pageY + 20;
    $tooltip.css({top: tPosY, left: tPosX});
  };
  var showTooltip = function(event) {
    var authorName = $(this).text();
    $tooltip
      .text('Highlight all articles by ' + authorName)
      .show();
    positionTooltip(event);
  };
  var hideTooltip = function() {
    $tooltip.hide();
  };
  $authorCells
    .addClass('clickable')
    .hover(showTooltip, hideTooltip)
    .mousemove(positionTooltip)
    .click(function(event) {
      var authorName = $(this).text();
      $authorCells.each(function(index) {
        if (authorName == $(this).text()) {
          $(this).parent().toggleClass('highlight');
        }
        else {
          $(this).parent().removeClass('highlight');
        }
      });
    });
});
```

Обратите внимание, что в качестве аргументов методам `.hover()` и `.mousemove()` передаются **ссылки** на функции, определенные в другом месте, – мы опустили круглые скобки, которые следуют за именами функций при их вызове. Теперь при наведении указателя мыши на ячейку с именем автора отображается подсказка, которая перемещается вместе с перемещением указателя мыши и скрывается при выходе указателя за пределы ячейки, как показано на рис. 7.24.

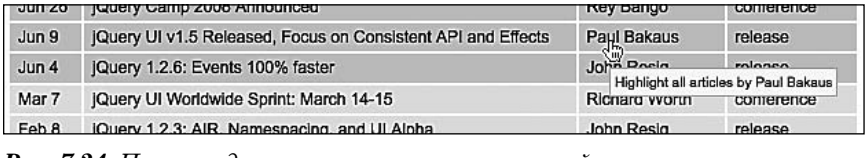


Рис. 7.24. *При наведении указателя мыши на ячейку с именем автора появляется подсказка*

Наша реализация имеет одну проблему, которая заключается в том, что подсказка продолжает предлагать щелкнуть на ячейке для выделения строк, даже когда текущая строка уже выделена, как показано на рис. 7.25.

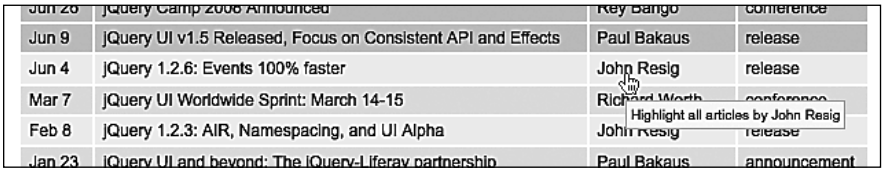


Рис. 7.25. *Подсказка по-прежнему предлагает выделить строки*

Нам необходимо предусмотреть изменение текста подсказки, когда строка под указателем мыши уже содержит класс `highlight`. Это можно реализовать, добавив проверку условия в функцию `showTooltip()`.Если родительский элемент `<tr>` текущей ячейки уже имеет класс `highlight`, необходимо при создании подсказки вместо слова Highlight (выделить) использовать слово Unhighlight (снять выделение):

```
var action = 'Highlight';
if ($(this).parent().is('.highlight')) {
  action = 'Unhighlight';
}
$tooltip
  .text(action + ' all articles by ' + authorName)
  .show();
```

Данная реализация корректно выбирает текст подсказки, когда указатель мыши наводится на ячейку, но нам необходимо повторно определить текст подсказки и в момент щелчка мышью. Для этого следует вызвать функцию `showTooltip()` внутри обработчика события `click`:

```
$(document).ready(function() {
  var $authorCells = $('table.striped td:nth-child(3)');
  var $tooltip = $('<div id="tooltip"></div>').appendTo('body');

  var positionTooltip = function(event) {
    var tPosX = event.pageX;
    var tPosY = event.pageY + 20;
    $tooltip.css({top: tPosY, left: tPosX});
  };
  var showTooltip = function(event) {
    var authorName = $(this).text();
    var action = 'Highlight';
    if ($(this).parent().is('.highlight')) {
      action = 'Unhighlight';
    }
    $tooltip
      .text(action + ' all articles by ' + authorName)
      .show();
    positionTooltip(event);
  };
  var hideTooltip = function() {
    $tooltip.hide();
  };
```

```
    $authorCells
      .addClass('clickable')
      .hover(showTooltip, hideTooltip)
      .mousemove(positionTooltip)
      .click(function(event) {
        var authorName = $(this).text();
        $authorCells.each(function(index) {
          if (authorName == $(this).text()) {
            $(this).parent().toggleClass('highlight');
          }
          else {
            $(this).parent().removeClass('highlight');
          }
        });
        showTooltip.call(this, event);
      });
});
```

Функция `call()` в языке JavaScript позволяет вызвать `showTooltip()` в области видимости обработчика события `click` ячейки с именем автора. Сделать это необходимо, чтобы во время выполнения функции `showTooltip()` ключевое слово `this` ссылалось на требуемый объект (данную ячейку таблицы).

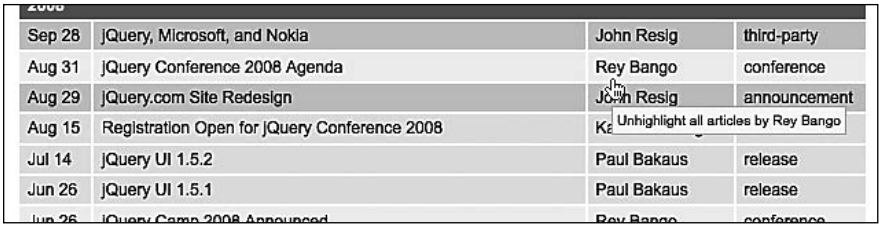


Рис. 7.26. *Подсказка содержит более уместное предложение*

Теперь при наведении указателя мыши на уже выделенную строку подсказка содержит более уместное предложение, как показано на рис. 7.26.

Свертывание и развертывание разделов таблицы

Когда большой объем данных делится на разделы, может быть полезно организовать сокрытие информации, которая в данный момент не представляет интереса для пользователя. В нашей таблице с заголовками новостей строки группируются по годам; **свертывание**, или сокрытие, новостей за тот или иной год обеспечит удобный способ получить общее представление обо всех данных в таблице без необходимости слишком долго перемещаться вверх и вниз по таблице.

Чтобы сделать разделы таблицы свертываемыми, сначала необходимо создать элемент страницы, который будет использоваться для до-

ступа к этой возможности. Обычно в качестве интерфейсного элемента, обозначающего разделы, допускающие сворачивание, используется знак минус, а для обозначения разделов, допускающих возможность разворачивания, используется знак плюс. Следуя стандартному приему **прогрессивного улучшения**, мы будем вставлять ярлык с помощью JavaScript.

```
$(document).ready(function() {
  var collapseIcon = '../images/bullet_toggle_minus.png';
  var collapseText = 'Collapse this section';
  var expandIcon = '../images/bullet_toggle_plus.png';
  var expandText = 'Expand this section';
  $('table.collapsible tbody').each(function() {
    var $section = $(this);
    $('<img />').attr('src', collapseIcon)
      .attr('alt', collapseText)
      .prependTo($section.find('th'));
  });
});
```

В начале функции мы определили местоположение ярлыков, а также их альтернативное текстовое представление в виде переменных. Это упростит возможность обращения к ним и изменения в случае необходимости. Вставку изображений мы реализовали в виде цикла `.each()`, что очень удобно, так как позднее нам необходимо будет иметь возможность ссылаться на вмещающий элемент `<tbody>`; он будет доступен через переменную `$section`, которая определяется здесь же.

Затем необходимо добавить к ярлыкам обработчики, которые будут обеспечивать сворачивание и разворачивание строк. Добавление класса `clickable` обеспечит необходимую обратную связь с пользователем, а класс `collapsed` в элементе `<tbody>` поможет определить, видимы ли строки текущего раздела в настоящий момент или нет.

```
$(document).ready(function() {
  var collapseIcon = '../images/bullet_toggle_minus.png';
  var collapseText = 'Collapse this section';
  var expandIcon = '../images/bullet_toggle_plus.png';
  var expandText = 'Expand this section';
  $('table.collapsible tbody').each(function() {
    var $section = $(this);
    $('<img />').attr('src', collapseIcon)
      .attr('alt', collapseText)
      .prependTo($section.find('th'))
      .addClass('clickable')
      .click(function() {
        if ($section.is('.collapsed')) {
          $section.removeClass('collapsed')
            .find('tr:not(:has(th))').fadeIn('fast');
          $(this).attr('src', collapseIcon)
            .attr('alt', collapseText);
```

```
        }
        else {
          $section.addClass('collapsed')
            .find('tr:not(:has(th))').fadeOut('fast');
          $(this).attr('src', expandIcon)
            .attr('alt', expandText);
        }
      });
  });
});
```

В случае выполнения щелчка мышью происходит следующее:

1. Добавляется или удаляется класс `collapsed` в элементе `<tbody>`, чтобы запомнить текущее состояние раздела таблицы.
2. Обнаруживаются все строки раздела, не содержащие заголовков, которые затем отображаются или скрываются с использованием эффекта плавного перехода от одного состояния к другому.
3. Переключается текущее состояние ярлыка за счет изменения значений атрибутов `src` и `alt`, чтобы отразить тип действия, которое будет выполняться при очередном щелчке.

Теперь после щелчка на ярлыке с подсказкой Collapse this section (свернуть этот раздел) в строке с текстом 2007 таблица будет выглядеть, как показано на рис. 7.27.

Jan 23	jQuery UI and beyond: The jQuery-Liferay partnership	Paul Bakaus	announcement
Jan 15	jQuery 1.2.2: 2nd Birthday Present	John Resig	release
⊞ 2007			
⊟ 2006			
Dec 27	The Path to 1.1	John Resig	source
Dec 18	Meet The People Behind jQuery	John Resig	announcement
Dec 13	Helping you understand jQuery	John Resig	tutorial

Рис. 7.27. *Вид таблицы со свернутым разделом новостей за 2007 год*

Заголовки новостей за 2007 год никуда не исчезают; они просто становятся невидимыми и остаются таковыми, пока мы не щелкнем на ярлыке с подсказкой Expand this section (развернуть этот раздел), появляющейся теперь в данной строке.

Примечание

Применение анимационных эффектов к строкам таблицы представляет собой определенную проблему, так как в броузерах используются различные значения (`table-row` и `block`) для свойства `display`, управляющего их видимостью. Методы `.hide()` и `.show()` без анимационных эффектов всегда можно безопасно использовать для управления видимостью строк таблицы. Если анимационный эффект является желательным, можно также применять методы `.fadeIn()` и `.fadeOut()`.

Фильтрация

Выше мы рассматривали сортировку и разбивку на страницы как приемы, облегчающие возможность сфокусировать внимание пользователя на соответствующих фрагментах данных в таблице. Мы видели, что обе эти возможности могут быть реализованы как с использованием серверных технологий, так и средствами JavaScript. **Фильтрация** дополняет этот арсенал стратегий управления данными. Отображая только те строки таблицы, которые удовлетворяют определенному критерию, мы можем устранить факторы, отвлекающие внимание.

Мы уже видели, как выполняется одна из разновидностей фильтрации: **выделение** набора строк. Теперь мы расширим эту идею фактическим **сокрытием** строк, не соответствующих критерию фильтра.

Начнем с создания места, куда будут помещаться ссылки фильтра. Используя типичную стратегию **прогрессивного улучшения**, мы будем вставлять необходимые элементы управления с помощью JavaScript, чтобы они не отображались в броузерах с отключенной поддержкой JavaScript:

```
$(document).ready(function() {
  $('table.filterable').each(function() {
    var $table = $(this);
    $table.find('th').each(function(column) {
      if ($(this).is('.filter-column')) {
        var $filters = $('<div class="filters"></div>');
        $('<h3></h3>')
          .text('Filter by ' + $(this).text() + ':')
          .appendTo($filters);
        $filters.insertBefore($table);
      }
    });
  });
});
```

Мы создали метку для поля фильтра, исходя из заголовков столбцов, благодаря чему этот программный код может легко использоваться и для других таблиц. Теперь у нас имеется заголовок, куда можно поместить некоторые кнопки, как показано на рис. 7.28.

Date	Headline	Author	Topic
2008			
Sep 28	jQuery, Microsoft, and Nokia	John Resig	third-party
Aug 31	jQuery Conference 2008 Agenda	Rey Bango	conference
Aug 29	jQuery.com Site Redesign	John Resig	announcement
Aug 15	Registration Open for jQuery Conference 2008	Karl Swedberg	conference
Jul 14	jQuery UI 1.5.2	Paul Bakaus	release

Filter by Topic:

Рис. 7.28. *Выделение раздела для дополнительных ссылок*

Параметры фильтра

Теперь можно перейти к фактической реализации фильтра. Для начала добавим фильтры для пары известных тем. Код для этого действия напоминает предыдущий пример выделения строк по автору:

```
$(document).ready(function() {
  $('table.filterable').each(function() {
    var $table = $(this);

    $table.find('th').each(function(column) {
      if ($(this).is('.filter-column')) {
        var $filters = $('<div class="filters"></div>');
        $('<h3></h3>')
          .text('Filter by ' + $(this).text() + ':')
          .appendTo($filters);

        var keywords = ['conference', 'release'];
        $.each(keywords, function(index, keyword) {
          $('<div class="filter"></div>').text(keyword)
            .bind('click', {key: keyword}, function(event) {
              $('tr:not(:has(th))', $table).each(function() {
                var value = $('td', this).eq(column).text();
                if (value == event.data['key']) {
                  $(this).show();
                }
                else {
                  $(this).hide();
                }
              });
              $(this).addClass('active')
                .siblings().removeClass('active');
            }).addClass('clickable').appendTo($filters);
        });
        $filters.insertBefore($table);
      }
    });
  });
});
```

Реализация начинается с определения статического массива ключевых слов, по которым будет выполняться фильтрация, затем выполняется обход массива в цикле и для каждого его элемента создается ссылка. Как и в примере с разбивкой на страницы, нам необходимо использовать аргумент `data` в вызове метода `.bind()`, чтобы избежать проблем, связанных с особенностями **замыканий**. После этого внутри обработчика события `click` содержимое каждой ячейки сравнивается с ключевым словом, и в случае несовпадения строка скрывается. Так как селектор строк исключает из набора строки, содержащие элемент `<th>`, можно не беспокоиться о том, что строки подзаголовков окажутся скрытыми.

Обе ссылки работают именно так, как и предполагалось (рис. 7.29).

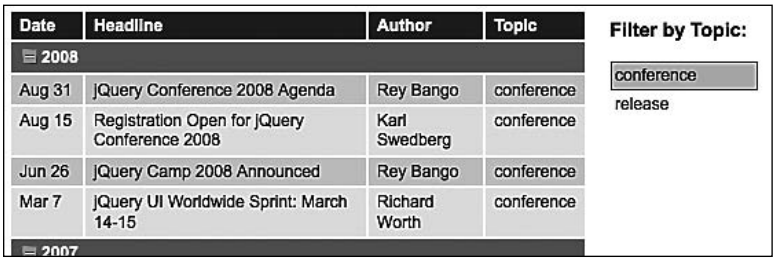


Рис. 7.29. *Вид таблицы после выполнения операции фильтрации*

Выборка параметров фильтра из содержимого

Теперь необходимо расширить набор параметров фильтра, чтобы охватить весь диапазон имеющихся тем. Мы можем отобрать их из текста, присутствующего в таблице. Для этого изменим определение массива `keywords` так, чтобы его элементы извлекались из таблицы:

```
var keywords = {};
$table.find('td:nth-child(' + (column + 1) + ')')
  .each(function() {
    keywords[$(this).text()] = $(this).text();
  });
```

В данной реализации используются следующие две особенности:

1. Используя для хранения найденных ключевых слов **отображение** вместо простого **массива**, мы автоматически устраняем возможность включения в массив дубликатов слов; каждый ключ может иметь только одно значение, и ключи всегда являются уникальными.
2. Функция `$.each()` из библиотеки jQuery позволяет работать с массивами и с отображениями одинаковым образом, благодаря чему нам не приходится изменять последующий программный код.

Теперь у нас имеется полный набор параметров фильтра (рис. 7.30).

Рис. 7.30. *На странице выводится полный набор параметров фильтра*

Изменение действия фильтра на противоположное

Для полноты реализации нам необходимо предусмотреть способ вернуться к полному списку после фильтрации. Добавить параметр, обозначающий все темы, очень просто:

```
$('<div class="filter">all</div>').click(function() {
  $table.find('tbody tr').show();
  $(this).addClass('active')
    .siblings().removeClass('active');
}).addClass('clickable active').appendTo($filters);
```

Данный фрагмент создает ссылку all (все), щелчок на которой приводит к отображению всех строк таблицы. Для большей наглядности эта новая ссылка с самого начала отмечена как активная, как показано на рис. 7.31.

Date	Headline	Author	Topic
⊟ 2008			
Sep 28	jQuery, Microsoft, and Nokia	John Resig	third-party
Aug 31	jQuery Conference 2008 Agenda	Rey Bango	conference
Aug 29	jQuery.com Site Redesign	John Resig	announcement
Aug 15	Registration Open for jQuery Conference 2008	Karl Swedberg	conference
Jul 14	jQuery UI 1.5.2	Paul Bakaus	release
Jun 26	jQuery UI 1.5.1	Paul Bakaus	release
Jun 26	jQuery Camp 2008 Announced	Rey Bango	conference
Jun 9	jQuery UI v1.5 Released, Focus on Consistent API and Effects	Paul Bakaus	release
Jun 4	jQuery 1.2.6: Events 100% faster	John Resig	release

Filter by Topic:

all
third-party
conference
announcement
release
plug-in
standards
documentation
tutorial
miscellaneous
source

Рис. 7.31. *Ссылка all (все) с самого начала отмечена как активная*

Взаимодействие с другим программным кодом

На примерах сортировки и разбивки на страницы мы узнали, что нельзя рассматривать различные возможности независимо друг от друга. Встраиваемые нами реализации поведения иногда могут оказывать неожиданное влияние друг на друга. По этой причине стоит повторно пересмотреть то, что было реализовано ранее, с целью анализа влияния, оказываемого на добавленную функцию фильтрации.

Чередующееся окрашивание строк

Функция выделения цветом чередующихся строк, реализованная ранее, вступает в конфликт с нашей новой функцией фильтрации. Так как после наложения фильтра выделение чередующихся строк не выполняется, строки сохраняют свою окраску, как если бы отфильтрованные строки все еще присутствовали в таблице.

Для учета отфильтрованных строк реализация окрашивания чередующихся строк должна обладать возможностью обнаруживать их. В выбо-

ре корректного набора строк для визуального оформления их чередования нам поможет псевдокласс :visible из библиотеки jQuery. При внесении соответствующих вышесказанному изменений мы можем подготовить программный код, выполняющий окрашивание чередующихся строк, к вызову из других мест в сценарии, создав для этого нестандартный тип события, как делалось, когда мы добивались корректной совместной работы функций сортировки и разбивки на страницы.

```
$(document).ready(function() {
  $('table.striped').bind('stripe', function() {
    $('tbody', this).each(function() {
      $(this).find('tr:visible:not(:has(th))')
        .removeClass('odd').addClass('even')
        .filter(function(index) {
          return (index % 6) < 3;
        }).removeClass('even').addClass('odd');
    });
  }).trigger('stripe');
});
```

Теперь каждый раз, когда выполняется операция фильтрации, можно вызывать $table.trigger('stripe'). Благодаря новому обработчику события и программному коду, возбуждающему его, сейчас после операции фильтрации производится корректное оформление чередования строк, как показано на рис. 7.32.

Date	Headline	Author	Topic
2008			
Jul 14	jQuery UI 1.5.2	Paul Bakaus	release
Jun 26	jQuery UI 1.5.1	Paul Bakaus	release
Jun 9	jQuery UI v1.5 Released, Focus on Consistent API and Effects	Paul Bakaus	release
Jun 4	jQuery 1.2.6: Events 100% faster	John Resig	release
Feb 8	jQuery 1.2.3: AIR, Namespacing, and UI Alpha	John Resig	release
Jan 15	jQuery 1.2.2: 2nd Birthday Present	John Resig	release
2007			
Sep 10	jQuery 1.2: jQuery.extend("Awesome")	John Resig	release
Aug 24	jQuery 1.1.4: Faster, More Tests, Ready for	John	release

Filter by Topic:
all
third-party
conference
announcement
release
plug-in
standards
documentation
tutorial
miscellaneous
source

Рис. 7.32. *После исправления проблемы влияния функции фильтрации на чередование окраски строк*

Развертывание и свертывание

Реализация развертывания и свертывания разделов таблицы, добавленная ранее, также вступает в конфликт с реализацией фильтров.

Если раздел был свернут, то в случае выбора нового фильтра соответствующие ему строки отображаются на странице несмотря на то, что раздел был свернут. Напротив, если раздел был развернут, то в случае применения фильтра отображаться будут все строки раздела независимо от того, соответствуют ли они фильтру или нет.

Один из способов исправить последнюю ситуацию состоит в том, чтобы изменить способ отображения и сокрытия строк. Если для сокрытия строк использовать класс CSS, тогда нам не потребуется явно вызывать методы `.hide()` и `.show()`. Заменив в реализации свертывания разделов вызов метода `.hide()` на вызов `.addClass('filtered')` и вызов метода `.show()` на вызов `.removeClass('filtered')` и добавив определение класса CSS, мы сможем обеспечить корректное сокрытие и отображение строк. При удалении класса из строк в свернутом разделе они больше не будут отображаться по ошибке.

Добавление нового класса `filtered` поможет нам и в решении обратной проблемы. Мы можем реализовать проверку отфильтрованных строк при развертывании раздела, пропуская их, вместо того чтобы отображать. Проверка наличия данного класса – это лишь вопрос добавления `:not(.filtered)` в селектор, используемый реализацией развертывания раздела.

Теперь наши реализации работают, как ожидается, и каждая из них способна скрывать и отображать строки независимо от другой.

Окончательная версия

Наш второй пример страницы демонстрирует реализацию чередования окраски, выделения строк, подсказок, свертывания/развертывания разделов и фильтрации. Ниже приводится окончательная версия программного кода JavaScript, полученная в результате совместного использования всех функций:

```
$(document).ready(function() {
  $('table.striped').bind('stripe', function() {
    $('tbody', this).each(function() {
      $(this).find('tr:visible:not(:has(th))')
        .removeClass('odd').addClass('even')
        .filter(function(index) {
          return (index % 6) < 3;
        }).removeClass('even').addClass('odd');
    });
  }).trigger('stripe');
});

$(document).ready(function() {
  var $authorCells = $('table.striped td:nth-child(3)');
  var $tooltip = $('<div id="tooltip"></div>').appendTo('body');

  var positionTooltip = function(event) {
```

```
    var tPosX = event.pageX;
    var tPosY = event.pageY + 20;
    $tooltip.css({top: tPosY, left: tPosX});
  };
  var showTooltip = function(event) {
    var authorName = $(this).text();
    var action = 'Highlight';
    if ($(this).parent().is('.highlight')) {
      action = 'Unhighlight';
    }
    $tooltip
      .text(action + ' all articles by ' + authorName)
      .show();
    positionTooltip(event);
  };
  var hideTooltip = function() {
    $tooltip.hide();
  };

  $authorCells
    .addClass('clickable')
    .hover(showTooltip, hideTooltip)
    .mousemove(positionTooltip)
    .click(function(event) {
      var authorName = $(this).text();
      $authorCells.each(function(index) {
        if (authorName == $(this).text()) {
          $(this).parent().toggleClass('highlight');
        }
        else {
          $(this).parent().removeClass('highlight');
        }
      });
      showTooltip.call(this, event);
    });
});

$(document).ready(function() {
  var collapseIcon = '../images/bullet_toggle_minus.png';
  var collapseText = 'Collapse this section';
  var expandIcon = '../images/bullet_toggle_plus.png';
  var expandText = 'Expand this section';
  $('table.collapsible tbody').each(function() {
    var $section = $(this);
    $('<img />').attr('src', collapseIcon)
      .attr('alt', collapseText)
      .prependTo($section.find('th'))
      .addClass('clickable')
      .click(function() {
        if ($section.is('.collapsed')) {
          $section.removeClass('collapsed')
```

```
            .find('tr:not(:has(th)):not(.filtered)')
            .fadeIn('fast');
          $(this).attr('src', collapseIcon)
            .attr('alt', collapseText);
        }
        else {
          $section.addClass('collapsed')
            .find('tr:not(:has(th))')
            .fadeOut('fast', function() {
              $(this).css('display', 'none');
            });
          $(this).attr('src', expandIcon)
            .attr('alt', expandText);
        }
        $section.parent().trigger('stripe');
      });
  });
});

$(document).ready(function() {
  $('table.filterable').each(function() {
    var $table = $(this);

    $table.find('th').each(function(column) {
      if ($(this).is('.filter-column')) {
        var $filters = $('<div class="filters"></div>');
        $('<h3></h3>')
          .text('Filter by ' + $(this).text() + ':')
          .appendTo($filters);
        $('<div class="filter">all</div>').click(function() {
          $table.find('tbody tr').removeClass('filtered');
          $(this).addClass('active')
            .siblings().removeClass('active');
          $table.trigger('stripe');
        }).addClass('clickable active').appendTo($filters);

        var keywords = {};
        $table.find('td:nth-child(' + (column + 1) + ')')
          .each(function() {
            keywords[$(this).text()] = $(this).text();
          });

        $.each(keywords, function(index, keyword) {
          $('<div class="filter"></div>').text(keyword)
            .bind('click', {key: keyword}, function(event) {
              $('tr:not(:has(th))', $table).each(function() {
                var value = $('td', this).eq(column).text();
                if (value == event.data['key']) {
                  $(this).removeClass('filtered');
                }
                else {
```

```
                $(this).addClass('filtered');
              }
            });
            $(this).addClass('active')
              .siblings().removeClass('active');
            $table.trigger('stripe');
          }).addClass('clickable').appendTo($filters);
        });
        $filters.insertBefore($table);
      }
    });
  });
});
```

В заключение

В этой главе мы исследовали некоторые методы формирования и отображения срезов данных в таблицах для наших веб-страниц, способные превратить их в красивые и функциональные контейнеры представления данных. Мы рассмотрели вопросы, связанные с **сортировкой** таблиц с использованием в качестве ключей сортировки различных типов данных (слов, чисел, дат) и с **разбивкой на страницы**, применяемой для упрощения обзора частей таблицы. Мы познакомились с усложненными приемами **чередования окраски строк** и с реализованными в JavaScript **подсказками**. Кроме того, мы реализовали **развертывание** и **свертывание** содержимого, а также **фильтрацию** и **выделение** строк, отвечающих заданному критерию.

Мы даже коротко коснулись некоторых более сложных тем, таких как реализация сортировки и разбивки на страницы на стороне сервера с использованием технологии AJAX, динамическое вычисление координат элементов страницы и создание модуля расширения для библиотеки jQuery.

Как мы уже убедились, семантика разметки HTML для таблиц заключает в себе массу тонкостей и сложностей. К счастью, библиотека jQuery способна помочь нам приручить этого зверя, обеспечивая мощные средства отображения табличных данных.

8

Интерактивные формы

Практически все веб-сайты, требующие обратной связи с пользователем, используют **формы** в том или ином виде. На протяжении всего времени существования Интернета формы играли роль вьючного мула, надежно доставляющего информацию от конечного пользователя на веб-сайт – надежно, но не слишком грациозно и изящно. Возможно, этот недостаток был вызван трудностями, лежащими на пути к серверу и обратно, или, может быть, это было связано с бескомпромиссными элементами формы, не желающими следовать последним веяниям моды. Какова бы ни была причина, лишь недавно, с всплеском интереса к сценариям на стороне клиента, формы обрели новую силу, назначение и оформление. В этой главе мы рассмотрим способы, которые позволят нам вдохнуть новую жизнь в формы. Мы улучшим их оформление, создадим процедуры проверки содержимого, будем использовать их содержимое для вычислений и отправлять на сервер результаты, которые никто не видит.

Улучшение простой формы

Используя библиотеку jQuery для создания веб-сайтов, мы постоянно должны задаваться вопросом, как будут выглядеть страницы, когда у посетителя окажется отключенной поддержка JavaScript (если, конечно, заранее не известно, как будет настроен броузер у каждого посетителя). Однако это не означает, что мы не можем создавать более привлекательные или функциональные сайты для посетителей, у которых имеется поддержка JavaScript. Принцип **прогрессивного улучшения** пользуется большой популярностью в среде разработчиков на JavaScript, потому что он позволяет учитывать нужды всех пользователей, обеспечивая при этом более широкие возможности для большинства из них. Для демонстрации принципа прогрессивного улучшения применительно к формам мы создадим форму ввода контактной информации, внешний вид и функциональность которой мы будем улучшать с помощью библиотеки jQuery.

Прогрессивное улучшение оформления формы

Для начала внесем в нашу форму некоторые эстетические улучшения. При отключенной поддержке JavaScript набор полей формы будет отображаться, как показано на рис. 8.1.

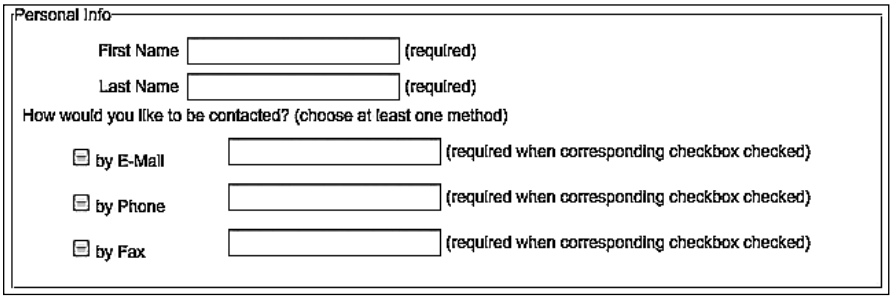


Рис. 8.1. *Первоначальный вид формы с полями*

Безусловно, форма выглядит достаточно функциональной и содержит значительный объем информации, которая поможет пользователю заполнить все поля формы, однако в нее определенно можно внести некоторые улучшения. Используя принцип прогрессивного улучшения, мы внесем в нее три усовершенствования:

1. Внесем изменения в дерево DOM, чтобы воспользоваться гибкостью оформления элемента `<legend>`.
2. Заменим текст возле обязательных полей (required – обязательное) звездочкой (*), а возле специальных полей (required only when the corresponding checkbox is checked – обязательное, только когда отмечен соответствующий флажок) – двумя звездочками (**). Метки возле всех обязательных полей выведем жирным шрифтом, а над формой добавим описание, что означают звездочки и двойные звездочки.
3. На этапе загрузки страницы скроем поля ввода, соответствующие флажкам, и будем отображать эти поля ввода или скрывать их, когда пользователь будет отмечать или снимать отметку с флажков.

Начнем с разметки HTML для тега `<fieldset>`:

```
<fieldset>
  <legend>Personal Info</legend>
  <ol>
    <li>
      <label for="first-name">First Name</label>
      <input class="required" type="text" name="first-name" id="first-name" />
      <span>(required)</span>
    </li>
    <li>
      <label for="last-name">Last Name</label>
      <input class="required" type="text" name="last-name" id="last-name" />
```

```
        <span>(required)</span>
      </li>
      <li>How would you like to be contacted? (choose at least one method)
        <ul>
          <li>
            <label for="by-email">
              <input type="checkbox" name="by-contact-type"
                                     value="E-mail" id="by-email" />
              by E-Mail
            </label>
            <input class="conditional" type="text" name="email" id="email" />
            <span>(required when corresponding checkbox checked)</span>
          </li>
          <li>
            <label for="by-phone">
              <input type="checkbox" name="by-contact-type"
                                     value="Phone" id="by-phone" />
              by Phone
            </label>
            <input class="conditional" type="text" name="phone" id="phone" />
            <span>(required when corresponding checkbox checked)</span>
          </li>
          <li>
            <label for="by-fax">
              <input type="checkbox" name="by-contact-type"
                value="Fax" id="by-fax" />
              by Fax
            </label>
            <input class="conditional" type="text" name="fax" id="fax" />
            <span>(required when corresponding checkbox checked)</span>
          </li>
        </ul>
      </li>
    </ol>
  </fieldset>
```

Обратите внимание, что каждый элемент или пара элементов является **элементом списка (**<li>**).** Все элементы находятся внутри **упорядоченного списка (**<ol>**),** а флажки (вместе с соответствующими им текстовыми полями ввода) помещены во вложенные **неупорядоченные списки (**<ul>**).** Кроме того, для оформления подписей к каждому полю используются элементы <label>. Для текстовых полей элемент <label> предшествует элементу <input>, для флажков – окружает элемент <input>. Несмотря на то, что нет никакой «стандартной» схемы размещения полей внутри элемента fieldset, упорядоченный список выглядит наиболее предпочтительным для представления семантического назначения элементов в форме ввода контактной информации.

Создав разметку HTML, можно приступать к использованию библиотеки jQuery для прогрессивного улучшения.

Легенда

Легенда формы – это один из элементов форм, наиболее трудно поддающихся оформлению с помощью стилей CSS. Несовместимость броузеров и ограничения в выборе позиции превращают работу с этим элементом в одну сплошную неприятность. И все же если мы заинтересованы в использовании выразительных, хорошо оформленных элементов страницы, элемент `legend` является привлекательным, пусть и не визуально, средством отображения заголовка для элемента формы `<fieldset>`.

Имея только разметку HTML и стили CSS, мы вынуждены выбирать между семантикой разметки и гибкостью верстки. Однако, мы можем изменить разметку HTML во время загрузки страницы, превратив каждый элемент `<legend>` в элемент `<h3>` для тех пользователей, чьи броузеры поддерживают JavaScript; пользователи с отключенной поддержкой JavaScript по-прежнему будут видеть элемент `<legend>`. Это легко можно реализовать с помощью метода `.replaceWith()`:

```
$(document).ready(function() {
  $('legend').each(function(index) {
    $(this).replaceWith('<h3>' + $(this).text() + '</h3>');
  });
});
```

Обратите внимание, что в данном случае мы не можем полагаться на механизм неявных итераций библиотеки jQuery. При замещении каждого элемента нам необходимо вставить уникальное текстовое содержимое этого элемента. Поэтому мы используем метод `.each()`, который позволяет извлекать конкретное содержимое с помощью вызова `$(this)`.

Теперь, когда для элемента `<h3>` в таблице стилей мы определили синий цвет фона и белый цвет текста, первый набор полей формы будет выглядеть, как показано на рис. 8.2.

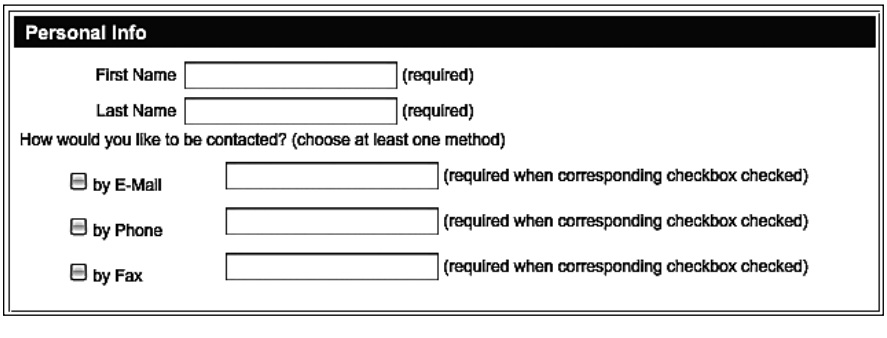


Рис. 8.2. *Вид формы после оформления заголовка*

Альтернативный вариант, когда элементы `<legend>` остаются нетронутыми, заключается в обертывании их содержимого в тег `<span>`:

```
$(document).ready(function() {
  $('legend').wrapInner('<span></span>');
});
```

У подхода, основанного на обертывании содержимого тега <legend> тегом , имеются, по меньшей мере, два преимущества перед заменой тега <legend> на <h3>: он сохраняет семантическое значение тега <legend> для устройств чтения с экрана, обладающих поддержкой JavaScript, и требует меньше усилий для создания сценария. Недостаток состоит в том, что при таком подходе сложнее добиться внешнего вида, напоминающего заголовок. Для этого нам потребуется как минимум установить значение свойства position для обоих элементов <fieldset> и , а также значения свойства padding-top элемента <fieldset> и свойства width элемента :

```
fieldset {
  position: relative;
  padding-top: 1.5em;
}

legend span {
  position: absolute;
  width: 100%;
}
```

Выберем ли мы способ, основанный на замещении элементов <legend> формы, или способ, основанный на добавлении в них элементов , в любом случае теперь они имеют привлекательное оформление, достаточное для наших целей, поэтому мы можем заняться удалением сообщений, находящихся рядом с обязательными полями.

Сообщения для обязательных полей

В нашей форме ввода контактной информации обязательные поля имеют атрибут class="required", что позволяет использовать его не только для реализации отклика на ввод пользователя, но и для визуального оформления. Все поля ввода, определяющие способ связи с пользователем, имеют атрибут class="conditional". Мы будем использовать эти классы для замены текста инструкций, выводимых в круглых скобках справа от каждого поля ввода.

Начнем с установки значений переменных requiredFlag и conditionalFlag, а затем заполним элементы , следующие за обязательными и условно-обязательными полями, текстом, хранящимся в этих переменных:

```
$(document).ready(function() {
  var requiredFlag = ' * ';
  var conditionalFlag = ' ** ';

  $('form :input')
    .filter('.required')
```

```
        .next('span').text(requiredFlag).end()
      .end()
      .filter('.conditional')
        .next('span').text(conditionalFlag);
});
```

Использование метода `.end()` позволяет нам расширить цепочку методов, чтобы мы могли продолжать работать с тем же набором элементов и свести к минимуму количество операций по созданию объектов и обходу дерева DOM. Вызов метода `.end()` выполняет один шаг назад в цепочке операций выбора элементов, возвращая набор элементов, который существовал перед последним вызовом метода, выполняющего обход дерева DOM. Здесь мы выполняем подряд два вызова метода `.end()`: первый вызов возвращает набор элементов, полученный вызовом метода `.filter('.required')`, а второй возвращает набор элементов, полученный вызовом функции `$('form :input')`. Благодаря этому, когда производится вызов `.filter('.conditional')`, отбирающий элементы с атрибутом `class="conditional"`, выборка производится из всех элементов ввода внутри формы.

Далее, так как одиночная звездочка (*) может оказаться не очень заметным индикатором для привлечения внимания пользователя, для всех обязательных полей мы также добавим к элементу `<label>` атрибут `class="req-label"` и применим правило `font-weight:bold` к этому классу. Для этого можно расширить цепочку методов еще больше.

```
$(document).ready(function() {
  var requiredFlag = ' * ';
  var conditionalFlag = ' ** ';

  $('form :input')
    .filter('.required')
      .next('span').text(requiredFlag).end()
      .prev('label').addClass('req-label').end()
    .end()
    .filter('.conditional')
      .next('span').text(conditionalFlag);
});
```

Такая длинная цепочка методов достаточно сложно воспринимается на глаз, поэтому в подобных случаях большое значение имеют непротиворечивое использование разрывов строк и введение отступов.

Набор полей после изменения текста и добавления класса теперь выглядит, как показано на рис. 8.3.

Неплохо. Однако в действительности не было ничего плохого в сообщениях возле обязательных и условно-обязательных полей, просто они повторялись слишком часто. Возьмем первые экземпляры каждого сообщения и отобразим их над формой рядом с индикаторами, символизирующими их.

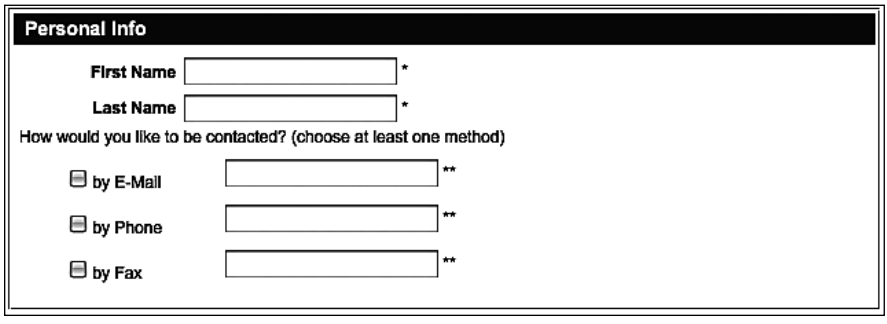


Рис. 8.3. *Вид формы после добавления класса req-label и изменения текста*

Прежде чем заполнить элементы `<span>` сообщениями, соответствующими индикаторам, нам необходимо сохранить их в паре переменных. Затем мы сможем убрать круглые скобки, используя **регулярное выражение:**

```
$(document).ready(function() {
  var requiredFlag = ' * ';
  var conditionalFlag = ' ** ';

  var requiredKey = $('input.required:first') .next('span').text();
  var conditionalKey = $('input.conditional:first') .next('span').text();

  requiredKey = requiredFlag + requiredKey.replace(/^\((.+)\)$/,'$1');
  conditionalKey = conditionalFlag +
                   conditionalKey.replace(/^\((.+)\)$/,'$1');

  // ... продолжение программного кода
});
```

Первые две дополнительные строки объявляют переменные `requiredKey` и `conditionalKey`, в которых сохраняются тексты сообщений двух типов полей. Следующие две строки изменяют текст в этих переменных, объединяя каждый индикатор с соответствующим ему текстом и исключая круглые скобки. Возможно, регулярное выражение и использование метода `.replace()` нуждаются в дальнейших пояснениях.

Регулярное выражение

Регулярное выражение заключено между двумя символами слеша и имеет вид: `/^\((.+)\)$/`. Первый символ, `^`, указывает, что сопоставление должно начинаться с начала строки. За ним следуют два символа, `\(`, совпадающие с открывающей круглой скобкой. Обратный слеш **экранирует** следующий за ним символ, сообщая механизму регулярных выражений, что он должен интерпретироваться буквально. Это совершенно необходимо, потому что круглые скобки входят в число символов, имеющих специальное назначение в регулярных выражениях, как мы

увидим далее. Следующие четыре символа, (.+), совпадают с одним или более (о чем говорит квантификатор +) символами любого вида в той же строке (символу . соответствует любой символ) и помещают их в группу благодаря использованию круглых скобок. Последние три символа, \)$, совпадают с закрывающей скобкой в конце строки. Таким образом, все регулярное выражение целиком совпадает со строкой, начинающейся с открывающей круглой скобки, за которой следует группа произвольных символов, и заканчивающейся закрывающей круглой скобкой.

Метод .replace() отыскивает в контекстной строке подстроку, совпадающую с указанным регулярным выражением, и замещает ее другой строкой. Синтаксис метода имеет следующий вид:

```
'context'.replace(/regular-expression/, 'replacement')
```

В качестве контекстных строк в двух наших вызовах метода .replace() используются переменные requiredKey и conditionalKey. Мы уже рассмотрели регулярное выражение, заключенное между двумя символами слеша. Запятая отделяет регулярное выражение от замещающей строки, которая в обоих случаях имеет вид '$1'. Комбинация $1 – это выражение-заполнитель, представляющее первую группу в регулярном выражении. Так как в нашем регулярном выражении имеется одна группа, содержащая один или более символов, которую с обеих сторон окружают круглые скобки, замещающая строка будет содержать все символы, за исключением круглых скобок.

Добавление легенды с сообщениями для полей

Теперь, когда мы получили сообщения без круглых скобок, можно вывести их над формой вместе с соответствующими им индикаторами:

```
$(document).ready(function() {
  var requiredFlag = ' * ';
  var conditionalFlag = ' ** ';

  var requiredKey = $('input.required:first')
                                    .next('span').text();
  var conditionalKey = $('input.conditional:first')
                                    .next('span').text();

  requiredKey = requiredFlag +
                 requiredKey.replace(/^\((.+)\)$/,'$1');
  conditionalKey = conditionalFlag +
                 conditionalKey.replace(/^\((.+)\)$/,'$1');

  $('<p></p>')
    .addClass('field-keys')
    .append(requiredKey + '<br />')
    .append(conditionalKey)
    .insertBefore('#contact');
});
```

Пять новых строк должны выглядеть знакомыми к настоящему моменту времени. Ниже описывается, что в них делается:

1. Создается новый элемент параграфа.
2. К параграфу добавляется класс `field-keys`.
3. В параграф добавляются содержимое переменной `requiredKey` и разрыв строки.
4. В параграф добавляется содержимое переменной `conditionalKey`.
5. Параграф вместе со всем своим содержимым вставляется в страницу перед формой.

Когда методу `.append()` передается строка с разметкой HTML, как это делается здесь, необходимо экранировать все специальные символы HTML, которые могут присутствовать в строке. В данном случае такое экранирование выполняет метод `.text()`, использовавшийся при объявлении переменных.

После определения свойств класса `.field-keys` в таблице стилей результат будет выглядеть, как показано на рис. 8.4.

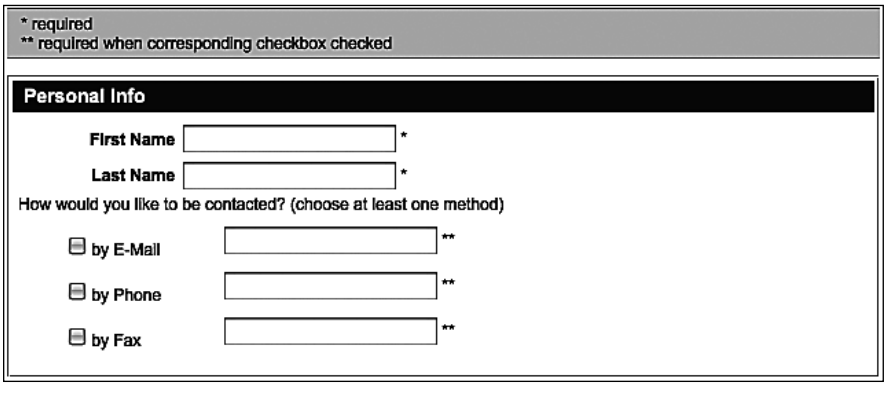


Рис. 8.4. *Вид формы с легендой после определения свойств класса .field-keys*

Наша работа с использованием библиотеки jQuery над первым набором полей полностью завершена.

Поля, отображаемые по условию

Продолжим улучшать группу полей, которые запрашивают у пользователя, как с ним связаться. Так как эти поля ввода становятся обязательными, только если отмечены соответствующие им флажки, на начальном этапе загрузки документа мы можем скрыть их, как и соответствующие им индикаторы:

```
$(document).ready(function() {
  $('input.conditional').next('span').andSelf().hide();
});
```

Теперь форма имеет более простой интерфейс, как показано на рис. 8.5:

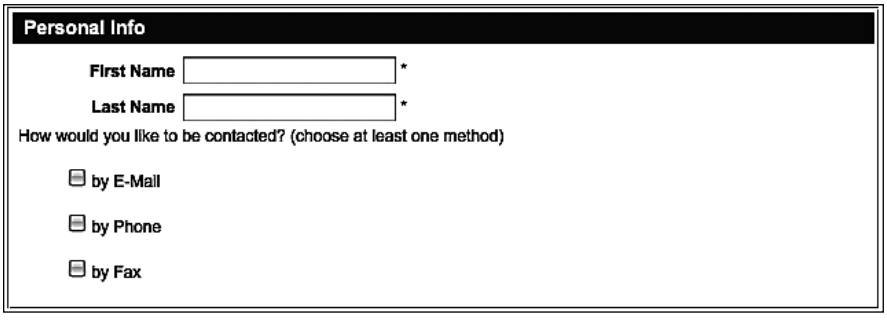


Рис. 8.5. *Вид формы после сокрытия условно-обязательных полей*

Чтобы обеспечить отображение текстовых полей ввода и индикаторов, мы подключим к каждому флажку обработчик события с помощью метода `.click()`. Эта операция выполняется в контексте каждого условно-обязательного поля ввода, поэтому мы можем определить пару переменных для многократного использования:

```
$(document).ready(function() {
  $('input.conditional').next('span').andSelf().hide()
    .end().end()
    .each(function() {
      var $thisInput = $(this);
      var $thisFlag = $thisInput.next('span');
      $thisInput.prev('label').find(':checkbox')
        .click(function() {
          // продолжение программного кода ...
        });
    });
});
```

Здесь мы снова два раза подряд вызвали метод `.end()`, на этот раз, чтобы вызвать метод `.each()` относительно исходного селектора `$('input.conditional')`.

Теперь у нас имеются переменные, содержащие текущее поле ввода и текущий индикатор. Когда пользователь выполняет щелчок на флажке, мы проверяем, был ли установлен флажок; и если был, отображаем поле ввода, индикатор и добавляем класс `req-label` к родительскому элементу `<label>`:

```
$(document).ready(function() {
  $('input.conditional').next('span').andSelf().hide()
    .end().end()
    .each(function() {
      var $thisInput = $(this);
      var $thisFlag = $thisInput.next('span');
```

```
        $thisInput.prev('label').find(':checkbox')
          .click(function() {
            if (this.checked) {
              $thisInput.show();
              $thisFlag.show();
              $(this).parent('label').addClass('req-label');
            }
          });
      });
});
```

Для проверки, отмечен ли флажок, в данном случае предпочтительнее использовать метод `this.checked`, потому что у нас имеется прямой доступ к узлу дерева DOM через ключевое слово `this`. Когда **узел дерева DOM** недоступен, можно использовать селектор `$('selector').is(':checked')`, благодаря тому что метод `.is()` возвращает значение типа **Boolean** (`true` или `false`).

Нам осталось сделать еще две вещи:

1. Снять отметки с флажков на этапе начальной загрузки страницы, так как некоторые броузеры сохраняют состояние элементов форм при обновлении страницы.
2. Добавить инструкцию `else`, которая будет скрывать условно-обязательные элементы и удалять класс `req-label` после снятия флажка.

```
$(document).ready(function() {
  $('input.conditional').next('span').andSelf().hide()
    .end().end()
    .each(function() {
      var $thisInput = $(this);
      var $thisFlag = $thisInput.next('span');
      $thisInput.prev('label').find(':checkbox')
        .attr('checked', false)
        .click(function() {
      if (this.checked) {
        $thisInput.show();
        $thisFlag.show();
        $(this).parent('label').addClass('req-label');
      } else {
        $thisInput.hide();
        $thisFlag.hide();
        $(this).parent('label')
          .removeClass('req-label');
      }
    });
});
```

На этом мы заканчиваем оформление внешнего вида формы. Далее мы добавим некоторые проверки содержимого полей ввода, выполняемые на стороне клиента.

Проверка содержимого формы

Прежде чем приступать к реализации проверки содержимого какой-либо формы с помощью библиотеки jQuery, необходимо запомнить одно важное правило: **проверка на стороне клиента** не отменяет необходимости проверки на стороне сервера. Мы не можем полагаться на то, что у всех пользователей будет включена поддержка JavaScript. Если действительно необходимо, чтобы некоторые поля были обязательными или информация вводилась в некотором специфическом формате, одна только поддержка JavaScript не может гарантировать ожидаемый результат. Некоторые пользователи предпочитают отключать поддержку JavaScript, некоторые устройства просто не имеют ее, а некоторые пользователи могут преднамеренно пытаться отправить злонамеренные данные в обход ограничений, реализованных с помощью JavaScript.

Зачем же тогда заниматься реализацией проверки с помощью jQuery? Проверка данных на стороне клиента, выполняемая с помощью jQuery, обладает одним преимуществом перед проверкой на стороне сервера: **быстрой обратной связи**. Программный код на стороне сервера, реализованный на ASP, PHP или каком-либо другом языке программирования, требует выполнения полной перезагрузки страницы для получения данных формы (если, конечно, доступ к нему не выполняется асинхронно, правда, для этого опять же необходима поддержка JavaScript). С помощью библиотеки jQuery, используя реализацию проверки каждого обязательного поля на стороне клиента, мы можем оперативно отреагировать на ввод пользователя, когда поле теряет фокус ввода (событие `blur`) или производится нажатие клавиши (событие `keyup`).

Обязательные поля

В нашей форме ввода контактной информации мы будем проверять содержимое каждого поля ввода, имеющего класс `required`, когда пользователь будет нажимать клавишу табуляции или выполнять щелчок мышью за пределами поля. Однако прежде чем перейти к соответствующему программному коду, мы должны ненадолго вернуться к нашим условно-обязательным полям ввода. Чтобы упростить процедуру проверки, мы можем добавлять класс `required` к элементу `<input>`, когда он видим, и удалять этот класс, когда выполняется скрытие элемента `<input>`. Ниже приводится программный код, реализующий эту особенность:

```
$thisInput.prev('label').find(':checkbox')
  .attr('checked', false)
  .click(function() {
    if (this.checked) {
      $thisInput.show().addClass('required');
      $thisFlag.show();
      $(this).parent('label').addClass('req-label');
    } else {
```

```
        $thisInput.hide().removeClass('required');
        $thisFlag.hide();
        $(this).parent('label').removeClass('req-label');
      }
    });
```

Разобравшись с классом `required`, мы готовы отреагировать на событие потери полем фокуса ввода, при котором оно остается пустым. Сообщение будет помещаться рядом с индикатором обязательного поля, а к элементу `<li>`, вмещающему это поле, будет добавляться класс `class="warning"` для получения возможности предупредить пользователя:

```
$(document).ready(function() {
  $('form :input').blur(function() {
    if ($(this).hasClass('required')) {
      var $listItem = $(this).parents('li:first');
      if (this.value == '') {
        var errorMessage = 'This is a required field';
        $('<span></span>')
          .addClass('error-message')
          .text(errorMessage)
          .appendTo($listItem);
        $listItem.addClass('warning');
      }
    }
  });
});
```

В обработчике события `blur`, подключаемого к каждому элементу формы, присутствуют две условные инструкции `if`: первая проверяет наличие класса `required`, а вторая, содержит ли поле пустую строку. В случае соблюдения обоих условий мы создаем сообщение об ошибке, вставляем его в элемент `<span class="error-message">` и добавляем все это в конец родительского элемента `<li>`.

Для условно-обязательных полей желательно создать немного иное сообщение, информирующее о том, что поле **обязательно, только когда отмечен соответствующий флажок**. Мы будем добавлять квалифицирующее сообщение в конец стандартного сообщения об ошибке. Для этого добавим еще одну условную инструкцию `if`, которая будет проверять наличие класса `conditional` только в случае выполнения двух первых условий:

```
$(document).ready(function() {
  $('form :input').blur(function() {
    if ($(this).hasClass('required')) {
      var $listItem = $(this).parents('li:first');
      if (this.value == '') {
        var errorMessage = 'This is a required field';
        if ($(this).hasClass('conditional')) {
          errorMessage += ', when its related ' +
```

```
                        'checkbox is checked';
        }
        $('<span></span>')
          .addClass('error-message')
          .text(errorMessage)
          .appendTo($listItem);
        $listItem.addClass('warning');
      }
    }
  });
});
```

Когда поле ввода теряет фокус ввода в первый раз, наш программный код прекрасно справляется со своими обязанностями; однако, когда пользователь повторно входит в поле ввода и выходит из него, возникают две проблемы, как показано на рис. 8.6.

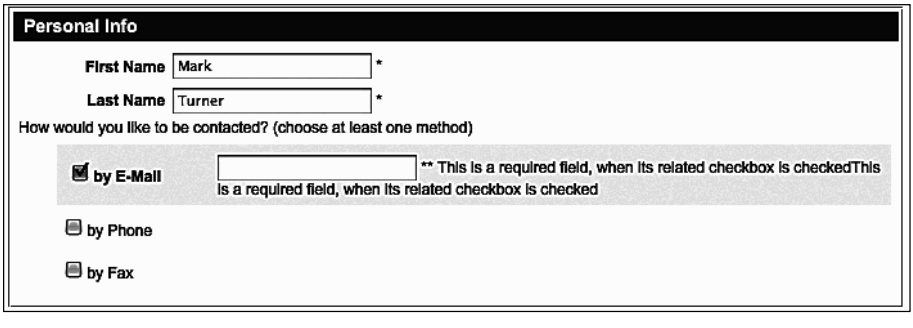


Рис. 8.6. *Многократное повторение текста сообщения об ошибке*

Если поле по-прежнему остается незаполненным, сообщение об ошибке будет повторено столько раз, сколько раз пользователь будет покидать поле. Если пользователь заполнит поле, класс `class="warning"` не будет удален. Безусловно, нам необходимо, чтобы рядом с полем выводилось только одно сообщение, и нам требуется удалить сообщение, когда пользователь исправит ошибку. Ликвидировать обе проблемы можно удалением класса `class= "warning"` из родительского элемента `<li>` текущего поля и `<span class="error-message">` из того же элемента `<li>` всякий раз, когда поле теряет фокус ввода, непосредственно перед выполнением проверки:

```
$(document).ready(function() {
  $('form :input').blur(function() {
    $(this).parents('li:first').removeClass('warning')
      .find('span.error-message').remove();
    if ($(this).hasClass('required')) {
      var $listItem = $(this).parents('li:first');
      if (this.value == '') {
        var errorMessage = 'This is a required field';
```

```
        if ($(this).hasClass('conditional')) {
          errorMessage += ', when its related checkbox
                          is checked';
        }
        $('<span></span>')
          .addClass('error-message')
          .text(errorMessage)
          .appendTo($listItem);
        $listItem.addClass('warning');
      }
    }
  });
});
```

Наконец, мы получили функционирующий сценарий проверки обязательных и условно-обязательных полей. Даже после повторного входа и выхода из обязательного поля сообщение об ошибке отображается корректно, как показано на рис. 8.7.

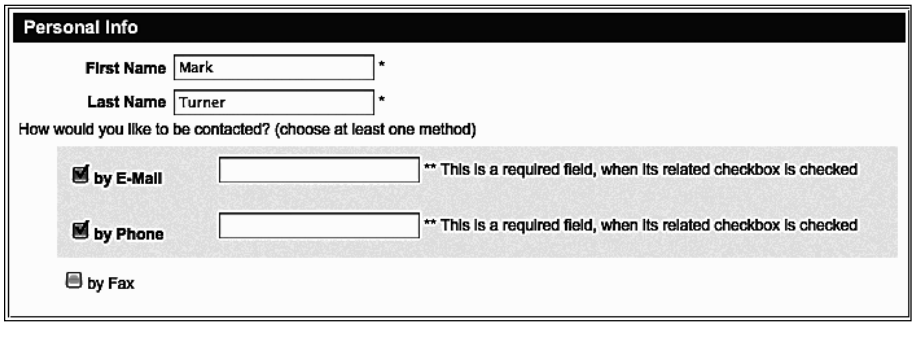


Рис. 8.7. *Теперь текст сообщения об ошибке выводится всего один раз*

Но минутку! Нам необходимо также удалять класс `warning` и `<span class="error-message">` из элемента `<li>`, когда пользователь снимает соответствующий флажок! Для реализации этого мы можем вернуться к программному коду, обрабатывающему флажки, и добавить в него запуск события `blur` для соответствующего поля ввода в случае снятия флажка:

```
if (this.checked) {
  $thisInput.show().addClass('required');
  $thisFlag.show();
  $(this).parent('label').addClass('req-label');
} else {
  $thisInput.hide().removeClass('required').blur();
  $thisFlag.hide();
  $(this).parent('label').removeClass('req-label');
}
```

Теперь, когда пользователь будет снимать флажок, соответствующий класс `warning` и сообщение об ошибке будут удаляться с глаз долой и из сердца вон.

Обязательные форматы

Существует еще один вид проверки, который можно реализовать в нашей форме ввода контактной информации, – проверка соблюдения **форматов ввода**. Иногда бывает полезно вывести предупреждение, если поле заполнено с ошибкой (в противоположность случаю, когда поле остается пустым). Основные кандидаты на реализацию проверки подобного вида – это поля ввода адреса электронной почты, номера телефона и номера кредитной карты. В демонстрационных целях для проверки содержимого поля ввода адреса электронной почты мы будем использовать относительно простое регулярное выражение. Прежде чем погружаться в исследование деталей регулярного выражения, рассмотрим полную реализацию проверки адреса электронной почты:

```
$(document).ready(function() {
  // ... продолжение программного кода ...
  if (this.id == 'email') {
    var $listItem = $(this).parents('li:first');
    if ($(this).is(':hidden')) {
      this.value = '';
    }
    if (this.value != '' &&
        !/.+@.+\.[a-zA-Z]{2,4}$/.test(this.value)) {
      var errorMessage = 'Please use proper e-mail format'
                       + ' (e.g. joe@example.com)';
      $('<span></span>')
        .addClass('error-message')
        .text(errorMessage)
        .appendTo($listItem);
      $listItem.addClass('warning');
    }
  }
  // ... продолжение программного кода ...
});
```

Этот программный код выполняет следующие действия:

- Проверяет значение атрибута `id` поля, если проверка увенчалась успехом:
  - Записывает в переменную ссылку на родительский элемент списка.
  - Проверяет, скрыто ли поле ввода адреса электронной почты. Если поле скрыто (что возможно, если снят соответствующий флажок), в качестве его значения устанавливается пустая строка. Это позволит программному коду, рассматривавшемуся ранее, корректно удалить класс `warning` и сообщение об ошибке.

- Проверяет, было ли заполнено поле и не соответствует ли его значение регулярному выражению. Если эти две проверки оканчиваются успехом, тогда сценарий:
  - Создаст сообщение об ошибке
  - Вставит сообщение в элемент `<span class="error-message">`
  - Добавит элемент `<span class="error-message">` со всем его содержимым в родительский элемент списка
  - Добавит класс `warning` в родительский элемент списка

Теперь отдельно рассмотрим регулярное выражение:

```
!/.+@.+\.[a-zA-Z]{2,4}$/.test(this.value)
```

Несмотря на то, что это регулярное выражение похоже на использовавшееся выше в этой главе, для работы с ним вместо метода `.replace()` используется метод `.test()`, поскольку нам необходимо лишь получить значение `true` или `false`. Как и прежде, регулярное выражение заключено в символы слеша. Оно проверяет соответствие строки, заключенной в круглые скобки в вызове метода `.test()` и в данном случае представляющей содержимое поля ввода адреса электронной почты.

С помощью этого регулярного выражения мы проверяем совпадение с группой из одного или более символов, не являющихся символами разрыва строки (`.+`), за которой следует символ `@`, за которым в свою очередь следует другая группа из одного или более символов, не являющихся символами разрыва строки. До настоящего момента проверку могла бы пройти любая строка, такая как `lucia@example`, как и миллионы других возможных перестановок, хотя она не является допустимым адресом электронной почты.

Мы можем сделать проверку более строгой, обеспечив проверку наличия символа `.`, за которым могут следовать от двух до четырех букв в диапазоне от `a` до `z`, завершающих строку. Именно такую проверку реализует оставшаяся часть регулярного выражения. Она сначала проверяет совпадение с символом в диапазоне от `a` до `z` или от `A` до `Z` — `[a-zA-Z]`. Затем требует, чтобы проверяемая строка содержала от двух до четырех букв из указанного диапазона — `{2,4}`. Наконец, она требует, чтобы эти буквы находились в самом конце строки: `$`. Теперь для такой строки, как `lucia@example.com`, будет возвращаться значение `true`, тогда как для строки `lucia@example`, или `lucia@example.2fn`, или `lucia@example.example`, или `lucia-example.com` будет возвращаться значение `false`.

Но нам необходимо, чтобы значение `true` возвращалось (то есть создавалось сообщение об ошибке и выполнялись другие сопутствующие действия), только когда введенная строка *не* соответствует формату адреса электронной почты. По этой причине мы добавили перед регулярным выражением знак восклицания (**оператор not**):

```
!/.+@.+\.[a-zA-Z]{2,4}$/.test(this.value)
```

Заключительная проверка

На этом реализация проверки содержимого формы ввода контактной информации почти закончена. Однако мы могли бы выполнить еще одну проверку в момент, когда пользователь попытается отправить ее, на этот раз всех полей формы сразу. С помощью обработчика события `.submit()` формы (а не кнопки Send (отправить)) мы запускаем событие `blur` для всех обязательных полей:

```
$(document).ready(function() {
  $('form').submit(function() {
    $('#submit-message').remove();
    $(':input.required').trigger('blur');
  });
});
```

Обратите внимание на строку, где удаляется элемент, который еще не существует: `<div id="submit-message">`. Мы добавим этот элемент на следующем шаге. Здесь мы просто заранее предусмотрели его удаление, зная, что это придется сделать, на примере встретившихся ранее в этой главе проблем, которые проявлялись в создании нескольких сообщений об ошибках.

После запуска событий `blur` к некоторым элементам в текущей форме может добавиться класс `warning`. Если появится хотя бы один такой элемент, мы создадим новый элемент `<div id="submit-message">` и вставим его непосредственно перед кнопкой Send, куда вероятнее всего в этот момент смотрит пользователь. Мы также прервем операцию передачи формы:

```
$(document).ready(function() {
  $('form').submit(function() {
    $('#submit-message').remove();
    $(':input.required').trigger('blur');
    var numWarnings = $('.warning', this).length;
    if (numWarnings) {
      $('<div></div>')
        .attr({
          'id': 'submit-message',
          'class': 'warning'
        })
        .append('Please correct errors with ' +
                numWarnings + ' fields')
        .insertBefore('#send');
        return false;
    }
  });
});
```

Помимо типичного предложения исправить ошибки, сообщение содержит число полей, в которые необходимо внести исправления, как показано на рис. 8.8.

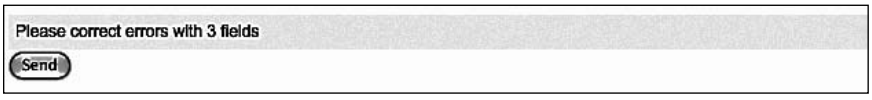


Рис. 8.8. *Сообщение указывает количество полей с ошибками*

Можно сделать даже лучше – вместо того, чтобы выводить число ошибок, мы можем перечислить названия полей, содержащих ошибки:

```
$(document).ready(function() {
  $('form').submit(function() {
    $('#submit-message').remove();
    $(':input.required').trigger('blur');
    var numWarnings = $('.warning', this).length;
    if (numWarnings) {
      var list = [];
      $('.warning label').each(function() {
        list.push($(this).text());
      });
      $('<div></div>')
        .attr({
          'id': 'submit-message',
          'class': 'warning'
        })
        .append('Please correct errors with the following ' +
      numWarnings + ' fields:<br />')
        .append('&bull; ' + list.join('<br />&bull; '))
        .insertBefore('#send');
      return false;
    };
  });
});
```

Во-первых, мы добавили переменную `list`, которую инициализировали пустым массивом. Затем мы получили набор элементов `label`, являющихся потомками элементов с классом `warning`, и добавили их текстовое содержимое в массив `list` (с помощью обычной функции JavaScript `push()`). Теперь текст каждой подписи хранится в виде отдельного элемента массива `list`.

Мы немного изменили предыдущую версию содержимого элемента `<div id="submit-message">` и добавили в него содержимое нашего массива `list`. Для преобразования массива в строку была использована обычная функция JavaScript `join()`; мы объединили все элементы массива, разделив их символом разрыва строки и маркером. В результате теперь сообщение выглядит так, как показано на рис. 8.9.

Следует признать, что разметка HTML, объединяющая поля в список, служит целям представления и не несет семантической нагрузки. Однако, из-за недолговечности этого списка, который был создан программным кодом JavaScript на последнем этапе и должен исчезнуть как мож-

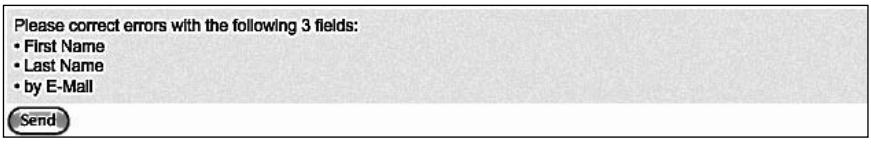


Рис. 8.9. *Сообщение содержит имена полей с ошибками*

но скорее, мы простим небрежность нашему коду ради его краткости и простоты.

Манипулирование флажками

В нашей форме ввода контактной информации имеется также раздел Miscellaneous (прочее), который содержит список флажков, как показано на рис. 8.10.

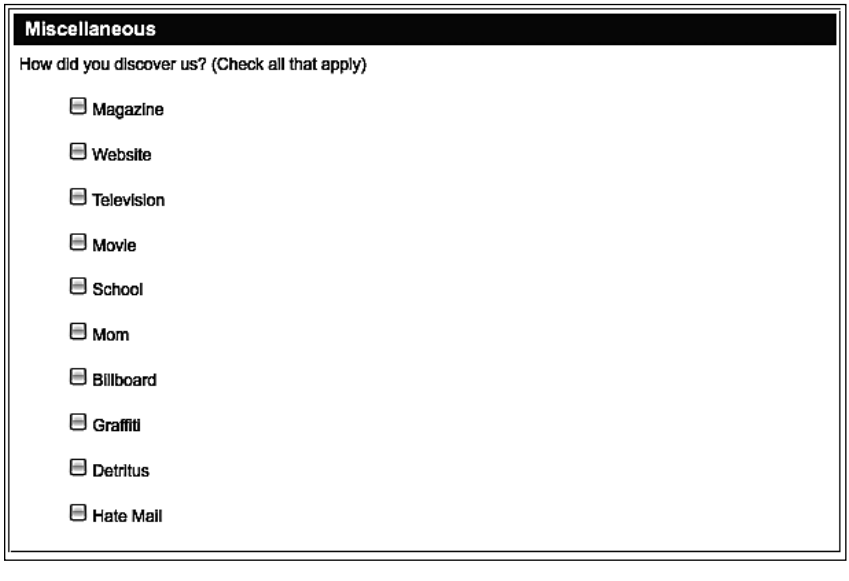


Рис. 8.10. *Список флажков в разделе Miscellaneous (прочее)*

Завершая усовершенствовать форму, мы поможем пользователю управлять этим списком. Группа из 10 флажков может показаться устрашающей, особенно если пользователю требуется пометить большую их часть. Возможность отметить или снять отметки со всех флажков оказывается весьма кстати в подобной ситуации. Давайте обеспечим такую возможность.

Для начала создадим новый элемент `<li>`, добавим в него элемент `<label>`, внутрь которого поместим элемент `<input type="checkbox" id="discover-all">` с поясняющим текстом, и вставим все это перед элементом `<ul>` внутри `<li class="discover">`:

```
$(document).ready(function() {
  $('<li></li>')
    .html('<label><input type="checkbox" id="discover-all" />' +
                                  ' <em>check all</em></label>')
    .prependTo('li.discover > ul');
});
```

Теперь у нас появился новый флажок с подписью check all (отметить все). Но он пока еще ничего не делает. Нам необходимо задействовать его метод .click():

```
$(document).ready(function() {
  $('<li></li>')
    .html('<label><input type="checkbox" id="discover-all" />' +
                                  ' <em>check all</em></label>')
    .prependTo('li.discover > ul');
  $('#discover-all').click(function() {
    var $checkboxes = $(this).parents('ul:first')
      .find(':checkbox');
    if (this.checked) {
      $checkboxes.attr('checked', true);
    } else {
      $checkboxes.attr('checked', '');
    }
  });
});
```

Внутри этого обработчика события мы сначала устанавливаем значение переменной $checkboxes, которое представляет собой объект jQuery, содержащий все флажки в текущем списке. Получив переменную, мы можем манипулировать флажками, просто отмечая их, если отмечен флажок check all, и снимая отметки, если флажок check all не отмечен.

Последний штрих, который можно добавить к этому флажку, заключается в том, чтобы добавить класс checkall к подписи флажка check all и изменить ее текст на un-check all (снять все отметки), после того как пользователь отметит его:

```
$(document).ready(function() {
  $('<li></li>')
    .html('<label><input type="checkbox" id="discover-all" />' +
                                  ' <em>check all</em></label>')
    .prependTo('li.discover > ul');
  $('#discover-all').click(function() {
    var $checkboxes = $(this) .parents('ul:first')
      .find(':checkbox');
    if (this.checked) {
      $(this).next().text(' un-check all');
      $checkboxes.attr('checked', true);
    } else {
      $(this).next().text(' check all');
      $checkboxes.attr('checked', '');
```

```
        };
      })
      .parent('label').addClass('checkall');
  });
```

Теперь группа флажков вместе с флажком check all выглядит, как показано на рис. 8.11.

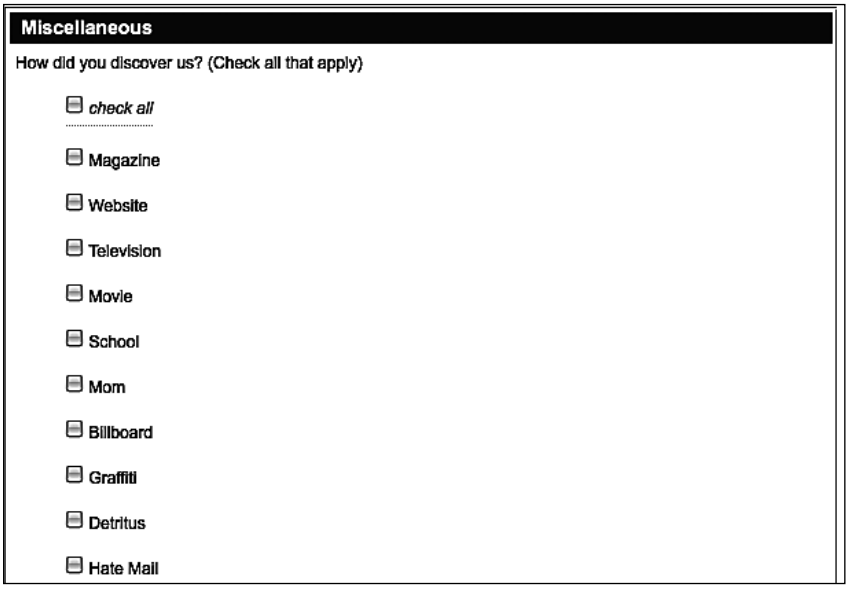


Рис. 8.11. *В форму добавлен флажок check all (отметить все)*

А после того как флажок check all будет отмечен, группа будет выглядеть как на рис. 8.12.

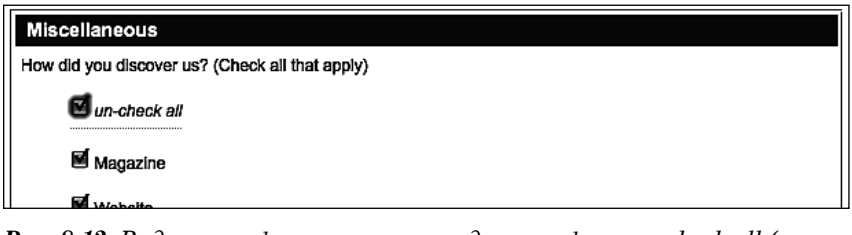


Рис. 8.12. *Вид списка флажков после выделения флажка check all (отметить все)*

Окончательная версия

Ниже приводится окончательная версия сценария для формы ввода контактной информации:

```
$(document).ready(function() {
  // Добавить оформление элементов формы.
  $('legend').each(function(index) {
    $(this).replaceWith('<h3>' + $(this).text() + '</h3>');
  });

  var requiredFlag = ' * ';
  var conditionalFlag = ' ** ';
  var requiredKey = $('input.required:first') .next('span').text();
  var conditionalKey = $('input.conditional:first') .next('span').text();
  requiredKey = requiredFlag + requiredKey.replace(/^\((.+)\)$/,'$1');
  conditionalKey = conditionalFlag +
                   conditionalKey.replace(/^\((.+)\)$/,'$1');
  $('<p></p>')
    .addClass('field-keys')
    .append(requiredKey + '<br />')
    .append(conditionalKey)
    .insertBefore('#contact');

  $('form :input')
    .filter('.required')
      .next('span').text(requiredFlag).end()
      .prev('label').addClass('req-label').end()
    .end()
    .filter('.conditional')
      .next('span').text(conditionalFlag);

  // Переключение флажков условно-обязательных полей ввода.
  $('input.conditional').next('span').andSelf().hide()
    .end().end()
  .each(function() {
    var $thisInput = $(this);
    var $thisFlag = $thisInput.next('span');
    $thisInput.prev('label').find(':checkbox')
      .attr('checked', false)
      .click(function() {
        if (this.checked) {
          $thisInput.show().addClass('required');
          $thisFlag.show();
          $(this).parent('label').addClass('req-label');
        } else {
          $thisInput.hide().removeClass('required').blur();
          $thisFlag.hide();
          $(this).parent('label').removeClass('req-label');
        }
      });
  });

  // Проверка полей по событию blur.
  $('form :input').blur(function() {
```

```
  $(this).parents('li:first').removeClass('warning')
    .find('span.error-message').remove();

  if ($(this).hasClass('required')) {
    var $listItem = $(this).parents('li:first');
    if (this.value == '') {
      var errorMessage = 'This is a required field';
      if ($(this).is('.conditional')) {
        errorMessage += ', when its related checkbox is checked';
      }
      $('<span></span>')
        .addClass('error-message')
        .text(errorMessage)
        .appendTo($listItem);
      $listItem.addClass('warning');
    }
  }

  if (this.id == 'email') {
    var $listItem = $(this).parents('li:first');
    if ($(this).is(':hidden')) {
      this.value = '';
    }
    if (this.value != '' &&
        !/.+@.+\.[a-zA-Z]{2,4}$/.test(this.value)) {
      var errorMessage = 'Please use proper e-mail format'
                       + ' (e.g. joe@example.com)';
      $('<span></span>')
        .addClass('error-message')
        .text(errorMessage)
        .appendTo($listItem);
      $listItem.addClass('warning');
    }
  }
});

// Проверка полей по событию submit.
$('form').submit(function() {
  $('#submit-message').remove();
  $(':input.required').trigger('blur');
  var numWarnings = $('.warning', this).length;
  if (numWarnings) {
    var fieldList = [];
    $('.warning label').each(function() {
      fieldList.push($(this).text());
    });
    $('<div></div>')
      .attr({
        'id': 'submit-message',
        'class': 'warning'
```

```
            })
            .append('Please correct errors with the following ' +
                    numWarnings + ' fields:<br />')
            .append('&bull; ' + fieldList.join('<br />&bull; '))
                    .insertBefore('#send');
            return false;
        };
    });

    // Флажки
    $('form :checkbox').removeAttr('checked');

    // Флажки с дополнительным флажком (un)check all.
    $('<li></li>')
      .html('<label><input type="checkbox" id="discover-all" />' +
                                    ' <em>check all</em></label>')
      .prependTo('li.discover > ul');
    $('#discover-all').click(function() {
      var $checkboxes = $(this) .parents('ul:first') .find(':checkbox');
      if (this.checked) {
        $(this).next().text(' un-check all');
        $checkboxes.attr('checked', true);
      } else {
        $(this).next().text(' check all');
        $checkboxes.attr('checked', '');
      };
    })
    .parent('label').addClass('checkall');
});
```

Несмотря на существенные усовершенствования формы ввода контактной информации, здесь еще много чего можно было бы сделать. Например, могли бы быть реализованы различные варианты проверки. Для большей гибкости можно было бы использовать модуль расширения, реализующий проверки, сведения о котором можно найти по адресу: *http://plugins.jquery.com/project/validate/*.

Компактные формы

Некоторые формы намного проще, чем форма ввода контактной информации. В действительности на многих сайтах на каждой отдельной странице присутствуют формы, состоящие из одного поля и выполняющие **функцию поиска** по сайту. Типичные атрибуты форм, такие как подписи к полям, кнопка отправки и поясняющий текст, слишком громоздки для таких небольших фрагментов страниц специального назначения. Мы можем использовать библиотеку jQuery, чтобы уменьшить размер формы, сохранив ее функциональность, и даже расширить ее возможности, сделав ее более удобной, чем полностраничный эквивалент.

Текст-заполнитель для полей

Элемент <label> для полей форм является основным компонентом, обеспечивающим доступность веб-сайтов. Каждое поле должно иметь подпись, чтобы устройства чтения с экрана и другие вспомогательные устройства могли идентифицировать его назначение. Даже в разметке HTML элемент label помогает описывать поле:

```
<form id="search" action="search/index.php" method="get">
  <label for="search-text">search the site</label>
  <input type="text" name="search-text" id="search-text" />
</form>
```

Без дополнительного оформления подпись выводится непосредственно перед полем, как показано на рис. 8.13.

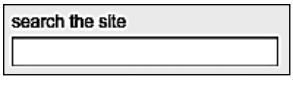


Рис. 8.13. *Без дополнительного оформления подпись выводится непосредственно перед полем*

Подпись не занимает много места, тем не менее в некоторых схемах верстки сайтов даже этот однострочный текст может оказаться слишком большим. Мы могли бы скрыть текст с помощью CSS, но тогда пользователь не смог бы определить назначение поля. Вместо этого мы будем использовать CSS, чтобы поместить подпись поверх поля, если имеется поддержка JavaScript, добавляя класс в форму поиска:

```
$(document).ready(function() {
  var $search = $('#search').addClass('overlabel');
});
```

В единственной строке мы добавили класс к форме поиска и сохранили селектор в переменной, чтобы к нему можно было обратиться позднее. Для оформления метки в таблице стилей определен класс overlabel:

```
.overlabel {
  position: relative;
}
.overlabel label {
  position: absolute;
  top: 6px;
  left: 3px;
  color: #999;
  cursor: text;
}
```

Мало того, что добавленный класс должным образом позиционирует подпись, он еще окрашивает текст подписи в серый цвет, чтобы она воспринималась как текст-заполнитель, как показано на рис. 8.14.

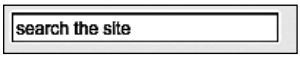


Рис. 8.14. *Вид подписи после применения стилей*

Мы получили замечательный эффект, но здесь есть пара проблем:

1. Текст подписи перекроет текст, который будет вводиться пользователем в поле, как показано на рис. 8.15.
2. Теперь перейти к полю ввода можно будет только клавишей табуляции. Поскольку подпись накрывает поле ввода, пользователь не сможет щелкнуть на поле ввода мышью.

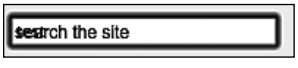


Рис. 8.15. *Текст подписи перекрывает текст в поле ввода*

Во избежание первой проблемы следует скрывать текст подписи в момент, когда поле ввода получает **фокус**, и отображать его снова, когда поле теряет фокус, при условии, что в поле отсутствует текст, введенный пользователем.

Примечание

Сведения о фокусе ввода можно найти в главе 3.

Скрыть подпись в момент получения полем фокуса ввода достаточно просто:

```
$(document).ready(function() {
  var $search = $('#search').addClass('overlabel');
  var $searchInput = $search.find('input');
  var $searchLabel = $search.find('label');

  $searchInput
    .focus(function() {
      $searchLabel.hide();
    })
    .blur(function() {
      if (this.value == '') {
        $searchLabel.show();
      }
    });
});
```

Теперь подпись скрывается, как только пользователь начинает вводить текст в поле, как показано на рис. 8.16.

Рис. 8.16. *Подпись скрывается с началом ввода текста в поле*

Вторая проблема также решается достаточно просто. Мы можем одновременно скрывать текст подписи и предоставлять пользователю доступ к полю, возбуждая в обработчике события `click` подписи событие `focus` элемента ввода:

```
$(document).ready(function() {
  var $search = $('#search').addClass('overlabel');
  var $searchInput = $search.find('input');
  var $searchLabel = $search.find('label');

  $searchInput
    .focus(function() {
      $searchLabel.hide();
    })
    .blur(function() {
      if (this.value == '') {
        $searchLabel.show();
      }
    });

  $searchLabel.click(function() {
    $searchInput.trigger('focus');
  });
});
```

Наконец, нам необходимо обработать ситуацию, когда текст остается в поле ввода при обновлении страницы, подобную ситуации при реализации проверки условно-обязательных полей ввода выше в этой главе. Если поле ввода имеет некоторое значение, подпись не отображается:

```
$(document).ready(function() {
  var $search = $('#search').addClass('overlabel');
  var $searchInput = $search.find('input');
  var $searchLabel = $search.find('label');

  if ($searchInput.val()) {
    $searchLabel.hide();
  }

  $searchInput
    .focus(function() {
      $searchLabel.hide();
    })
    .blur(function() {
      if (this.value == '') {
        $searchLabel.show();
      }
    });

  $searchLabel.click(function() {
    $searchInput.trigger('focus');
```

```
    });
});
```

Одно из преимуществ использования подписи вместо значения по умолчанию, вводимого непосредственно в текстовое поле, заключается в возможности адаптировать любое текстовое поле описанным способом, не беспокоясь о потенциальных конфликтах с реализацией проверки содержимого поля.

Функция автодополнения на основе технологии AJAX

Поле ввода для функции поиска можно еще усовершенствовать, дополнив его возможностью **автодополнения** содержимого. Это даст пользователям возможность вводить начало искомой строки и получать все возможные варианты ее окончания. Так как список предлагаемых вариантов может извлекаться из базы данных, управляющей сайтом, пользователь может знать, что поиск даст результаты, если ввести один из предлагаемых вариантов. Кроме того, если база данных предоставляет варианты в порядке популярности или рядом с каждым вариантом выводится количество результатов, пользователь может выполнять поиск более целенаправленно.

Реализация автодополнения – очень сложная тема, в которой имеется масса тонкостей, обусловленных различными способами взаимодействий с пользователем. Здесь мы создадим работающий пример, но не будем рассматривать такие сложные концепции, как ограничение частоты поступления запросов или дополнение строк из нескольких слов. Для действующих веб-приложений и как отправную точку в поиске более сложных реализаций мы рекомендуем простой виджет автодополнения, входящий в состав модуля расширения **jQuery UI**. Его можно найти по адресу: *http://ui.jquery.com/*.

Основная идея, лежащая в основе процедуры автодополнения, заключается в организации реакции на нажатия клавиш и передаче на сервер запросов AJAX, содержащих текущее содержимое поля ввода. Результаты должны содержать список возможных вариантов дополнения содержимого поля. Сценарий может представлять этот список в виде раскрывающейся области ниже поля ввода.

На стороне сервера

Нам необходимо создать на стороне сервера некоторый программный код, который будет обрабатывать запросы. Действующие реализации обычно опираются на использование баз данных при создании списка возможных вариантов дополнения, однако в данном примере мы будем использовать простой сценарий на языке PHP с результатами, встроенными прямо в него:

```
<?php
  if (strlen($_REQUEST['search-text']) < 1) {
```

```
    print '[]';
    exit;
  }
  $terms = array(
    'access',
    'action',
    // Продолжение списка...
    'xaml',
    'xoops',
  );
  $possibilities = array();
  foreach ($terms as $term) {
    if (strpos($term, strtolower($_REQUEST['search-text']))
        === 0) {
      $possibilities[] = "'". str_replace("'", "\\'", $term)
                         ."'";
    }
  }
  print ('['. implode(', ', $possibilities) .']');
>
```

Этот сценарий сравнивает полученную строку с началом каждого из возможных вариантов и составляет **массив** совпадений в формате **JSON.** Строковые операции, используемые здесь (такие как `str_replace()` и `implode()`), обеспечивают корректный вывод информации в формате **JSON,** чтобы избежать появления ошибок при синтаксическом анализе массива в JavaScript.

В броузере

Теперь можно отправить запрос этому сценарию PHP из программного кода на JavaScript:

```
$(document).ready(function() {
  var $autocomplete = $('<ul class="autocomplete"></ul>')
    .hide()
    .insertAfter('#search-text');

  $('#search-text').keyup(function() {
    $.ajax({
      'url': '../search/autocomplete.php',
      'data': {'search-text': $('#search-text').val()},
      'dataType': 'json',
      'type': 'GET',
      'success': function(data) {
        if (data.length) {
          $autocomplete.empty();
          $.each(data, function(index, term) {
            $('<li></li>').text(term).appendTo($autocomplete);
          });
          $autocomplete.show();
        }
```

```
        }
      });
    });
});
```

В качестве события, по которому будет отправляться запрос AJAX, следует использовать `keyup`, а не `keydown` или `keypress`. Последние два события возникают в процессе нажатия клавиши, то есть до того как символ фактически будет добавлен к значению поля. Если для отправки запросов попробовать задействовать эти события, то список вариантов дополнения будет отставать от значения поля на один символ. Например, после ввода третьего символа в запросе AJAX будут переданы только первые два символа. Используя событие `keyup`, мы можем избежать этой проблемы.

В таблице стилей мы указали, что список вариантов автодополнения будет иметь абсолютное позиционирование, поэтому он сможет накладываться на текст, находящийся ниже. Теперь при вводе строки в поле поиска мы увидим наши варианты дополнения, как показано на рис. 8.17.

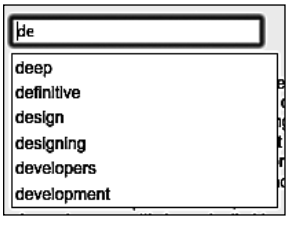


Рис. 8.17. *Список вариантов дополнения*

Для корректного отображения списка вариантов мы учли наличие механизма автодополнения, встроенного в некоторые броузеры. Броузеры часто запоминают, что вводил пользователь в поля формы, и предлагают эти варианты в следующий раз. Предлагаемый броузером список, выведенный поверх нашего собственного списка вариантов, как показано на рис. 8.18, может сбивать с толку.

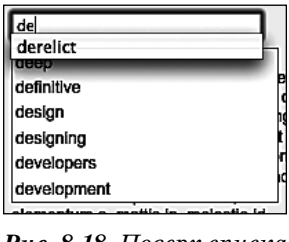


Рис. 8.18. *Поверх списка наших вариантов выводится список, предлагаемый броузером*

К счастью, имеется возможность запретить работу механизма автодополнения, встроенного в броузеры, установив атрибут `autocomplete` поля

формы в значение `off`. Мы могли бы сделать это прямо в разметке HTML, но это привело бы к нарушению принципа прогрессивного улучшения, потому что тем самым мы отключили бы механизм автодополнения в броузере, не предложив свой собственный. Поэтому мы будем устанавливать значение данного атрибута в нашем сценарии:

```
$('#search-text').attr('autocomplete', 'off')
```

Заполнение поля поиска

Наш список вариантов дополнения будет не слишком полезен без возможности вставлять выбранный вариант в поле ввода. Для начала мы дадим пользователю возможность подтверждать выбор щелчком мыши:

```
'success': function(data) {
  if (data.length) {
    $autocomplete.empty();
    $.each(data, function(index, term) {
      $('<li></li>').text(term)
        .appendTo($autocomplete)
        .click(function() {
          $('#search-text').val(term);
          $autocomplete.hide();
        });
    });
    $autocomplete.show();
  }
}
```

Данное изменение выполняет запись текста в поле поиска, когда выполняется щелчок на элементе списка. Кроме того, после щелчка мы скрываем список вариантов, так как работа с ним на этом заканчивается.

Навигация с помощью клавиатуры

Так как при вводе искомого слова пользователь уже работает с клавиатурой, ему будет очень удобно управлять выбором варианта из списка с ее помощью. Чтобы обеспечить такую возможность, нам необходимо следить за вариантом, выбранным в текущий момент. Сначала мы можем добавить вспомогательную функцию, которая будет сохранять индекс элемента списка и воспроизводить необходимые визуальные эффекты, чтобы показать, какой элемент выбран в настоящий момент времени:

```
var selectedItem = null;
var setSelectedItem = function(item) {
  selectedItem = item;
  if (selectedItem === null) {
    $autocomplete.hide();
    return;
  }
```

```
  if (selectedItem < 0) {
    selectedItem = 0;
  }
  if (selectedItem >= $autocomplete.find('li').length) {
    selectedItem = $autocomplete.find('li').length - 1;
  }
  $autocomplete.find('li').removeClass('selected')
    .eq(selectedItem).addClass('selected');
  $autocomplete.show();
};
```

Переменная `selectedItem` будет содержать значение `null`, когда не выбран ни один из элементов списка. Обращаясь к функции `setSelectedItem()`, изменяющей значение переменной, мы можем гарантировать, что список вариантов будет отображаться, только когда имеется выбранный элемент.

Используются две проверки числового значения переменной `selectedItem`, чтобы **поместить** результаты в соответствующие рамки. Без этих проверок переменная `selectedItem` могла бы принимать любые значения, даже отрицательные. Эта функция гарантирует, что текущее значение переменной `selectedItem` всегда будет представлять допустимый индекс в списке вариантов.

Теперь можно дополнить существующую реализацию, добавив в нее вызов новой функции:

```
$('#search-text').attr('autocomplete', 'off').keyup(function() {
  $.ajax({
    'url': '../search/autocomplete.php',
    'data': {'search-text': $('#search-text').val()},
    'dataType': 'json',
    'type': 'GET',
    'success': function(data) {
      if (data.length) {
        $autocomplete.empty();
        $.each(data, function(index, term) {
          $('<li></li>').text(term)
            .appendTo($autocomplete)
            .mouseover(function() {
              setSelectedItem(index);
            })
            .click(function() {
              $('#search-text').val(term);
              $autocomplete.hide();
            });
        });
        setSelectedItem(0);
      }
      else {
        setSelectedItem(null);
```

```
        }
      }
    });
});
```

Эта версия обладает несколькими неоспоримыми преимуществами. Во-первых, список вариантов скрывается при отсутствии вариантов дополнения искомой строки. Во-вторых, мы добавили обработчик события `mouseover`, который выделяет элемент, находящийся под указателем мыши. В-третьих, первый элемент списка выделяется сразу же после отображения списка, как показано на рис. 8.19.

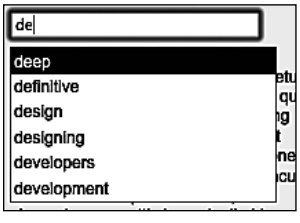


Рис. 8.19. *Первый вариант автоматически выделяется после отображения списка*

Теперь нам следует обеспечить возможность выбора элемента списка с помощью клавиатуры.

Обработка клавиш со стрелками

Для определения нажатой клавиши можно использовать атрибут `keyCode` объекта события. Это позволит нам поджидать нажатия клавиш с кодами 38 и 40, соответствующих клавишам со стрелками вверх и вниз, и реагировать подобающим образом:

```
$('#search-text').attr('autocomplete', 'off').keyup(function(event) {
  if (event.keyCode > 40 || event.keyCode == 8) {
    // Клавиши с кодами 40 и ниже являются специальными клавишами
    // (enter, клавиши со стрелками, escape и другие).
    // Клавиша с кодом 8 - это клавиша забоя (backspace).
    $.ajax({
      'url': '../search/autocomplete.php',
      'data': {'search-text': $('#search-text').val()},
      'dataType': 'json',
      'type': 'GET',
      'success': function(data) {
        if (data.length) {
          $autocomplete.empty();
          $.each(data, function(index, term) {
            $('<li></li>').text(term)
              .appendTo($autocomplete)
              .mouseover(function() {
                setSelectedItem(index);
              })
```

```
                .click(function() {
                  $('#search-text').val(term);
                  $autocomplete.hide();
                });
            });
            setSelectedItem(0);
          }
          else {
            setSelectedItem(null);
          }
        }
      });
    }
    else if (event.keyCode == 38 && selectedItem !== null) {
      // Нажата клавиша со стрелкой вверх.
      setSelectedItem(selectedItem - 1);
      event.preventDefault();
    }
    else if (event.keyCode == 40 && selectedItem !== null) {
      // Нажата клавиша со стрелкой вниз.
      setSelectedItem(selectedItem + 1);
      event.preventDefault();
    }
  });
```

Теперь наш обработчик события `keyup` проверяет значение свойства `keyCode`, определяя код нажатой клавиши, и выполняет соответствующие действия. Запрос AJAX теперь не посылается, если была нажата специальная клавиша, такая как *клавиша со стрелкой* или *escape*. Если было определено нажатие *клавиши со стрелкой* и при этом в текущий момент список вариантов отображается, обработчик изменяет индекс выбранного элемента на 1 в соответствующем направлении. Так как функция `setSelectedItem()` предусматривает **ограничение** значений диапазоном допустимых индексов в списке, нам можно не беспокоиться о том, что пользователь *выйдет за пределы списка*.

Вставка варианта в поле

Далее нам необходимо обработать нажатие на клавишу Enter (или Return на компьютерах Mac). Когда список вариантов отображается на экране, нажатие на клавишу Enter должно приводить к записи текущего выбранного элемента в поле ввода. Поскольку эту операцию нам потребуется вызывать в двух местах, мы выделим процедуру заполнения поля (которая уже была реализована ранее и вызывается в ответ на щелчок мышью) в отдельную функцию:

```
var populateSearchField = function() {
  $('#search-text').val($autocomplete
    .find('li').eq(selectedItem).text());
    setSelectedItem(null);
  };
```

Теперь можно упростить наш обработчик события `click` и вызывать в нем эту функцию. Эту функцию можно вызывать также при обработке события нажатия на клавишу Enter:

```
$('#search-text').keypress(function(event) {
  if (event.keyCode == 13 && selectedItem !== null) {
    // Пользователь нажал клавишу enter.
    populateSearchField();
    event.preventDefault();
  }
});
```

Этот обработчик подключается к событию `keypress`, а не к событию `keyup`, как прежде. Это обусловлено необходимостью предотвратить отправку формы по нажатию на клавишу. Если бы мы дождались события `keyup`, форма к этому моменту была бы уже отправлена.

Удаление списка вариантов

Нам осталось внести в реализацию функции автодополнения последнее усовершенствование. Мы должны обеспечить сокрытие списка, когда пользователь решит выполнить на странице какое-либо другое действие. Прежде всего мы можем среагировать на нажатие клавиши Escape в обработчике события `keyup` и дать пользователю возможность ликвидировать список таким способом:

```
else if (event.keyCode == 27 && selectedItem !== null) {
  // Пользователь нажал клавишу escape.
  setSelectedItem(null);
}
```

Еще более важно скрывать список, когда поле поиска теряет фокус ввода. Первая версия реализации этой особенности выглядит очень просто:

```
$('#search-text').blur(function(event) {
  setSelectedItem(null);
});
```

Однако она обладает нежелательным побочным эффектом. Так как в результате щелчка мышью на элементе списка поле ввода теряет фокус, будет вызван данный обработчик, который скроет список. Это означает, что наш обработчик события `click`, определенный ранее, никогда не будет вызван, в результате чего станет невозможно взаимодействовать со списком с помощью мыши.

Данная проблема не имеет простого решения. Обработчик события `blur` всегда вызывается перед обработчиком события `click`. Обходное решение заключается в том, чтобы скрывать список, когда поле теряет фокус ввода, но откладывать эту операцию на долю секунды:

```
$('#search-text').blur(function(event) {
  setTimeout(function() {
```

```
      setSelectedItem(null);
    }, 250);
  });
```

Это даст обработчику события `click` элемента списка шанс отработать прежде, чем список будет скрыт.

Автодополнение и оперативный поиск

В предыдущем примере мы сфокусировались на функции автодополнения текстового поля, используемой во многих формах. Однако часто предпочтительнее может оказаться альтернативная функция, которая называется **оперативным поиском**. Эта функция фактически производит поиск содержимого по мере ввода пользователем.

Функционально автодополнение и оперативный поиск очень похожи. В обоих случаях нажатие на клавишу инициирует отправку запроса AJAX на сервер, и в этом запросе передается текущее содержимое поля. При получении ответа результаты помещаются в раскрывающийся список, расположенный ниже поля ввода. В случае автодополнения, как мы уже видели, результатами являются возможные варианты искомой строки. В случае оперативного поиска результатами являются фактические страницы, содержащие введенную искомую строку.

С позиции программного кода JavaScript обе эти функции реализуются практически идентично, поэтому мы не будем здесь погружаться в описание деталей. Решение, какую функцию выбрать, в основном зависит от предпочтений; оперативный поиск дает пользователю возможность получить больше информации с меньшими усилиями, но, как правило, эта функция более требовательна к ресурсам.

Окончательная версия

Ниже приводится окончательная версия программного кода, реализующего функцию автодополнения для поля ввода:

```
$(document).ready(function() {
  var $search = $('#search').addClass('overlabel');
  var $searchInput = $search.find('input');
  var $searchLabel = $search.find('label');

  if ($searchInput.val()) {
    $searchLabel.hide();
  }

  $searchInput
    .focus(function() {
      $searchLabel.hide();
    })
    .blur(function() {
      if (this.value == '') {
```

```
      $searchLabel.show();
    }
  });

$searchLabel.click(function() {
  $searchInput.trigger('focus');
});

var $autocomplete = $('<ul class="autocomplete"></ul>')
  .hide()
  .insertAfter('#search-text');
var selectedItem = null;

var setSelectedItem = function(item) {
  selectedItem = item;
  if (selectedItem === null) {
    $autocomplete.hide();
    return;
  }
  if (selectedItem < 0) {
    selectedItem = 0;
  }
  if (selectedItem >= $autocomplete.find('li').length) {
    selectedItem = $autocomplete.find('li').length - 1;
  }
  $autocomplete.find('li').removeClass('selected')
    .eq(selectedItem).addClass('selected');
  $autocomplete.show();
};

var populateSearchField = function() {
  $('#search-text').val($autocomplete
    .find('li').eq(selectedItem).text());
  setSelectedItem(null);
};

$('#search-text')
  .attr('autocomplete', 'off')
  .keyup(function(event) {
    if (event.keyCode > 40 || event.keyCode == 8) {
      // Клавиши с кодами 40 и ниже являются специальными клавишами
      // (enter, клавиши со стрелками, escape и другие).
      // Клавиша с кодом 8 – это клавиша забоя (backspace).
      $.ajax({
        'url': '../search/autocomplete.php',
        'data': {'search-text': $('#search-text').val()},
        'dataType': 'json',
        'type': 'GET',
        'success': function(data) {
          if (data.length) {
            $autocomplete.empty();
```

```
            $.each(data, function(index, term) {
              $('<li></li>').text(term)
                .appendTo($autocomplete)
                .mouseover(function() {
                  setSelectedItem(index);
                }).click(populateSearchField);
            });
            setSelectedItem(0);
          }
          else {
            setSelectedItem(null);
          }
        }
      });
    }
    else if (event.keyCode == 38 && selectedItem !== null) {
      // Пользователь нажал клавишу со стрелкой вверх.
      setSelectedItem(selectedItem - 1);
      event.preventDefault();
    }
    else if (event.keyCode == 40 && selectedItem !== null) {
      // Пользователь нажал клавишу со стрелкой вниз.
      setSelectedItem(selectedItem + 1);
      event.preventDefault();
    }
    else if (event.keyCode == 27 && selectedItem !== null) {
      // Пользователь нажал клавишу escape.
      setSelectedItem(null);
    }
  }).keypress(function(event) {
    if (event.keyCode == 13 && selectedItem !== null) {
      // Пользователь нажал клавишу enter.
      populateSearchField();
      event.preventDefault();
    }
  }).blur(function(event) {
    setTimeout(function() {
      setSelectedItem(null);
    }, 250);
  });
});
```

Работа с числовыми данными в формах

К настоящему моменту мы рассмотрели некоторые особенности, которые могут применяться к текстовым полям ввода формы. Однако достаточно часто в первую очередь формы служат для ввода чисел. Мы можем внести еще несколько усовершенствований формы при работе с числовыми значениями форм.

В нашем книжном интернет-магазине основным кандидатом на представление в виде формы с числовыми данными является корзина для покупок. Нам необходимо предоставить пользователю возможность изменять количество приобретаемых экземпляров, а также требуется возвращать пользователю числовые значения в виде цен и общей стоимости.

Структура таблицы для корзины с покупками

Разметка HTML корзины с покупками описывает таблицу с одной из наиболее подходящих структур, с которыми нам уже приходилось встречаться:

```
<form action="checkout.php" method="post">
  <table id="cart">
    <thead>
      <tr>
        <th class="item">Item</th>
        <th class="quantity">Quantity</th>
        <th class="price">Price</th>
        <th class="cost">Total</th>
      </tr>
    </thead>
    <tfoot>
      <tr class="subtotal">
        <td class="item">Subtotal</td>
        <td class="quantity"></td>
        <td class="price"></td>
        <td class="cost">$152.95</td>
      </tr>
      <tr class="tax">
        <td class="item">Tax</td>
        <td class="quantity"></td>
        <td class="price">6%</td>
        <td class="cost">$9.18</td>
      </tr>
      <tr class="shipping">
        <td class="item">Shipping</td>
        <td class="quantity">5</td>
        <td class="price">$2 per item</td>
        <td class="cost">$10.00</td>
      </tr>
      <tr class="total">
        <td class="item">Total</td>
        <td class="quantity"></td>
        <td class="price"></td>
        <td class="cost">$172.13</td>
      </tr>
      <tr class="actions">
        <td></td>
```

```
      <td>
        <input type="button" name="recalculate"
               value="Recalculate" id="recalculate" />
      </td>
      <td></td>
      <td>
        <input type="submit" name="submit"
               value="Place Order" id="submit" />
      </td>
    </tr>
  </tfoot>
  <tbody>
    <tr>
      <td class="item">
        Building Telephony Systems With Asterisk
      </td>
      <td class="quantity">
        <input type="text" name="quantity-2" value="1"
               id="quantity-2" maxlength="3" />
      </td>
      <td class="price">$26.99</td>
      <td class="cost">$26.99</td>
    </tr>
    <tr>
      <td class="item">
        Smarty PHP Template Programming and Applications
      </td>
      <td class="quantity">
        <input type="text" name="quantity-1" value="2"
               id="quantity-1" maxlength="3" />
      </td>
      <td class="price">$35.99</td>
      <td class="cost">$71.98</td>
    </tr>
    <tr>
      <td class="item">
        Creating your MySQL Database
      </td>
      <td class="quantity">
        <input type="text" name="quantity-3" value="1"
               id="quantity-3" maxlength="3" />
      </td>
      <td class="price">$17.99</td>
      <td class="cost">$17.99</td>
    </tr>
    <tr>
      <td class="item">
        Drupal: Creating Blogs, Forums, Portals, and
        Community Websites
      </td>
```

```
          <td class="quantity">
            <input type="text" name="quantity-4" value="1"
                   id="quantity-4" maxlength="3" />
            </td>
            <td class="price">$35.99</td>
            <td class="cost">$35.99</td>
        </tr>
      </tbody>
    </table>
  </form>
```

В этой таблице присутствует элемент, который редко встречается «в дикой природе», `<tfoot>`. Как и `<thead>`, этот элемент используется для группировки строк таблицы. Обратите внимание: несмотря на то что в разметке этот элемент предшествует определению тела таблицы, он будет отображаться после тела, как показано на рис. 8.20.

Item	Quantity	Price	Total
Building Telephony Systems With Asterisk	1	$26.99	$26.99
Smarty PHP Template Programming and Applications	2	$35.99	$71.98
Creating your MySQL Database	1	$17.99	$17.99
Drupal: Creating Blogs, Forums, Portals, and Community Websites	1	$35.99	$35.99
Subtotal			$152.95
Tax		6%	$9.18
Shipping	5	$2 per item	$10.00
Total			**$172.13**
	Recalculate		Place Order

Рис. 8.20. *Нижний колонтитул таблицы отображается после ее тела*

Такое упорядочение кода разметки, не соответствующее ходу мыслей дизайнера, который думает об отображении таблицы визуальными категориями, полезно для тех, кто испытывает сложности со зрительным восприятием. Когда таблица читается вслух вспомогательными устройствами, нижний колонтитул таблицы будет прочитан перед, возможно, достаточно длинным содержимым, что даст пользователю возможность сразу же получить суммарную информацию о том, что следует ниже.

Кроме того, в каждую ячейку таблицы мы добавили класс CSS, указывающий, в каком столбце таблицы находится эта ячейка. В предыдущей главе мы продемонстрировали некоторые способы поиска ячеек в столбце в ходе определения порядкового номера ячейки в строке. Здесь мы решили упростить программный код JavaScript за счет некоторого усложнения разметки HTML. Благодаря наличию класса, определяющего столбец в каждой ячейке, наши селекторы станут намного проще.

Прежде чем мы продолжим работу с полями формы, добавим стандартный программный код, обеспечивающий чередование окраски строк таблицы:

```
$(document).ready(function() {
  $('#cart tbody tr:nth-child(even)').addClass('alt');
});
```

Здесь, как и ранее, мы выполняем окрашивание только тех строк, которые находятся в теле таблицы, как показано на рис. 8.21.

Item	Quantity	Price	Total
Building Telephony Systems With Asterisk	1	$26.99	$26.99
Smarty PHP Template Programming and Applications	2	$35.99	$71.98
Creating your MySQL Database	1	$17.99	$17.99
Drupal: Creating Blogs, Forums, Portals, and Community Websites	1	$35.99	$35.99
Subtotal			$152.95
Tax		6%	$9.18
Shipping	5	$2 per item	$10.00
Total			$172.13
	Recalculate		Place Order

Рис. 8.21. *Чередование окраски строк в теле таблицы*

Предотвращение возможности ввода нечисловых значений

Совершенствуя форму ввода контактной информации, мы рассматривали некоторые приемы проверки вводимых данных. С помощью JavaScript мы выполняли **проверку ввода**, выясняя, совпадает ли ввод пользователя с тем, что ожидается получить, чтобы можно было обеспечить обратную связь еще до того, как форма будет отправлена на сервер. Теперь мы исследуем похожий прием, который называется **маскированием ввода**.

В ходе проверки ввода выясняется, соответствует ли ввод пользователя некоторым критериям. При маскировании ввода критерии применяются непосредственно в процессе ввода, а недопустимые символы просто отвергаются. В нашей форме, моделирующей корзину с покупками, например, поля ввода могут содержать только числа. Программный код, реализующий маскирование ввода, может просто игнорировать нажатия нецифровых клавиш, когда фокусом ввода обладает одно из таких полей:

```
$('td.quantity input').keypress(function(event) {
  if (event.which && (event.which < 48 ||
      event.which > 57)) {
```

```
      event.preventDefault();
    }
  });
```

Когда мы разрабатывали функцию автодополнения для поля поиска, момент нажатия клавиши определялся нами по событию keyup. Это давало нам возможность проверять содержимое свойства .keyCode объекта события, по которому можно было определить, какая клавиша была нажата. Однако в данном примере мы будем использовать событие keypress. Объект event этого события не имеет свойства .keyCode, но вместо него он предлагает свойство .which, которое содержит фактический символ ASCII, соответствующий нажатой клавише.

Если была нажата символьная клавиша (то есть это не клавиша со стрелкой, не клавиша delete или какая-либо другая специальная клавиша редактирования) и символ, который она представляет, не входит в диапазон кодов ASCII, соответствующих цифрам, то вызывается метод .preventDefault() объекта события. Как уже говорилось ранее, этот метод предотвращает выполнение броузером действий по умолчанию, что в данном случае означает, что символ не будет добавлен в поле. Теперь в поля, описывающие количественные характеристики, можно будет вводить только числа.

Арифметические вычисления

Теперь перейдем к реализации манипуляций фактическими числами, которые пользователь вводит в форму корзины с покупками. На нашей форме присутствует кнопка Recalculate (пересчитать), которая могла бы вызывать передачу формы на сервер, где вычисляются новые суммарные значения, и возврат формы пользователю. Однако для этого необходимо выполнить обновление страницы, которое не является обязательным; все эти действия могут быть выполнены на стороне броузера с помощью jQuery.

Самое простое вычисление в этой форме выполняется для ячейки в строке Shipping (доставка), где отображается общее количество заказанных товаров. Когда пользователь изменяет количество в одной из строк, нам необходимо сложить все введенные значения, чтобы получить новую сумму, и отобразить эту сумму в ячейке:

```
var $quantities = $('td.quantity input');
$quantities.change(function() {
  var totalQuantity = 0;
  $quantities.each(function() {
    var quantity = parseInt(this.value);
    totalQuantity += quantity;
  });
  $('tr.shipping td.quantity').text(String(totalQuantity));
});
```

Для выполнения вычислений мы можем на выбор использовать несколько событий. Мы могли бы использовать событие `keypress` и производить пересчет при каждом нажатии клавиши. Мы могли бы также использовать событие `blur`, которое возбуждается всякий раз, когда поле теряет фокус ввода. Мы можем проявить более консервативное отношение к использованию вычислительных ресурсов и производить пересчет только по событию `change`. При таком способе мы будем производить пересчет, только если пользователь оставит поле со значением, отличающимся от предыдущего.

Общее количество приобретаемых предметов вычисляется с помощью простого цикла `.each()`. Свойство `.value` поля содержит строковое представление значения поля, поэтому для преобразования его в целое число мы используем встроенную функцию `parseInt()`. Этот прием позволяет избежать ситуации, когда оператор сложения интерпретируется как оператор конкатенации строк, поскольку обе эти операции обозначаются одним и тем же символом. Для отображения результатов вычислений мы должны передать методу `.text()` библиотеки jQuery строковое значение, поэтому мы используем функцию `String()`, которая создает новую строку, используя вычисленное нами суммарное количество.

Теперь изменение количества приобретаемых предметов автоматически будет приводить к пересчету общей стоимости, как показано на рис. 8.22.

Item	Quantity	Price	Total
Building Telephony Systems With Asterisk	11	$26.99	$26.99
Smarty PHP Template Programming and Applications	2	$35.99	$71.98
Creating your MySQL Database	1	$17.99	$17.99
Drupal: Creating Blogs, Forums, Portals, and Community Websites	1	$35.99	$35.99
Subtotal			$152.95
Tax		6%	$9.18
Shipping	15	$2 per item	$10.00
Total			**$172.13**
	Recalculate		Place Order

Рис. 8.22. *Изменение количества предметов вызывает пересчет общей стоимости*

Интерпретирование и форматирование денежных значений

Теперь можно перейти к вычислению сумм в правом столбце. Общая стоимость в каждой строке должна вычисляться как произведение количества приобретаемых предметов и цены одного предмета. Поскольку теперь для каждой строки будет выполняться несколько операций, мы начнем с вычисления количества предметов, но уже не по полю, а по строке:

```
$('#cart tbody tr').each(function() {
  var quantity = parseInt($('td.quantity input', this).val());
  totalQuantity += quantity;
});
```

Здесь получается тот же результат, но на этот раз мы получили удобное место, чтобы вставить вычисление общей стоимости для каждой строки:

```
$('td.quantity input').change(function() {
  var totalQuantity = 0;
  $('#cart tbody tr').each(function() {
    var price = parseFloat($('td.price', this).text()
      .replace(/^[^\d.]*/, ''));
    price = isNaN(price) ? 0 : price;
    var quantity =
      parseInt($('td.quantity input', this).val());
    var cost = quantity * price;
    $('td.cost', this).text('$' + cost);
    totalQuantity += quantity;
  });
  $('tr.shipping td.quantity').text(String(totalQuantity));
});
```

Мы извлекаем цену одного предмета из таблицы, используя тот же прием, который был необходим нам ранее для реализации сортировки таблицы по цене. Регулярное выражение сначала удаляет символ валюты, находящийся в начале значения, и затем передает полученную строку функции `parseFloat()`, которая интерпретирует значение как число с плавающей точкой. Поскольку в вычислениях участвует результат, нам необходимо убедиться, что число было найдено, в противном случае установить цену равной `0`. В заключение мы умножаем цену на количество, а результат помещаем в столбец Total (всего), добавив в начало значения символ `$`. Теперь производится вычисление всех суммарных значений, как показано на рис. 8.23.

Item	Quantity	Price	Total
Building Telephony Systems With Asterisk	11	$26.99	$296.89
Smarty PHP Template Programming and Applications	3	$35.99	$107.97
Creating your MySQL Database	1	$17.99	$17.99
Drupal: Creating Blogs, Forums, Portals, and Community Websites	1	$35.99	$35.99
Subtotal			$152.95
Tax		6%	$9.18
Shipping	16	$2 per item	$10.00
Total			**$172.13**
	Recalculate		Place Order

Рис. 8.23. *Изменение количества предметов вызывает пересчет всех суммарных стоимостей*

Обработка дробной части

Несмотря на то, что мы добавили знак доллара в начало каждого суммарного значения, JavaScript даже не подозревает, что имеет дело с денежными величинами. С точки зрения компьютера, это всего лишь числа, которые он отображает соответствующим образом. Следовательно, если общая сумма оканчивается нулем после десятичной точки, этот ноль будет отброшен, как показано на рис. 8.24.

Item	Quantity	Price	Total
Building Telephony Systems With Asterisk	11	$26.99	$296.89
Smarty PHP Template Programming and Applications	30	$35.99	$1079.7
Creating your MySQL Database	1	$17.99	$17.99
Drupal: Creating Blogs, Forums, Portals, and Community Websites	1	$35.99	$35.99
Subtotal			$152.95
Tax		6%	$9.18
Shipping	43	$2 per item	$10.00
Total			**$172.13**
	Recalculate		Place Order

Рис. 8.24. *Завершающий ноль в дробной части был отброшен*

Как видно на снимке с экрана, сумма, которая должна выглядеть как **$1079.70**, отображается как **$1079.7**. Хуже того, ограничения точности представления чисел в JavaScript могут иногда приводить к ошибкам округления, вследствие чего результаты вычислений будут казаться полностью неверными, как показано на рис. 8.25.

Item	Quantity	Price	Total
Building Telephony Systems With Asterisk	11	$26.99	$296.89
Smarty PHP Template Programming and Applications	30	$35.99	$1079.7
Creating your MySQL Database	10	$17.99	$179.89999999999998
Drupal: Creating Blogs, Forums, Portals, and Community Websites	1	$35.99	$35.99
Subtotal			$152.95
Tax		6%	$9.18
Shipping	52	$2 per item	$10.00
Total			**$172.13**
	Recalculate		Place Order

Рис. 8.25. *Ограничения точности вычислений приводят к неверному отображению результатов*

К счастью, обе проблемы решаются достаточно просто. Класс `Number` в JavaScript обладает несколькими методами, позволяющими решать

проблемы подобного рода, и в данном случае мы используем метод .toFixed(). Этот метод принимает в виде аргумента число знаков после десятичной точки и возвращает строку, представляющую число с плавающей точкой, округленное до указанного числа знаков в дробной части:

```
$('#cart tbody tr').each(function() {
  var price = parseFloat($('td.price', this).text()
    .replace(/^[^\d.]*/, ''));
  price = isNaN(price) ? 0 : price;
  var quantity = parseInt($('td.quantity input', this).val());
  var cost = quantity * price;
  $('td.cost', this).text('$' + cost.toFixed(2));
  totalQuantity += quantity;
});
```

Теперь наши суммарные значения выглядят как нормальные денежные значения (см. рис. 8.26).

Item	Quantity	Price	Total
Building Telephony Systems With Asterisk	11	$26.99	$296.89
Smarty PHP Template Programming and Applications	30	$35.99	$1079.70
Creating your MySQL Database	10	$17.99	$179.90
Drupal: Creating Blogs, Forums, Portals, and Community Websites	1	$35.99	$35.99
Subtotal			$152.95
Tax		6%	$9.18
Shipping	52	$2 per item	$10.00
Total			**$172.13**
	Recalculate		Place Order

Рис. 8.26. *Теперь суммарные значения выглядят как денежные величины*

Примечание

После длинной серии арифметических вычислений округление чисел с плавающей точкой может приводить к выявлению накопленных ошибок, которые даже метод .toFixed() не сможет исправить. Самый безопасный способ работы с денежными величинами в крупных приложениях заключается в том, чтобы хранить и оперировать целочисленными значениями в центах; десятичная точка может быть добавлена позднее только для отображения значений.

Прочие вычисления

Остальные вычисления в странице производятся по аналогичному шаблону. Для вычисления промежуточного итога (строка Subtotal) мы можем выполнить суммирование всех сумм в строках во время их вы-

числения и использовать для отображения то же форматирование, что и прежде:

```
$('td.quantity input').change(function() {
  var totalQuantity = 0;
  var totalCost = 0;
  $('#cart tbody tr').each(function() {
    var price = parseFloat($('td.price', this).text()
    price = isNaN(price) ? 0 : price;
    var quantity =
      parseInt($('td.quantity input', this).val());
    var cost = quantity * price;
    $('td.cost', this).text('$' + cost.toFixed(2));
    totalQuantity += quantity;
    totalCost += cost;
  });
  $('tr.shipping td.quantity').text(String(totalQuantity));
  $('tr.subtotal td.cost').text('$' + totalCost.toFixed(2));
});
```

Item	Quantity	Price	Total
Building Telephony Systems With Asterisk	11	$26.99	$296.89
Smarty PHP Template Programming and Applications	1	$35.99	$35.99
Creating your MySQL Database	10	$17.99	$179.90
Drupal: Creating Blogs, Forums, Portals, and Community Websites	1	$35.99	$35.99
Subtotal			$548.77
Tax		6%	$9.18
Shipping	23	$2 per item	$10.00
Total			$172.13
	Recalculate		Place Order

Рис. 8.27. *Теперь выполняется вычисление промежуточного итога (Subtotal)*

Округление значений

Для расчета налога (строка Tax) необходимо разделить указанное число на 100 и затем умножить полученное значение `taxRate` на промежуточный итог. Так как при расчетах сумма налога всегда округляется вверх, мы должны гарантировать, что для отображения и в последующих расчетах будет использоваться корректное значение. Функция `Math.ceil()` в языке JavaScript может округлять число до ближайшего большего целого числа, но так как мы имеем дело с долларами и центами, нам следует быть немного хитрее:

```
var taxRate = parseFloat($('tr.tax td.price').text()) / 100;
var tax = Math.ceil(totalCost * taxRate * 100) / 100;
$('tr.tax td.cost').text('$' + tax.toFixed(2));
totalCost += tax;
```

Сумма налога сначала умножается на 100, чтобы полученное значение оказалось выраженным в центах, а не в долларах. После этого ее можно безопасно округлить с помощью функции `Math.ceil()` и потом разделить на 100, чтобы снова вернуться к исчислению в долларах. В заключение, как и прежде, вызывается метод `.toFixed()`, чтобы воспроизвести корректный результат, как показано на рис. 8.28.

Item	Quantity	Price	Total
Building Telephony Systems With Asterisk	3	$26.99	$80.97
Smarty PHP Template Programming and Applications	1	$35.99	$35.99
Creating your MySQL Database	2	$17.99	$35.98
Drupal: Creating Blogs, Forums, Portals, and Community Websites	1	$35.99	$35.99
Subtotal			$188.93
Tax		6%	$11.34
Shipping	7	$2 per item	$10.00
Total			$172.13
	Recalculate		Place Order

Рис. 8.28. *Теперь выполняется вычисление суммы налога (Tax)*

Последние штрихи

Стоимость доставки (строка Shipping) вычисляется проще, чем сумма налога, поскольку в данном случае не выполняется округление. Стоимость доставки одного предмета просто умножается на количество предметов, и полученное значение используется как общая стоимость доставки:

```
$('tr.shipping td.quantity').text(String(totalQuantity));
var shippingRate = parseFloat($('tr.shipping td.price')
  .text().replace(/^[^\d.]*/, ''));
var shipping = totalQuantity * shippingRate;
$('tr.shipping td.cost').text('$' + shipping.toFixed(2));
totalCost += shipping;
```

В ходе вычислений мы постоянно следили за общей итоговой суммой, поэтому нам осталось всего лишь отформатировать значение переменной `totalCost` для записи в последнюю ячейку:

```
$('tr.total td.cost').text('$' + totalCost.toFixed(2));
```

Таким образом, мы полностью повторили вычисления, которые выполняются на стороне сервера, поэтому теперь мы можем скрыть кнопку Recalculate, как показано на рис. 8.29:

```
$('#recalculate').hide();
```

Данное изменение еще раз повторяет принцип **прогрессивного улучшения:** сначала убедитесь, что страница корректно работает без JavaScript,

а затем используйте библиотеку jQuery, чтобы выполнить те же задачи более элегантным способом, если это возможно.

Item	Quantity	Price	Total
Building Telephony Systems With Asterisk	3	$26.99	$80.97
Smarty PHP Template Programming and Applications	1	$35.99	$35.99
Creating your MySQL Database	2	$17.99	$35.98
Drupal: Creating Blogs, Forums, Portals, and Community Websites	1	$35.99	$35.99
Subtotal			$188.93
Tax		6%	$11.34
Shipping	7	$2 per item	$14.00
Total			$214.27
			Place Order

Рис. 8.29. *Вид формы со скрытой кнопкой Recalculate (пересчитать)*

Удаление элементов

Если покупатель, посетивший наш сайт, передумал приобретать какие-либо книги, уже добавленные им в корзину с покупками, он может изменить значение поля Quantity (количество) для этой покупки, записав в него значение 0. Однако мы можем реализовать более наглядную возможность, добавив в каждый элемент списка кнопку Delete (удалить). Щелчок на кнопке может давать тот же эффект, что и изменение поля Quantity, но визуальная обратная связь может подкрепить факт, что данный предмет не будет куплен.

Для начала нам необходимо добавить новые кнопки. Так как они не могут функционировать без поддержки JavaScript, мы не будем определять их в разметке HTML. Вместо этого мы добавим их в каждую строку таблицы с помощью jQuery:

```
$('<th> </th>')
  .insertAfter('#cart thead th:nth-child(2)');
$('#cart tbody tr').each(function() {
  $deleteButton = $('<img />').attr({
    'width': '16',
    'height': '16',
    'src': '../images/cross.png',
    'alt': 'remove from cart',
    'title': 'remove from cart',
    'class': 'clickable'
  });
  $('<td></td>')
    .insertAfter($('td:nth-child(2)', this))
    .append($deleteButton);
});
```

```
$('<td> </td>')
  .insertAfter('#cart tfoot td:nth-child(2)');
```

Нам нужно добавить в таблицу пустые ячейки в заголовке и в нижнем колонтитуле таблицы, чтобы сохранить правильное расположение столбцов. Кнопки добавляются только в строки тела таблицы, как показано на рис. 8.30.

Item	Quantity		Price	Total
Building Telephony Systems With Asterisk	3	✖	$26.99	$26.99
Smarty PHP Template Programming and Applications	1	✖	$35.99	$71.98
Creating your MySQL Database	2	✖	$17.99	$17.99
Drupal: Creating Blogs, Forums, Portals, and Community Websites	1	✖	$35.99	$35.99
Subtotal				$152.95
Tax			6%	$9.18
Shipping	5		$2 per item	$10.00
Total				**$172.13**
				Place Order

Рис. 8.30. *Кнопки добавляются только в строки тела таблицы*

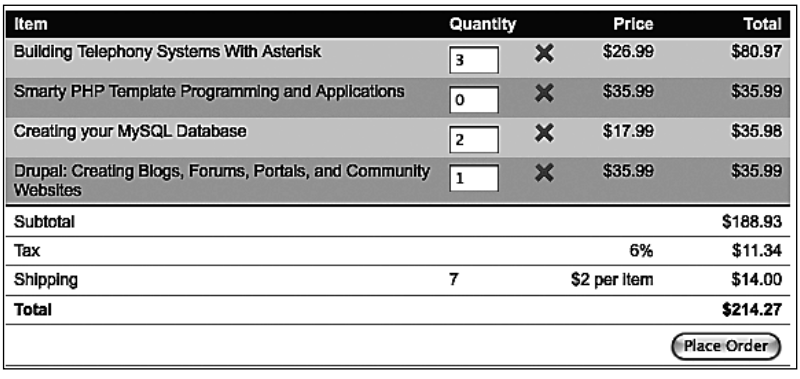
Теперь необходимо реализовать поведение кнопок. Для этого изменим определение кнопок и добавим обработчик события `click`:

```
$deleteButton = $('<img />').attr({
  'width': '16',
  'height': '16',
  'src': '../images/cross.png',
  'alt': 'remove from cart',
  'title': 'remove from cart',
  'class': 'clickable'
}).click(function() {
  $(this).parents('tr').find('td.quantity input')
  .val(0);
});
```

Обработчик события выбирает поле с количеством предметов, находящееся в той же строке, что и кнопка, и устанавливает его значение равным `0`. Значение поля изменилось, но пересчет не был выполнен, как видно из рис. 8.31.

Нам необходимо инициировать пересчет, как если бы значение поля было изменено пользователем:

```
$deleteButton = $('<img />').attr({
  'width': '16',
  'height': '16',
  'src': '../images/cross.png',
  'alt': 'remove from cart',
```

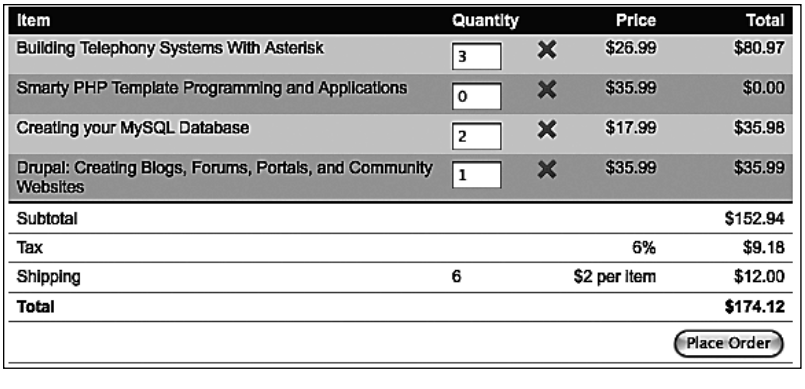


Item	Quantity		Price	Total
Building Telephony Systems With Asterisk	3	✖	$26.99	$80.97
Smarty PHP Template Programming and Applications	0	✖	$35.99	$35.99
Creating your MySQL Database	2	✖	$17.99	$35.98
Drupal: Creating Blogs, Forums, Portals, and Community Websites	1	✖	$35.99	$35.99
Subtotal				$188.93
Tax			6%	$11.34
Shipping	7		$2 per item	$14.00
Total				**$214.27**
				Place Order

Рис. 8.31. *Значение поля изменилось, но пересчет не был выполнен*

```
    'title': 'remove from cart',
    'class': 'clickable'
  }).click(function() {
    $(this).parents('tr').find('td.quantity input')
      .val(0).trigger('change');
  });
```

Теперь щелчок на кнопке вызывает пересчет суммарных значений, как показано на рис. 8.32.



Item	Quantity		Price	Total
Building Telephony Systems With Asterisk	3	✖	$26.99	$80.97
Smarty PHP Template Programming and Applications	0	✖	$35.99	$0.00
Creating your MySQL Database	2	✖	$17.99	$35.98
Drupal: Creating Blogs, Forums, Portals, and Community Websites	1	✖	$35.99	$35.99
Subtotal				$152.94
Tax			6%	$9.18
Shipping	6		$2 per item	$12.00
Total				**$174.12**
				Place Order

Рис. 8.32. *Теперь щелчок на кнопке вызывает пересчет суммарных значений*

Далее требуется обеспечить визуальную обратную связь. Нам необходимо скрыть строку, где только что был выполнен щелчок, чтобы сделать удаление элемента из корзины более наглядным, как показано на рис. 8.33.

```
$deleteButton = $('<img />').attr({
  'width': '16',
  'height': '16',
```

```
    'src': '../images/cross.png',
    'alt': 'remove from cart',
    'title': 'remove from cart',
    'class': 'clickable'
  }).click(function() {
    $(this).parents('tr').find('td.quantity input')
      .val(0).trigger('change')
      .end().hide();
  });
```

Item	Quantity		Price	Total
Building Telephony Systems With Asterisk	3	✖	$26.99	$80.97
Creating your MySQL Database	2	✖	$17.99	$35.98
Drupal: Creating Blogs, Forums, Portals, and Community Websites	1	✖	$35.99	$35.99
Subtotal				$152.94
Tax			6%	$9.18
Shipping	6		$2 per item	$12.00
Total				**$174.12**
				Place Order

Рис. 8.33. *Строка с удаленным элементом скрывается из таблицы*

Несмотря на то, что строка была скрыта, поле все еще присутствует в форме. Это означает, что оно будет отправлено вместе с остальной частью формы, после чего элемент будет удален уже на стороне сервера.

Удаление строки из таблицы повлекло нарушение чередования окраски строк. Чтобы исправить этот недостаток, мы переместим реализацию окраски строк в отдельную функцию, чтобы потом ее можно было вызвать еще раз. В то же время нам необходимо модифицировать программный код, чтобы он игнорировал скрытые строки. К сожалению, даже если отфильтровать все скрытые строки, мы все равно не сможем использовать селектор `:nth-child(even)`, потому что он будет применять класс `alt` ко всем видимым строкам, которые являются «четными» потомками своих родителей. Посмотрим, что произойдет, если задействовать следующую модификацию программного кода к таблице из четырех строк, в которой вторая строка скрыта:

```
$('#cart tbody tr').removeClass('alt')
  .filter(':visible:nth-child(even)').addClass('alt');
```

Мы получим следующую разметку HTML [сокращено]:

```
<tr> . . . </tr>
<tr style="display: none"> . . . </tr>
<tr> . . . </tr>
<tr class="alt"> . . . </tr>
```

В таблице присутствуют три видимые строки, но только четвертая получила класс `alt`. То что нам нужно – это выражение селектора `:visible:odd`, так как он отберет каждую вторую строку уже после исключения скрытых строк из выбора (при этом будет автоматически учтен переход от селектора, начинающего отсчет с 1, к селектору, начинающему отсчет с 0). Как уже отмечалось в главе 2, использование селектора `:odd` или `:even` может приводить к неожиданным результатам при наличии более одного элемента `<tbody>`, но в этом отношении у нас все в порядке. После замены выражения селектора наш программный код выглядит, как показано ниже:

```
var stripe = function() {
  $('#cart tbody tr').removeClass('alt')
    .filter(':visible:odd').addClass('alt');
};
stripe();
```

Теперь эту функцию можно вызвать еще раз после удаления строки:

```
$deleteButton = $('<img />').attr({
  'width': '16',
  'height': '16',
  'src': '../images/cross.png',
  'alt': 'remove from cart',
  'title': 'remove from cart',
  'class': 'clickable'
}).click(function() {
  $(this).parents('tr').find('td.quantity input')
    .val(0).trigger('change')
    .end().hide();
  stripe();
});
```

Теперь удаленная строка скрывается бесследно, как показано на рис. 8.34.

Item	Quantity		Price	Total
Building Telephony Systems With Asterisk	3	✖	$26.99	$80.97
Creating your MySQL Database	2	✖	$17.99	$35.98
Drupal: Creating Blogs, Forums, Portals, and Community Websites	1	✖	$35.99	$35.99
Subtotal				$152.94
Tax			6%	$9.18
Shipping	6		$2 per item	$12.00
Total				**$174.12**
				Place Order

Рис. 8.34. *Теперь удаленная строка скрывается бесследно*

Этим завершается реализация еще одного усовершенствования с использованием библиотеки jQuery, которое полностью независимо от программного кода на стороне сервера. Для сервера вполне достаточно, чтобы пользователь ввел в поле значение 0, но для пользователя это отчетливо идентифицируемая операция удаления, отличная от операции изменения количества.

Изменение информации с адресом доставки

На странице, имитирующей корзину с покупками, кроме всего прочего присутствует форма ввода адреса доставки. Фактически на момент загрузки страницы это вообще не форма, и без поддержки JavaScript она остается небольшим прямоугольником (рис. 8.35), спрятанным с правой стороны области содержимого страницы и содержащим ссылку на страницу, где пользователь сможет отредактировать адрес доставки.

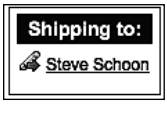


Рис. 8.35. *Ссылка на страницу, где пользователь сможет отредактировать адрес доставки*

Но при наличии поддержки JavaScript и с использованием всей мощи библиотеки jQuery, имеющейся в нашем распоряжении, мы можем превратить такую маленькую ссылку в полноценную форму. Мы сделаем это, запросив форму из страницы PHP. Обычно данные, которыми заполняется форма, сохраняются в базе данных, но исключительно в демонстрационных целях мы будем хранить некоторые статические данные непосредственно в массиве PHP.

Чтобы воспроизвести форму внутри области с заголовком Shipping to (доставить по адресу), мы использовали метод `$.get()` внутри обработчика события `.click()`:

```
$(document).ready(function() {
  $('#shipping-name').click(function() {
    $.get('shipping.php', function(data) {
      $('#shipping-name').remove();
      $(data).hide().appendTo('#shipping').slideDown();
    });
    return false;
  });
});
```

При вызове методом `$.get()`, прежде чем вывести большую часть страницы, сценарий PHP (`shipping.php`) на стороне сервера проверяет наличие переменной `$_SERVER['HTTP_X_REQUESTED_WITH']`, и в случае ее отсутствия возвращает только часть страницы – саму форму.

Внутри функции обратного вызова, которая передается методу `$.get()`, мы удаляем имя, на котором только что был выполнен щелчок, и на его место вставляем форму с данными, полученную от сценария `shipping.php`. Чтобы предотвратить действие по умолчанию (загрузку страницы, указанной в атрибуте `href`), предусмотренное для обработки щелчка на ссылке, мы добавили инструкцию `return false`. После этого область с заголовком Shipping to превращается в форму, доступную для редактирования, как показано на рис. 8.36.

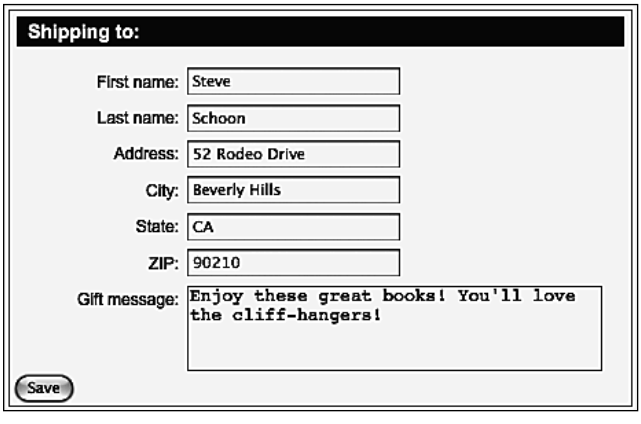


Рис. 8.36. *Область с заголовком Shipping to (доставить по адресу) превращается в форму*

Теперь пользователь может отредактировать адрес доставки, не покидая страницу.

Следующий шаг – отправка формы и доставка отредактированных данных обратно на сервер с помощью jQuery. Мы начнем с сериализации данных, присутствующих в форме, и сохранения результата в переменной `postData`. Затем мы отправим данные на сервер, еще раз воспользовавшись сценарием `shipping.php`:

```
$(document).ready(function() {
  $('shipping form').submit(function() {
    var postData = $(this).serialize();
    $.post('shipping.php', postData);
    return false;
  };
});
```

Примечание

Модуль расширения Form к библиотеке jQuery реализует более надежный метод `.serialize()`. Этот модуль расширения, рекомендуемый в большинстве ситуаций, когда отправка формы выполняется с применением технологии AJAX, можно найти по адресу: *http://www.malsup.com/jquery/form/*.

В данный момент есть смысл удалить форму из страницы и вернуть область с заголовком Shipping to в начальное состояние. Это можно реализовать внутри функции обратного вызова, которая передается только что использованному методу `$.post()`:

```
$(document).ready(function() {
  $('#shipping form').submit(function() {
    var postData = $(this).serialize();
    $.post('shipping.php', postData, function(data) {
      $('#shipping form').remove();
      $(data).appendTo('#shipping');
    });
    return false;
  };
});
```

Но она не работает! Дело в том, что в нашей реализации мы подключили обработчик события `.submit()` к форме Shipping to сразу же после загрузки дерева DOM, но форма не будет включена в дерево DOM, пока пользователь не щелкнет на имени в ссылке Shipping to. Событие не может быть подключено к тому, чего не существует.

Для преодоления этой проблемы можно поместить программный код, создающий форму, в функцию с именем `editShipping()`, а реализацию отправки формы и ее удаления – в функцию с именем `saveShipping()`. Тогда мы сможем использовать функцию `saveShipping()` в функции обратного вызова метода `$.get()` уже после того, как форма будет создана. Точно так же можно выполнить привязку функции `editShipping()` сразу же после создания дерева DOM и после повторного создания ссылки Edit shipping (редактировать адрес доставки) в функции обратного вызова метода `$.post()`:

```
$(document).ready(function() {
  var editShipping = function() {
    $.get('shipping.php', function(data) {
      $('#shipping-name').remove();
      $(data).hide().appendTo('#shipping').slideDown();
      $('#shipping form').submit(saveShipping);
    });
    return false;
  };
  var saveShipping = function() {
    var postData = $('#shipping :input').serialize();
    $.post('shipping.php', postData, function(data) {
      $('#shipping form').remove();
      $(data).appendTo('#shipping');
      $('#shipping-name').click(editShipping);
    });
    return false;
  };
  $('#shipping-name').click(editShipping);
});
```

Этот программный код образован с применением шаблона циклических ссылок, в котором две функции позволяют друг другу повторно подключать себя в качестве соответствующих обработчиков событий.

Окончательная версия

Вся реализация корзины с покупками занимает не более 80 строк, что достаточно мало, если принять во внимание достигнутую в ней функциональность, причем это достижение становится особенно внушительным с учетом стиля оформления программного кода для большей удобочитаемости. Благодаря возможности составления цепочек из вызовов методов библиотеки jQuery многие строки можно было бы объединить, если бы нас беспокоило число строк. Как бы то ни было, ниже приводится окончательная версия программного кода для страницы корзины с покупками, которая завершает главу, рассказывающую о формах:

```
$(document).ready(function() {
  var stripe = function() {
    $('#cart tbody tr').removeClass('alt')
      .filter(':visible:odd').addClass('alt');
  };
  stripe();

  $('#recalculate').hide();

  $('.quantity input').keypress(function(event) {
    if (event.which && (event.which < 48 || event.which > 57)) {
      event.preventDefault();
    }
  }).change(function() {
    var totalQuantity = 0;
    var totalCost = 0;
    $('#cart tbody tr').each(function() {
      var price = parseFloat($('.price', this)
                             .text().replace(/^[^\d.]*/, ''));
      price = isNaN(price) ? 0 : price;
      var quantity = parseInt($('.quantity input', this).val(), 10);
      var cost = quantity * price;
      $('.cost', this).text('$' + cost.toFixed(2));
      totalQuantity += quantity;
      totalCost += cost;
    });
    $('.subtotal .cost').text('$' + totalCost.toFixed(2));
    var taxRate = parseFloat($('.tax .price').text()) / 100;
    var tax = Math.ceil(totalCost * taxRate * 100) / 100;
    $('.tax .cost').text('$' + tax.toFixed(2));
    totalCost += tax;
    $('.shipping .quantity').text(String(totalQuantity));
    var shippingRate = parseFloat($('.shipping .price')
                                  .text().replace(/^[^\d.]*/, ''));
    var shipping = totalQuantity * shippingRate;
```

```
    $('.shipping .cost').text('$' + shipping.toFixed(2));
    totalCost += shipping;
    $('.total .cost').text('$' + totalCost.toFixed(2));
  });

  $('<th> </th>')
    .insertAfter('#cart thead th:nth-child(2)');
  $('#cart tbody tr').each(function() {
    $deleteButton = $('<img />').attr({
      'width': '16',
      'height': '16',
      'src': '../images/cross.png',
      'alt': 'remove from cart',
      'title': 'remove from cart',
      'class': 'clickable'
    }).click(function() {
      $(this).parents('tr').find('td.quantity input')
        .val(0).trigger('change')
        .end().hide();
      stripe();
    });
    $('<td></td>')
      .insertAfter($('td:nth-child(2)', this))
      .append($deleteButton);
  });
  $('<td> </td>')
    .insertAfter('#cart tfoot td:nth-child(2)');
});

$(document).ready(function() {
  var editShipping = function() {
    $.get('shipping.php', function(data) {
      $('#shipping-name').remove();
      $(data).hide().appendTo('#shipping').slideDown();
      $('#shipping form').submit(saveShipping);
    });
    return false;
  };
  var saveShipping = function() {
    var postData = $(this).serialize();
    $.post('shipping.php', postData, function(data) {
      $('#shipping form').remove();
      $(data).appendTo('#shipping');
      $('#shipping-name').click(editShipping);
    });
    return false;
  };
  $('#shipping-name').click(editShipping);
});
```

В заключение

В этой главе мы исследовали способы улучшения внешнего вида форм и функциональных возможностей обычных элементов форм HTML. Мы узнали, как реализовывать дополнительное оформление формы, не затрагивая семантику оригинальной разметки, как скрывать поля формы при соблюдении определенных условий, как отображать поля, основываясь на значениях других полей, и как выполнять проверку содержимого полей перед отправкой на сервер и в процессе ввода данных. Мы охватили такие особенности, как функция автодополнения текстовых полей с применением технологии AJAX, ограничение возможности ввода в поле определенным набором символов и выполнение арифметических операций над числовыми значениями полей. Мы также узнали, как выполнять отправку формы с помощью технологии AJAX, не вызывая обновление всей страницы.

Элементы форм часто играют роль связующего звена, скрепляющего интерактивные сайты. С помощью библиотеки jQuery мы легко можем сделать заполнение форм более приятным занятием для пользователей и сохранить при этом их практичность и гибкость.

9

Прокрутка и перемещение

Мы уже видели несколько способов сокрытия информации, когда в ней нет необходимости, и отображения ее по требованию, например путем использования раскрывающихся элементов с информацией. Однако иногда бывает желательно перемещать информацию в или из области видимости с применением еще более интересных эффектов. Такого рода эффекты известны под многими именами: **карусель**, **круговое вращение**, **прокрутка** и **вертушка**. Общее в них то, что они позволяют производить быструю смену кадров с фрагментами данных привлекательным и выразительным способом.

В третьей и заключительной главе из раздела «как это делается» мы исследуем эти дополнительные анимационные эффекты, объединив их с приемами использования AJAX и тонкостями CSS, чтобы с их общей помощью уметь производить нужное впечатление.

В данной главе мы рассмотрим два больших примера: прокрутка заголовков и карусель изображений. Эти примеры помогут нам понять, как:

- Анимировать изменение положения элемента
- Интерпретировать документы XML
- Подготавливать улучшенные эффекты оформления, используя полупрозрачность
- Извлекать информацию из других доменов
- Создавать элемент пользовательского интерфейса с горизонтальной прокруткой
- Накладывать слои друг на друга
- Изменять масштаб изображения

Прокрутка заголовков

В качестве первого примера реализации **прокрутки** возьмем ленту новостей; мы будем прокручивать ее, отображая по одному заголовку вме-

сте с фрагментом статьи за один раз. Статьи будут выдвигаться в область просмотра, приостанавливаться для прочтения и затем скользить вверх за пределы страницы, как если бы в странице проворачивалась бесконечная лента с информацией.

Подготовка страницы

Реализовать эту особенность в самом простом виде не составит труда. Но, как мы увидим ниже, чтобы довести ее до приемлемого уровня, потребуется проявить некоторое искусство.

Как обычно, начнем с фрагмента HTML. Мы поместим ленту новостей во врезку:

```
<h3>Recent News</h3>
<div id="news-feed">
  <a href="news/index.html">News Releases</a>
</div>
```

К данному моменту в области содержимого ленты новостей находится единственная ссылка на основную страницу с новостями, как показано на рис. 9.1.

Рис. 9.1. *Первоначальный вид врезки с лентой новостей*

Это результат нашего следования принципу **постепенной деградации** на случай, если поддержка JavaScript окажется отключенной у пользователя. Содержимое страницы останется работоспособным, но будет ссылаться на фактический источник **рассылки RSS**.

Стили CSS для данного элемента `<div>` весьма важны, поскольку определяют не только, какая часть каждого элемента ленты новостей будет видна, но и где на странице будет отображаться врезка с лентой. Вместе с правилами, определяющими оформление отдельных элементов ленты новостей, таблица CSS выглядит так, как показано ниже:

```
#news-feed {
  position: relative;
```

```
    height: 200px;
    width: 17em;
    overflow: hidden;
  }
  .headline {
    position: absolute;
    height: 200px;
    top: 210px;
    overflow: hidden;
  }
```

Обратите внимание, что высота и отдельного элемента новостей (представленного классом `headline`), и его контейнера равна `200px`. Кроме того, так как элементы новостей имеют абсолютное позиционирование относительно `#news-feed`, мы сможем поместить верхний край элемента на уровне нижнего края контейнера. А благодаря тому, что мы установили свойство `overflow` для `#news-feed` в значение `hidden`, изначально новости невидимы.

Установка свойства `position` в классе `headline` в значение `absolute` обусловлена и другой причиной: для получения возможности перемещать какой-либо элемент страницы необходимо, чтобы класс CSS для этого элемента имел значение `absolute` или `relative` в свойстве `position`, а не значение по умолчанию `static`.

Теперь, когда у нас имеются разметка HTML и таблица стилей CSS, можно попробовать вставить элементы новостей из рассылки RSS. Для начала обернем программный код методом `.each()`, который будет действовать как своеобразная условная инструкция `if` и содержать программный код в частном **пространстве имен**:

```
$(document).ready(function() {
  $('#news-feed').each(function() {
    var $container = $(this);
    $container.empty();
  });
});
```

Обычно метод `.each()` используется для выполнения итераций по возможно большому набору элементов. Однако здесь наш селектор `#news-feed` отбирает элементы по значению атрибута `id`, поэтому могут существовать только два результата работы этого селектора. Фабричная функция может создать объект jQuery, соответствующий единственному уникальному элементу со значением `news-feed` в атрибуте `id`, или создать пустой объект jQuery, не обнаружив в странице элементов с указанным значением атрибута `id`. Метод `.each()` обеспечивает выполнение содержащегося в нем программного кода, если и только если объект jQuery не является пустым.

В самом начале цикла `.each()` производится очистка контейнера ленты новостей с целью подготовки его к наполнению новым содержимым, как показано на рис. 9.2.

Рис. 9.2. *Вид контейнера после очистки*

Получение рассылки

Для взаимодействия с сервером и получения рассылки мы будем использовать метод `$.get()`, одну из множества функций поддержки технологии AJAX, имеющихся в библиотеке jQuery. Метод `$.get()`, как мы уже видели ранее, позволяет оперировать данными, полученными из удаленных источников, с помощью **обработчика случая успешного получения данных.** Содержимое рассылки передается этому обработчику в формате XML. Благодаря чему мы получаем возможность обрабатывать данные с применением механизма селекторов библиотеки jQuery.

```
$(document).ready(function() {
  $('#news-feed').each(function() {
    var $container = $(this);
    $container.empty();
    $.get('news/feed.xml', function(data) {
      $('rss item', data).each(function() {
        // Здесь выполняются действия с элементами новостей.
      });
    });
  });
});
```

Примечание

За дополнительной информацией о методе `$.get()` и других методах поддержки AJAX обращайтесь к главе 6.

Теперь нам необходимо объединить части элементов в пригодный для использования фрагмент разметки HTML. Для обхода всех элементов рассылки и создания ссылок на новости мы снова можем использовать метод `.each()`:

```
$(document).ready(function() {
  $('#news-feed').each(function() {
    var $container = $(this);
    $container.empty();
    $.get('news/feed.xml', function(data) {
      $('rss item', data).each(function() {
        var $link = $('<a></a>')
          .attr('href', $('link', this).text())
          .text($('title', this).text());
        var $headline = $('<h4></h4>').append($link);

        $('<div></div>')
          .append($headline)
          .appendTo($container);
      });
    });
  });
});
```

Мы извлекаем текстовое содержимое элементов `<title>` и `<link>` и конструируем из них элементы `<a>`. Каждая полученная ссылка обертывается элементом `<h4>`. Каждый заголовок новости помещается в элемент `<div id="news-feed">`, но, чтобы было проще следить за ходом разработки, мы пока не применяем класс `headline` к элементам, помещаемым в элемент `<div>`, как показано на рис. 9.3.

Рис. 9.3. *Вид контейнера с заголовками новостей*

Помимо заголовков нам требуется отобразить некоторую сопроводительную информацию о каждой статье. Мы будем извлекать дату публикации и краткое описание и отображать их.

```
$(document).ready(function() {
  $('#news-feed').each(function() {
    var $container = $(this);
    $container.empty();
```

```
    $.get('news/feed.xml', function(data) {
      $('rss item', data).each(function() {
        var $link = $('<a></a>')
          .attr('href', $('link', this).text())
          .text($('title', this).text());
        var $headline = $('<h4></h4>').append($link);

        var pubDate = new Date(
          $('pubDate', this).text());
        var pubMonth = pubDate.getMonth() + 1;
        var pubDay = pubDate.getDate();
        var pubYear = pubDate.getFullYear();
        var $publication = $('<div></div>')
          .addClass('publication-date')
          .text(pubMonth + '/' + pubDay + '/'
                + pubYear);
        var $summary = $('<div></div>')
          .addClass('summary')
          .html($('description', this).text());

        $('<div></div>')
          .append($headline, $publication, $summary)
          .appendTo($container);
      });
    });
  });
});
```

Даты в рассылке RSS представлены в формате RFC 822 и включают информацию о дате, времени и часовом поясе (например, `Sun, 28 Sep 2008 18:01:55 +0000`). Этот формат не очень удобно воспринимать на глаз, поэтому для воспроизведения дат в более компактном виде (например, `9/28/2008`) мы использовали объект `Date`, встроенный в JavaScript.

Краткое описание легко извлекается и форматируется. Тем не менее следует отметить, что в рассылке в тексте описания могут присутствовать некоторые **объекты HTML**. Чтобы избежать их автоматического экранирования библиотекой jQuery, для вставки описания в страницу необходимо использовать метод `.html()`, а не `.text()`.

После того как эти новые элементы будут созданы, мы вставляем их в документ с помощью метода `.append()`. Обратите внимание, что здесь используется новая для нас особенность данного метода: когда метод получает несколько аргументов, он добавляет всех их последовательно.

Как видно на рис. 9.4, теперь для каждой новости выводятся заголовок, дата публикации и краткое описание. Все, что осталось сделать, это добавить класс `headline` с помощью вызова метода `.addClass('headline')` (что приведет к сокрытию новостей вследствие имеющегося определения класса CSS), после чего мы будем готовы приступить к воспроизведению анимационного эффекта.

Рис. 9.4. *Вид контейнера с заголовками новостей, датами публикации и краткими описаниями*

Подготовка к выполнению прокрутки

Так как со временем видимость элементов новостей будет изменяться, нам требуется способ, с помощью которого легко можно было бы узнать, какие элементы видимы и где они находятся. Сначала мы определим значения двух переменных, одна из них будет хранить индекс текущего видимого заголовка, а вторая – индекс заголовка, который только что был прокручен за пределы области видимости. Изначально обе переменные имеют значение `0`.

```
var currentHeadline = 0, oldHeadline = 0;
```

Затем следует побеспокоиться о начальном позиционировании заголовков. Напомним, что в таблице стилей мы уже установили свойство `top` заголовков в значение, на 10 пикселов превышающее значение свойства `height` контейнера, а так как свойство `overflow` контейнера имеет значение `hidden`, изначально заголовки не отображаются. Если сохранить начальное значение для данного свойства в переменной, позднее нам проще будет перемещать заголовки в эту позицию.

```
var hiddenPosition = $container.height() + 10;
```

Кроме того, нам необходимо, чтобы первый заголовок отображался сразу же после первоначальной загрузки страницы. Для этого мы можем установить свойство `top` равным `0`.

```
$('div.headline').eq(currentHeadline).css ('top', 0);
```

Теперь область прокрутки на странице находится в требуемом начальном состоянии, как показано на рис. 9.5.

Наконец, мы сохраним общее количество заголовков для последующего использования и определим переменную, где будет храниться предельное время ожидания, для оформления паузы между циклами прокрутки.

Рис. 9.5. *Область прокрутки находится в требуемом начальном состоянии*

```
var headlineCount = $('div.headline').length;
var pause;
```

В данный момент нет необходимости определять значение переменной `pause` – оно будет устанавливаться всякий раз перед сменой заголовка. Однако мы всегда должны объявлять **локальные переменные** с помощью ключевого слова `var`, чтобы избежать опасности конфликта имен с **глобальными переменными**.

Функция прокрутки заголовков

Теперь все готово к воспроизведению эффекта прокрутки заголовков, дат и кратких описаний. Мы определим функцию, которая будет выполнять это действие, чтобы его можно было повторять, когда потребуется.

Сначала позаботимся об обновлении значений переменных, используемых для отслеживания активного заголовка. Оператор **деления по модулю** (`%`) позволит легко организовать циклический обход списка заголовков. Всякий раз, когда будет вызываться наша функция, мы можем добавлять 1 к значению переменной `currentHeadline`, затем вычислять остаток от деления полученного результата на значение переменной `headlineCount` и тем самым определять допустимый порядковый номер следующего заголовка.

Примечание

Напомним, что такой же прием использовался для реализации чередования окраски строк таблицы в главе 7.

Нам также необходимо обновить значение переменной `oldHeadline`, чтобы было проще манипулировать заголовком, покидающим область видимости.

```
var headlineRotate = function() {
  currentHeadline = (oldHeadline + 1) % headlineCount;

  // Изменение позиции заголовка производится здесь.

  oldHeadline = currentHeadline;
};
```

Теперь заполним промежуток программным кодом, выполняющим перемещение заголовков. Сначала добавим анимационный эффект, который перемещает старый заголовок за пределы области видимости. Затем вставим другой анимационный эффект, который будет передвигать в пределы области видимости новый заголовок.

```
var headlineRotate = function() {
  currentHeadline = (oldHeadline + 1) % headlineCount;
  $('div.headline').eq(oldHeadline).animate(
    {top: -hiddenPosition}, 'slow', function() {
      $(this).css('top', hiddenPosition);
    });
  $('div.headline').eq(currentHeadline).animate(
    {top: 0}, 'slow', function() {
      pause = setTimeout(headlineRotate, 5000);
    });
  oldHeadline = currentHeadline;
};
```

В обоих случаях анимационный эффект основан на изменении свойства `top` элементов новостей. Напомним, что элементы скрыты, потому что их свойство `top` имеет значение `hiddenPosition` (которое больше высоты контейнера). Когда это свойство приобретает значение `0`, элемент оказывается в области видимости. При дальнейшем изменении его до значения `-hiddenPosition` элемент снова выходит за область видимости.

Примечание

В главе 4 говорилось, что свойство `top` в таблицах CSS влияет на позицию элемента, только если свойство `position` элемента имеет значение `absolute` или `relative`.

Кроме того, в обоих случаях мы определили функции обратного вызова, которые вызываются по завершении воспроизведения анимационного эффекта. Когда старый заголовок полностью покидает область видимости, его свойство `top` устанавливается в значение `hiddenPosition`, что обеспечивает его подготовку к отображению позднее. Когда завершится воспроизведение анимационного эффекта для нового заголовка, нам необходимо запустить следующий цикл смены заголовков. Делается это с помощью функции JavaScript `setTimeout()`, которая регистрирует указанную функцию для вызова через заданный интервал време-

ни. В данном случае мы предусматриваем повторный вызов функции `headlineRotate()` через пять секунд (5000 миллисекунд).

Сейчас у нас имеется замкнутый цикл операций; как только завершается один цикл воспроизведения анимационных эффектов, тут же выполняется подготовка к запуску следующего цикла. Нам осталось только вызвать функцию в самый первый раз; мы сделаем это с помощью еще одного вызова функции `setTimeout()`, благодаря которому первый цикл воспроизведения эффектов начнется через 5 секунд после получения рассылки RSS. Теперь у нас имеется работоспособная функция прокрутки.

```
$(document).ready(function() {
  $('#news-feed').each(function() {
    var $container = $(this);
    $container.empty();
    $.get('news/feed.xml', function(data) {
      $('rss item', data).each(function() {
        var $link = $('<a></a>')
          .attr('href', $('link', this).text())
          .text($('title', this).text());
        var $headline = $('<h4></h4>').append($link);

        var pubDate = new Date($('pubDate', this).text());
        var pubMonth = pubDate.getMonth() + 1;
        var pubDay = pubDate.getDate();
        var pubYear = pubDate.getFullYear();
        var $publication = $('<div></div>')
          .addClass('publication-date')
          .text(pubMonth + '/' + pubDay + '/' + pubYear);

        var $summary = $('<div></div>')
          .addClass('summary')
          .html($('description', this).text());

        $('<div></div>')
          .addClass('headline')
          .append($headline, $publication, $summary)
          .appendTo($container);
      });

      var currentHeadline = 0, oldHeadline = 0;
      var hiddenPosition = $container.height() + 10;
      $('div.headline').eq(currentHeadline).css('top', 0);
      var headlineCount = $('div.headline').length;
      var pause;

      var headlineRotate = function() {
        currentHeadline = (oldHeadline + 1) % headlineCount;
        $('div.headline').eq(oldHeadline).animate(
```

```
            {top: -hiddenPosition}, 'slow', function() {
              $(this).css('top', hiddenPosition);
            });
          $('div.headline').eq(currentHeadline).animate(
            {top: 0}, 'slow', function() {
              pause = setTimeout(headlineRotate, 5000);
            });
          oldHeadline = currentHeadline;
        };
        pause = setTimeout(headlineRotate, 5000);
      });
    });
});
```

В ходе воспроизведения анимационного эффекта можно наблюдать, как один заголовок постепенно исчезает за верхним краем, а следующий за ним появляется из-за нижнего края, как показано на рис. 9.6.

***Рис. 9.6.** Один заголовок постепенно исчезает за верхним краем, а другой появляется из-за нижнего края*

Приостановка при наведении указателя мыши

Теперь прокрутка заголовков полностью работоспособна, но есть одна проблема, связанная с удобством использования, которую нам следует решить, – описание новости может покинуть область видимости еще до того, как пользователь успеет щелкнуть на ссылке. В таком случае пользователь будет вынужден ждать, пока функция прокрутки пройдет по всем заголовкам, прежде чем даст ему второй шанс. Мы можем уменьшить вероятность появления этой проблемы, приостановив прокрутку, если пользователь навел указатель мыши на заголовок.

```
$container.hover(function() {
  clearTimeout(pause);
}, function() {
  pause = setTimeout(headlineRotate, 250);
});
```

Когда указатель мыши попадет в область заголовка, первый обработчик события `.hover()` вызовет функцию JavaScript `clearTimeout()`. Это остановит работу таймера и предотвратит вызов функции `headlineRotate()`. Когда указатель мыши покинет область заголовка, второй обработчик `.hover()` перезапустит таймер и тем самым обеспечит вызов функции `headlineRotate()` с 250-миллисекундной задержкой.

В большинстве случаев этот программный код будет прекрасно справляться с решением проблемы. Однако, если пользователь несколько раз быстро переместит указатель мыши то в пределы элемента `<div>`, то из него, может возникнуть очень нежелательный эффект. Одновременно будут прокручиваться сразу несколько заголовков, накладываясь друг на друга, как показано на рис. 9.7.

Рис. 9.7. *Одновременно прокручиваются несколько заголовков, накладывающихся друг на друга*

К сожалению, нам придется выполнить серьезную хирургическую операцию, чтобы удалить эту опухоль. Добавим еще одну переменную перед определением функции `headlineRotate()`:

```
var rotateInProgress = false;
```

Теперь в самой первой строке функции мы можем проверить, не выполняется ли в настоящий момент цикл прокрутки, и продолжить работу, только если переменная `rotateInProgress` имеет значение `false`. Для этого мы обернем тело функции условной инструкцией `if`. Внутри условной инструкции мы тут же установим эту переменную в значение `true`, а в функции обратного вызова второго метода `.animate()` опять вернем ей значение `false`.

```
var headlineRotate = function() {
  if (!rotateInProgress) {
    rotateInProgress = true;
    currentHeadline = (oldHeadline + 1) % headlineCount;
    $('div.headline').eq(oldHeadline).animate(
```

```
        {top: -hiddenPosition}, 'slow', function() {
          $(this).css('top', hiddenPosition);
        });
      $('div.headline').eq(currentHeadline).animate(
        {top: 0}, 'slow', function() {
          rotateInProgress = false;
          pause = setTimeout(headlineRotate, 5000);
        });
      oldHeadline = currentHeadline;
    }
  };
```

Эти несколько дополнительных строк существенно улучшают функцию прокрутки заголовков. Быстрые повторяющиеся движения указателя мыши больше не будут приводить к наложению заголовков друг на друга. И все же такое поведение пользователя по-прежнему может доставлять нам одну досадную проблему: ритм работы функции нарушается, и она подряд выполняет два-три цикла воспроизведения анимационных эффектов вместо того, чтобы растянуть их в пятисекундные интервалы.

Проблема заключается в том, что после таких манипуляций с мышью одновременно могут появиться сразу несколько активных таймеров, прежде чем существующий таймер завершит отсчет. Поэтому нам необходимо предусмотреть еще один защитный механизм и воспользоваться переменной `pause` как флагом, показывающим, что можно запланировать воспроизведение очередного анимационного эффекта. Для этого мы будем устанавливать данную переменную в значение `false` после принудительной остановки таймера и по завершении отсчета. Теперь мы сможем проверять значение переменной, чтобы убедиться в отсутствии активных таймеров, прежде чем запускать новый.

```
var headlineRotate = function() {
  if (!rotateInProgress) {
    rotateInProgress = true;
    pause = false;
    currentHeadline = (oldHeadline + 1) % headlineCount;
    $('div.headline').eq(oldHeadline).animate(
      {top: -hiddenPosition}, 'slow', function() {
        $(this).css('top', hiddenPosition);
      });
    $('div.headline').eq(currentHeadline).animate(
      {top: 0}, 'slow', function() {
        rotateInProgress = false;
        if (!pause) {
          pause = setTimeout(headlineRotate, 5000);
        }
      });
    oldHeadline = currentHeadline;
  }
};
if (!pause) {
```

```
    pause = setTimeout(headlineRotate, 5000);
  }
  $container.hover(function() {
    clearTimeout(pause);
    pause = false;
  }, function() {
    if (!pause) {
      pause = setTimeout(headlineRotate, 250);
    }
  });
```

Наконец, наша реализация прокрутки может противостоять любым попыткам пользователя помешать ее работе.

Получение рассылки из другого домена

В нашем примере в качестве службы рассылки новостей мы использовали локальный файл, но нам может потребоваться получить рассылку с совершенно постороннего сайта. Как мы уже видели в главе 6, запросы AJAX, как правило, не могут выполняться к сайтам, отличным от того, откуда была получена просматриваемая страница. В той главе мы обсуждали применение формата данных JSONP как средство преодоления этого ограничения. Однако здесь мы будем полагать, что у нас отсутствует возможность изменять исходные данные, поэтому нам требуется найти иное решение.

Чтобы обеспечить возможность получения файла средствами AJAX, мы задействуем некоторый программный код на стороне сервера, который будет играть роль **посредника** для запросов, вследствие чего JavaScript будет полагать, что файл XML находится на нашем сервере, несмотря на то, что фактически он располагается в другом месте. Мы напишем короткий сценарий PHP, который будет перемещать содержимое рассылки новостей на наш сервер, и поставлять эти данные по запросу сценария jQuery. Этот сценарий PHP, который мы назовем `feed.php`, может вызываться точно так же, как ранее извлекался файл `feed.xml`:

```
$.get('news/feed.php', function(data) {
  // ...
});
```

Внутри файла `feed.php` мы получим содержимое рассылки новостей от удаленного сайта, а затем выведем его.

```
<?php
  header('Content-Type: text/xml');
  print file_get_contents('http://jquery.com/blog/feed');
?>
```

Обратите внимание, что здесь нам пришлось явно определить тип содержимого страницы как `text/xml`, чтобы библиотека jQuery смогла извлечь его и обработать, как если бы это был обычный статичный документ XML.

Примечание

Некоторые поставщики услуг веб-хостинга из соображений безопасности могут запрещать использование функции `file_get_contents()` языка PHP для получения удаленных файлов. В таких случаях может оказаться доступным альтернативное решение, которое заключается в использовании библиотеки **cURL**. Дополнительную информацию об этой библиотеке можно найти по адресу: *http://wiki.dreamhost.com/CURL*.

Добавление индикатора загрузки

Перемещение удаленного файла может потребовать некоторого времени, продолжительность которого зависит от ряда факторов, поэтому мы должны сообщить пользователю о том, что идет процесс загрузки. Для этого до отправки запроса AJAX мы добавим в страницу изображение **индикатора загрузки**.

```
var $loadingIndicator = $('<img/>')
  .attr({
    'src': 'images/loading.gif',
    'alt': 'Loading. Please wait.'
  })
  .addClass('news-wait')
  .appendTo($container);
```

Затем в первой строке функции обратного вызова, передаваемой методу `$.get()`, которая обрабатывает успешное завершение запроса, мы сможем удалить изображение из страницы простой командой:

```
$loadingIndicator.remove();
```

Теперь, если после начальной загрузки страницы при получении заголовков новостей возникнет задержка, вместо пустой области мы увидим изображение индикатора загрузки, как показано на рис. 9.8.

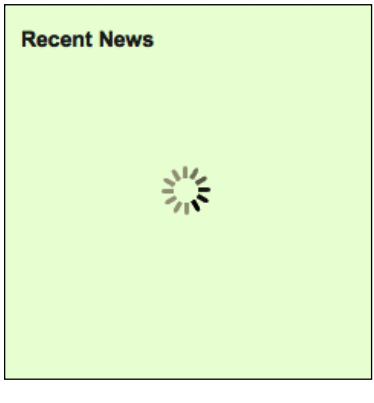


Рис. 9.8. *Если при получении заголовков новостей возникнет задержка, мы увидим индикатор загрузки*

Это анимированное изображение GIF, которое будет вращаться в окне броузера, показывая, что выполняется некоторое действие.

Примечание

Мы легко можем создавать новые анимированные изображения GIF для применения в качестве индикаторов загрузки AJAX, воспользовавшись службой *http://ajaxload.info/*.

Эффект изменения прозрачности по высоте

Прежде чем мы покинем пример прокрутки заголовков, внесем последний штрих, заставив текст заголовка появляться с эффектом увеличения непрозрачности, начиная с нижней границы контейнера. **Эффект изменения прозрачности по высоте** проявляется как непрозрачный текст в верхней части области просмотра и прозрачный – в нижней.

Однако единственный текстовый элемент не может одновременно иметь несколько градаций прозрачности. Чтобы сымитировать данный эффект, нам необходимо покрыть область проявления эффекта последовательностью элементов, каждый из которых будет иметь свою степень прозрачности. В качестве таких элементов мы будем использовать элементы `<div>` с некоторыми общими свойствами стиля, которые мы объявили в нашей таблице стилей:

```
.fade-slice {
  position: absolute;
  width: 20em;
  height: 2px;
  background: #efd;
  z-index: 3;
}
```

Значения свойств `width` и `background-color` этих элементов совпадают со значениями тех же свойств вмещающего их элемента `<div id="news-feed">`. Это позволит обмануть глаз пользователя, который будет полагать, что текст постепенно проявляется, а не покрыт другими элементами.

Теперь можно создать элементы `<div class="fade-slice">`. Чтобы создать нужное число элементов, сначала определим высоту всей области проявления эффекта в пикселах. В данном случае мы выбрали 25 процентов высоты элемента `<div id="news-feed">`. Для выполнения итераций по высоте этой области мы будем использовать цикл `for`; в ходе каждой итерации будет создаваться новый покрывающий элемент для каждого сегмента области эффекта высотой в два пиксела:

```
$(document).ready(function() {
  $('#news-feed').each(function() {
    var $container = $(this);
    $container.empty();
    var fadeHeight = $container.height() / 4;
```

```
    for (var yPos = 0; yPos < fadeHeight; yPos += 2) {
      $('<div></div>')
        .addClass('fade-slice')
        .appendTo($container);
    }
  });
});
```

Теперь у нас имеется 25 элементов (по одному на каждый сегмент, высотой два пиксела, в области проявления эффекта высотой 50 пикселов), но все они пока находятся в верхней части контейнера. Чтобы наш трюк сработал, необходимо каждый элемент поместить на свое место и придать ему свою степень непрозрачности. Мы можем использовать переменную цикла `yPos`, чтобы арифметическим путем определить значения для свойств `opacity` и `top` каждого элемента:

```
$(document).ready(function() {
  $('#news-feed').each(function() {
    var $container = $(this);
    $container.empty();
    var fadeHeight = $container.height() / 4;
    for (var yPos = 0; yPos < fadeHeight; yPos += 2) {
      $('<div></div>').css({
        opacity: yPos / fadeHeight,
        top: $container.height() - fadeHeight + yPos
      }).addClass('fade-slice').appendTo($container);
    }
  });
});
```

Для кого-то может оказаться непросто понять эти вычисления, поэтому мы расположили получающиеся результаты в виде таблицы (табл. 9.1). Значения свойства `opacity` увеличиваются от значения, соответствующего полной прозрачности, до значения, соответствующего полной непрозрачности, по мере изменения значения свойства `top` от значения, соответствующего верхней границе области проявления эффекта (150), до значения, равного высоте контейнера.

Таблица 9.1. Зависимость значений свойства opacity от координаты top

yPos	opacity	top
0	0	150
2	0.04	152
4	0.08	154
6	0.12	156
8	0.16	158
…	…	…

40	0.80	190
42	0.84	192
44	0.88	194
46	0.92	196
48	0.96	198

Обратите внимание: так как координата верхней границы последнего элемента `<div class="fade-slice">` равна 198, а его высота равна двум пикселам, он аккуратно накроет самые нижние два пиксела элемента-контейнера `<div>`, имеющего высоту 200 пикселов.

Теперь после добавления этого программного кода текст заголовка новости будет красиво проявляться в нижней части контейнера, как показано на рис. 9.9.

Рис. 9.9. *Текст красиво проявляется в нижней части контейнера*

Окончательная версия

На этом мы закончили разработку первого примера реализации прокрутки. Теперь элементы новостей извлекаются с удаленного сервера, форматируются и отображаются с воспроизведением анимационного эффекта и применением привлекательного оформления:

```
$(document).ready(function() {
  $('#news-feed').each(function() {
    var $container = $(this);
    $container.empty();

    var fadeHeight = $container.height() / 4;
    for (var yPos = 0; yPos < fadeHeight; yPos += 2) {
      $('<div></div>').css({
```

```
    opacity: yPos / fadeHeight,
    top: $container.height() - fadeHeight + yPos
  }).addClass('fade-slice').appendTo($container);
}

var $loadingIndicator = $('<img/>')
  .attr({
    'src': 'images/loading.gif',
    'alt': 'Loading. Please wait.'
  })
  .addClass('news-wait')
  .appendTo($container);

$.get('news/feed.php', function(data) {
  $loadingIndicator.remove();
  $('rss item', data).each(function() {
    var $link = $('<a></a>')
      .attr('href', $('link', this).text())
      .text($('title', this).text());
    var $headline = $('<h4></h4>').append($link);

    var pubDate = new Date($('pubDate', this).text());
    var pubMonth = pubDate.getMonth() + 1;
    var pubDay = pubDate.getDate();
    var pubYear = pubDate.getFullYear();
    var $publication = $('<div></div>')
      .addClass('publication-date')
      .text(pubMonth + '/' + pubDay + '/' + pubYear);

    var $summary = $('<div></div>')
      .addClass('summary')
      .html($('description', this).text());

    $('<div></div>')
      .addClass('headline')
      .append($headline, $publication, $summary)
      .appendTo($container);
  });

  var currentHeadline = 0, oldHeadline = 0;
  var hiddenPosition = $container.height() + 10;
  $('div.headline').eq(currentHeadline).css('top', 0);
  var headlineCount = $('div.headline').length;
  var pause;
  var rotateInProgress = false;

  var headlineRotate = function() {
    if (!rotateInProgress) {
      rotateInProgress = true;
      pause = false;
```

```
            currentHeadline = (oldHeadline + 1) % headlineCount;
            $('div.headline').eq(oldHeadline).animate(
              {top: -hiddenPosition}, 'slow', function() {
                $(this).css('top', hiddenPosition);
              });
            $('div.headline').eq(currentHeadline).animate(
              {top: 0}, 'slow', function() {
                rotateInProgress = false;
                if (!pause) {
                  pause = setTimeout(headlineRotate, 5000);
                }
              });
            oldHeadline = currentHeadline;
          }
        };
        if (!pause) {
          pause = setTimeout(headlineRotate, 5000);
        }
        $container.hover(function() {
          clearTimeout(pause);
          pause = false;
        }, function() {
          if (!pause) {
            pause = setTimeout(headlineRotate, 250);
          }
        });
      });
    });
});
```

Карусель изображений

В качестве другого примера прокручивания содержимого страницы мы реализуем **галерею изображений** для главной страницы книжного интернет-магазина. Галерея будет рекламировать книги, выставленные на продажу, со ссылками на увеличенное изображение первой страницы обложки для каждой из них. В отличие от предыдущего примера, где прокручивание заголовков новостей производилось программно, здесь мы будем использовать средства библиотеки jQuery, чтобы реализовать прокрутку изображений поперек страницы в ответ на действия пользователя.

Примечание

Альтернативный механизм прокрутки набора изображений реализован в модуле расширения **jCarousel** для библиотеки jQuery. Кроме того, существует модуль **SerialScroll**, обладающий высокой гибкостью и позволяющий прокручивать содержимое любого типа. Эти модули не идентичны результатам, которых мы достигнем здесь, тем не менее они позволяют воспроизводить высококачественные эффекты прокручивания с помощью небольшого объема

программного кода. Дополнительная информация об использовании модулей расширения приводится в главе 10.

Подготовка страницы

Как всегда, начнем с создания разметки HTML и правил CSS, чтобы пользователи, не обладающие поддержкой JavaScript, могли получать информацию в привлекательном и функциональном представлении:

```
<div id="featured-books">
  <div class="covers">
    <a href="images/covers/large/1847190871.jpg"
      title="Community Server Quickly">
      <img src="images/covers/medium/1847190871.jpg"
        width="120" height="148"
        alt="Community Server Quickly" />
      <span class="price">$35.99</span>
    </a>
    <a href="images/covers/large/1847190901.jpg"
      title="Deep Inside osCommerce: The Cookbook">
      <img src="images/covers/medium/1847190901.jpg"
        width="120" height="148"
        alt="Deep Inside osCommerce: The Cookbook" />
      <span class="price">$44.99</span>
    </a>
    <a href="images/covers/large/1847190979.jpg"
      title="Learn OpenOffice.org Spreadsheet Macro
        Programming: OOoBasic and Calc automation">
      <img src="images/covers/medium/1847190979.jpg"
        width="120" height="148"
        alt="Learn OpenOffice.org Spreadsheet Macro
        Programming: OOoBasic and Calc automation" />
      <span class="price">$35.99</span>
    </a>
    <a href="images/covers/large/1847190987.jpg"
      title="Microsoft AJAX C# Essentials: Building
        Responsive ASP.NET 2.0 Applications">
      <img src="images/covers/medium/1847190987.jpg"
        width="120" height="148"
        alt="Microsoft AJAX C# Essentials: Building
        Responsive ASP.NET 2.0 Applications" />
      <span class="price">$31.99</span>
    </a>
    <a href="images/covers/large/1847191002.jpg"
      title="Google Web Toolkit GWT Java AJAX Programming">
      <img src="images/covers/medium/1847191002.jpg"
        width="120" height="148"
        alt="Google Web Toolkit GWT Java AJAX Programming" />
      <span class="price">$40.49</span>
    </a>
```

```
      <a href="images/covers/large/1847192386.jpg"
        title="Building Websites with Joomla! 1.5 Beta 1">
        <img src="images/covers/medium/1847192386.jpg"
          width="120" height="148"
          alt="Building Websites with Joomla! 1.5 Beta 1" />
        <span class="price">$40.49</span>
      </a>
    </div>
  </div>
```

Каждое изображение заключено в якорный тег, ссылающийся на увеличенную версию изображения обложки книги. У нас также имеется цена каждой книги; она пока скрыта, но мы будем отображать ее с помощью JavaScript в соответствующие моменты времени.

Для экономии места на главной странице мы будем отображать одновременно только три изображения обложек книг. Без применения JavaScript этого можно добиться установкой свойства `overflow` контейнера в значение `scroll` и соответствующей ширины:

```
#featured-books {
  position: relative;
  background: #ddd;
  width: 440px;
  height: 186px;
  overflow: scroll;
  margin: 1em auto;
  padding: 0;
  text-align: center;
  z-index: 2;
}
#featured-books .covers {
  position: relative;
  width: 840px;
  z-index: 1;
}
#featured-books a {
  float: left;
  margin: 10px;
  height: 146px;
}
#featured-books .price {
  display: none;
}
```

Данные стили стоят того, чтобы рассмотреть их поближе. Самый внешний элемент должен иметь значение свойства `z-index` большее, чем у вложенного в него; это позволит в броузере Internet Explorer скрыть части внутреннего элемента, которые выступают за пределы содержащего его контейнера. Мы установили ширину внешнего элемента равной `440px`, что позволяет вместить три изображения с учетом поля ши-

риной 10px, окружающего каждое из них, и дополнительных 20px для полосы прокрутки.

С применением этих стилей уже можно просматривать изображения, используя стандартную полосу прокрутки, как показано на рис. 9.10.

Рис. 9.10. *Первоначальный вид области просмотра изображений с полосой прокрутки*

Исправление стилей с помощью JavaScript

Теперь, когда у нас имеется галерея изображений, пригодная к использованию без поддержки JavaScript, нам необходимо изменить некоторые детали. Полоса прокрутки станет ненужной, когда мы реализуем свой собственный механизм прокрутки, а автоматическое размещение изображений, реализованное с помощью свойства float, будет мешать, когда нам потребуется позиционировать изображения в ходе воспроизведения анимационного эффекта. Поэтому в первую очередь мы переопределим некоторые стили:

```
$(document).ready(function() {
  var spacing = 140;
  $('#featured-books').css({
    'width': spacing * 3,
    'height': '166px',
    'overflow': 'hidden'
  }).find('.covers a').css({
    'float': 'none',
    'position': 'absolute',
    'left': 1000
  });
  var $covers = $('#featured-books .covers a');
  $covers.eq(0).css('left', 0);
  $covers.eq(1).css('left', spacing);
  $covers.eq(2).css('left', spacing * 2);
});
```

Переменная spacing пригодится нам во многих вычислениях. Она хранит ширину одного изображения обложки плюс отступы с обеих сторон. Так как теперь отпала необходимость в выделении пространства

для полосы прокрутки, свойство `width` элемента-контейнера может быть установлено в значение, в точности соответствующее суммарной ширине трех изображений. В действительности же мы изменили значение свойства `overflow` на `hidden`, и прощай полоса прокрутки.

Все изображения обложки позиционируются абсолютно и изначально имеют левую координату со значением `1000`. Благодаря этому они оказываются за пределами области видимости. Затем мы по одному перемещаем первые три изображения в соответствующие позиции. Переменная `$covers`, хранящая все якорные элементы, также пригодится нам позднее.

Теперь видимы первые три изображения, при этом механизм прокрутки отсутствует, как показано на рис. 9.11.

Рис. 9.11. *Видимы первые три изображения, при этом механизм прокрутки отсутствует*

Прокрутка изображений щелчком мыши

Теперь нам необходимо добавить программный код, обрабатывающий щелчок мышью на любом из крайних изображений и выполняющий переупорядочение изображений, если это необходимо. Когда щелчок выполняется на левом изображении, это означает, что пользователь желает увидеть изображения, находящиеся левее, то есть мы должны сдвинуть изображения *вправо*. Аналогично, когда щелчок выполняется по правому изображению, мы должны сдвинуть изображения *влево*. Нам требуется, чтобы карусель **вращалась по кругу**, поэтому при смещении изображения за левую границу оно будет добавляться справа. Для начала мы будем просто изменять координаты изображений, не добавляя анимационный эффект.

```
$(document).ready(function() {
  var spacing = 140;
  $('#featured-books').css({
    'width': spacing * 3,
    'height': '166px',
    'overflow': 'hidden'
  }).find('.covers a').css({
    'float': 'none',
```

```
    'position': 'absolute',
    'left': 1000
  });

  var setUpCovers = function() {
    var $covers = $('#featured-books .covers a');

    $covers.unbind('click');

    // Левое изображение - сдвинуть вправо
    // (чтобы просмотреть изображения слева).
    $covers.eq(0)
      .css('left', 0)
      .click(function(event) {
        $covers.eq(2).css('left', 1000);
        $covers.eq($covers.length - 1)
          .prependTo('#featured-books .covers');
        setUpCovers();

        event.preventDefault();
      });

    // Правое изображение - сдвинуть влево
    // (чтобы просмотреть изображения справа).
    $covers.eq(2)
      .css('left', spacing * 2)
      .click(function(event) {
        $covers.eq(0).css('left', 1000);
        $covers.eq(0)
          .appendTo('#featured-books .covers');
        setUpCovers();

        event.preventDefault();
      });

    // Центральное изображение.
    $covers.eq(1)
      .css('left', spacing);
  };
  setUpCovers();
});
```

Новая функция `setUpCovers()` содержит написанный нами ранее программный код, выполняющий позиционирование изображений. Поместив его в функцию, мы получили возможность повторно выполнять позиционирование изображений после переупорядочения элементов; это очень важно, в чем вы вскоре убедитесь.

Всего в нашем примере имеется шесть изображений (на которые программный код JavaScript будет ссылаться по порядковым номерам от 0 до 5). Видимыми являются изображения с порядковыми номерами

0, 1 и 2. Когда выполняется щелчок на изображении №0, мы должны выполнить сдвиг всех изображений на одну позицию вправо. Сначала мы сдвигаем за пределы области просмотра изображение №2 (вызовом `.css('left', 1000)`), потому что после сдвига оно не должно оставаться видимым. Затем мы перемещаем последнее изображение (№5) в начало очереди (используя вызов `.prependTo()`). Это вызывает переупорядочение всех изображений, поэтому, когда снова будет вызвана функция `setUpCovers()`, элемент, который прежде был под №5, теперь будет элементом под №0, №0 станет №1, и №1 станет №2. Следовательно, имеющегося в данной функции программного кода, выполняющего позиционирование, вполне достаточно, чтобы переместить изображения в новые местоположения.

Рис. 9.12. *В случае щелчка на изображении №2 выполняется обратный процесс*

В случае щелчка на изображении №2 выполняется обратный процесс. На этот раз уже №0 покидает область видимости и перемещается в конец очереди. В результате №1 смещается в позицию с №0, №2 – в позицию с №1 и №3 – в позицию с №2, как показано на рис. 9.12.

Нам следует позаботиться о двух вещах, чтобы уберечь пользователя от аномалий взаимодействия:

1. Внутри обработчика события `click` нам нужно вызвать метод `.preventDefault()`, так как в разметке HTML изображения ссылаются на увеличенные версии. Без этого вызова броузер будет выполнять переход по ссылке, и мы никогда не сможем наблюдать эффект прокручивания.
2. В начале функции `setUpCovers()` необходимо отключить все обработчики события `click`, в противном случае по мере вращения карусели к одному и тому же изображению могут оказаться подключенными сразу несколько обработчиков.

Добавление анимационного эффекта сдвига

Поскольку смена изображений происходит мгновенно, пользователь может не сразу понять, что произошло в результате щелчка на изображении, и подумать, что имела место просто смена изображений, а не сдвиг. Чтобы смягчить данный недостаток, мы можем добавить анима-

ционный эффект, который будет вызывать плавное перемещение изображений на новые места вместо мгновенного появления их на новых позициях. Для этого требуется внести изменения в функцию `setUpCovers()`**:**

```
var setUpCovers = function() {
  var $covers = $('#featured-books .covers a');

  $covers.unbind('click');

  // Левое изображение – сдвинуть вправо
  // (чтобы просмотреть изображения слева).
  $covers.eq(0)
    .css('left', 0)
    .click(function(event) {
      $covers.eq(0).animate({'left': spacing}, 'fast');
      $covers.eq(1).animate({'left': spacing * 2}, 'fast');
      $covers.eq(2).animate({'left': spacing * 3}, 'fast');
      $covers.eq($covers.length - 1)
        .css('left', -spacing)
        .animate({'left': 0}, 'fast', function() {
          $(this).prependTo('#featured-books .covers');
          setUpCovers();
        });
      event.preventDefault();
    });

  // Правое изображение сдвинуть влево
  // (чтобы просмотреть изображения справа).
  $covers.eq(2)
    .css('left', spacing * 2)
    .click(function(event) {
      $covers.eq(0)
        .animate({'left': -spacing}, 'fast', function() {
          $(this).appendTo('#featured-books .covers');
          setUpCovers();
        });
      $covers.eq(1).animate({'left': 0}, 'fast');
      $covers.eq(2).animate({'left': spacing}, 'fast');
      $covers.eq(3)
        .css('left', spacing * 3)
        .animate({'left': spacing * 2}, 'fast');

      event.preventDefault();
    });

  // Центральное изображение.
  $covers.eq(1)
    .css('left', spacing);
};
```

В случае щелчка на левом изображении мы можем переместить все три видимых изображения вправо на ширину одного изображения (используя переменную `spacing`, которая была определена ранее). В этой части все просто, но нам также необходимо переместить в область просмотра новое изображение. Для этого мы берем изображение в конце очереди и сначала устанавливаем его в позицию, находящуюся непосредственно за левым краем экрана (`-spacing`). Затем перемещаем его в область просмотра вместе с другими элементами, как показано на рис. 9.13.

Рис. 9.13. *Перемещение изображений с воспроизведением анимационного эффекта*

Несмотря на то, что анимационный эффект выполняет перемещение, нам по-прежнему необходимо изменить порядок следования изображений повторным вызовом функции `setUpCovers()`. Если этого не сделать, следующий щелчок мышью будет обрабатываться неправильно. Так как функция `setUpCovers()` изменяет позиции изображений на экране, нам следует отложить вызов функции, дождавшись завершения воспроизведения анимационного эффекта, поэтому мы поместили этот вызов в функцию обратного вызова для анимационного эффекта.

Щелчок на самом правом изображении выполняет похожую последовательность анимационных эффектов, только в обратном порядке. На этот раз за границу области просмотра смещается самое левое изображение, и оно должно быть перемещено в конец очереди до того, как по завершении анимационного эффекта будет вызвана функция `setUpCovers()`. Напротив, изображение справа должно переместиться в новую позицию (`spacing * 3`) до того, как начнется воспроизведение анимационного эффекта.

Отображение ярлыков, обозначающих действие

Теперь наша карусель изображений вращается гладко, но мы не даем пользователю никаких подсказок, что щелчок на изображении может вызывать их прокручивание. Мы можем помочь пользователю, отображая соответствующие ярлыки при наведении указателя мыши на изображение.

В данном случае мы поместим ярлыки поверх существующих изображений. Используя свойство `opacity`, мы по-прежнему сможем видеть

изображения обложек при отображении ярлыков. Чтобы не слишком загораживать изображения обложек, мы будем использовать простые черно-белые ярлыки, как показано на рис. 9.14.

***Рис. 9.14.** Ярлыки, обозначающие действия*

Нам потребуется три ярлыка: по одному для левого и правого изображений, которые будут выбираться пользователем для выполнения прокрутки, и один для среднего изображения, на котором пользователь сможет щелкнуть, чтобы увидеть увеличенную версию изображения. Мы можем создать элементы HTML, которые будут ссылаться на ярлыки, и сохранить их в переменных для последующего использования:

```
var $leftRollover = $('<img/>')
  .attr('src', 'images/left.gif')
  .addClass('control')
  .css('opacity', 0.6)
  .css('display', 'none');
var $rightRollover = $('<img/>')
  .attr('src', 'images/right.gif')
  .addClass('control')
  .css('opacity', 0.6)
  .css('display', 'none');
var $enlargeRollover = $('<img/>')
  .attr('src', 'images/enlarge.gif')
  .addClass('control')
  .css('opacity', 0.6)
  .css('display', 'none');
```

Возможно, вы обратили внимание на большое число повторяющихся строк в данном фрагменте. Чтобы уменьшить объем программного кода, мы можем переместить реализацию этих действий в функцию, которая будет вызываться для создания каждого ярлыка:

```
function createControl(src) {
  return $('<img/>')
    .attr('src', src)
    .addClass('control')
    .css('opacity', 0.6)
    .css('display', 'none');
}

var $leftRollover = createControl('images/left.gif');
```

```
var $rightRollover = createControl('images/right.gif');
var $enlargeRollover = createControl('images/enlarge.gif');
```

В таблице стилей CSS для нашей страницы мы присвоили свойствам `z-index` этих элементов значение, которое больше значения этого же свойства у изображений, и предусмотрели абсолютное их позиционирование, чтобы ярлыки могли накладываться на изображения:

```
#featured-books .control {
  position: absolute;
  z-index: 3;
  left: 0;
  top: 0;
}
```

Для всех ярлыков используется один и тот же класс `control`, из-за чего может появиться соблазн определить значение свойства `opacity` этого класса непосредственно в таблице стилей CSS. Однако **прозрачность элементов** не во всех броузерах обрабатывается одинаково; в Internet Explorer 60% уровень непрозрачности устанавливается как `filter: alpha(opacity=60)`. Вместо того чтобы бороться с этими различиями, мы установим значение свойства `opacity` с помощью метода `.css()` из библиотеки jQuery, который позволяет обойти эту несовместимость броузеров.

Теперь все, что осталось сделать в обработчиках события `hover`, это поместить изображения в нужное место в дереве DOM и отобразить их.

```
var setUpCovers = function() {
  var $covers = $('#featured-books .covers a');

  $covers.unbind('click mouseenter mouseleave');

  // Левое изображение - сдвинуть вправо
  // (чтобы просмотреть изображения слева).
  $covers.eq(0)
    .css('left', 0)
    .click(function(event) {
      $covers.eq(0).animate({'left': spacing}, 'fast');
      $covers.eq(1).animate({'left': spacing * 2}, 'fast');
      $covers.eq(2).animate({'left': spacing * 3}, 'fast');
      $covers.eq($covers.length - 1)
        .css('left', -spacing)
        .animate({'left': 0}, 'fast', function() {
          $(this).prependTo('#featured-books .covers');
          setUpCovers();
        });
      event.preventDefault();
    }).hover(function() {
      $leftRollover.appendTo(this).show();
    }, function() {
      $leftRollover.hide();
    });
```

```
  // Правое изображение сдвинуть влево
  // (чтобы просмотреть изображения справа).
  $covers.eq(2)
    .css('left', spacing * 2)
    .click(function(event) {
      $covers.eq(0)
        .animate({'left': -spacing}, 'fast', function() {
          $(this).appendTo('#featured-books .covers');
          setUpCovers();
        });
      $covers.eq(1).animate({'left': 0}, 'fast');
      $covers.eq(2).animate({'left': spacing}, 'fast');
      $covers.eq(3)
        .css('left', spacing * 3)
        .animate({'left': spacing * 2}, 'fast');
      event.preventDefault();
    }).hover(function() {
      $rightRollover.appendTo(this).show();
    }, function() {
      $rightRollover.hide();
    });

    // Центральное изображение.
    $covers.eq(1)
      .css('left', spacing)
      .hover(function() {
        $enlargeRollover.appendTo(this).show();
      }, function() {
        $enlargeRollover.hide();
      });
  };
```

Как и в случае с событием `click` выше, мы отключаем обработчики событий `mouseenter` и `mouseleave` в самом начале функции `setUpCovers()`, чтобы не возникло ситуации подключения нескольких обработчиков события к одному и тому же элементу. Здесь мы использовали другую особенность метода `.unbind()`: можно отключить обработчики сразу для нескольких событий, если перечислить в вызове метода типы событий через пробел.

Почему `mouseenter` и `mouseleave`? Когда подключение обработчиков события производится с помощью метода `.hover()`, библиотека jQuery выполняет подключение к двум разным событиям. Первая функция, переданная методу, подключается как обработчик события `mouseenter`, а вторая – как обработчик события `mouseleave`. Поэтому, чтобы отключить обработчики, которые были подключены с помощью метода `.hover()`, нам необходимо отключить события `mouseenter` и `mouseleave`.

Теперь при наведении указателя мыши на изображение поверх него будет появляться соответствующий ярлык, как показано на рис. 9.15.

Рис. 9.15. *При наведении указателя мыши поверх изображения будет появляться соответствующий ярлык*

Увеличение изображения

Теперь наша галерея обладает всей полнотой функциональности. Карусель дает пользователю возможность перемещаться между изображениями. Щелчок мышью на центральном изображении вызывает переход на страницу с увеличенной версией изображения. Но мы можем расширить функциональность увеличения размера этого изображения.

Вместо того чтобы в результате щелчка на центральном изображении перемещать пользователя на другую страницу, мы можем отобразить увеличенное изображение обложки книги прямо поверх страницы.

Примечание

Существует множество реализаций функции отображения информации поверх страницы в виде модулей расширения для библиотеки jQuery. В число наиболее популярных входят: **FancyBox**, **ShadowBox**, **Thickbox**, **SimpleModal** и **jqModal**. Дополнительная информация об использовании модулей расширения приводится в главе 10.

Для отображения увеличенной версии необходим новый элемент изображения, который можно создать в тот же момент, когда создаются ярлыки:

```
var $enlargedCover = $('<img/>')
  .addClass('enlarged')
  .hide()
  .appendTo('body');
```

Мы определим для этого нового класса ряд правил оформления, напоминающих те, что мы уже видели выше:

```
img.enlarged {
  position: absolute;
  z-index: 5;
  cursor: pointer;
}
```

Благодаря тому, что увеличенная версия изображения имеет абсолютное позиционирование, а значение его свойства `z-index` больше значений этого же свойства других элементов, мы сможем помещать его поверх других изображений. Теперь нам необходимо выполнить фактическое позиционирование увеличенного изображения в момент, когда будет выполнен щелчок мышью на центральном изображении в карусели:

```
// Центральное изображение; увеличенная версия.
$covers.eq(1)
  .css('left', spacing)
  .click(function(event) {
    $enlargedCover.attr('src', $(this).attr('href'))
      .css({
        'left': ($('body').width() - 360) / 2,
        'top' : 100,
        'width': 360,
        'height': 444
      }).show();
    event.preventDefault();
  })
  .hover(function() {
    $enlargeRollover.appendTo(this).show();
  }, function() {
    $enlargeRollover.hide();
  });
```

Чтобы узнать, где на сервере находится увеличенное изображение обложки, мы можем использовать ссылки, которые уже присутствуют

Рис. 9.16. *Увеличенное изображение позиционируется в центре страницы по горизонтали*

в разметке HTML. Мы извлекаем его из атрибута href ссылки и устанавливаем в качестве значения атрибута src элемента с увеличенной версией изображения.

Теперь необходимо выполнить позиционирование изображения. Сейчас значения свойств top, width и height жестко определены в программном коде, но, чтобы задать значение свойства left, требуется выполнить некоторые вычисления. Нам нужно, чтобы увеличенное изображение отображалось в центре страницы, но нам заранее не известно значение соответствующей координаты, которое удовлетворит нашему требованию. Мы можем определить горизонтальную координату середины страницы, определив ширину элемента <body> и разделив ее пополам. Увеличенное изображение будет простираться влево и вправо от этой точки на половину своей ширины, поэтому значение свойства left изображения вычисляется как ($('body').width() - 360) / 2, где 360 — это ширина увеличенного изображения обложки. Теперь изображение позиционируется, как требуется, в центре страницы по горизонтали, как показано на рис. 9.16.

Сокрытие увеличенного изображения

Нам необходим механизм удаления увеличенного изображения обложки после его отображения. Самый простой способ заключается в том, чтобы с помощью эффекта растворения скрыть его по событию click:

```
// Центральное изображение; увеличенная версия.
$covers.eq(1)
  .css('left', spacing)
  .click(function(event) {
    $enlargedCover.attr('src', $(this).attr('href'))
      .css({
        'left': ($('body').width() - 360) / 2,
        'top' : 100,
        'width': 360,
        'height': 444
      })
      .show()
      .one('click', function() {
        $enlargedCover.fadeOut();
      });
    event.preventDefault();
  })
  .hover(function() {
    $enlargeRollover.appendTo(this).show();
  }, function() {
    $enlargeRollover.hide();
  });
```

Для подключения обработчика этого события click мы использовали метод .one(), который позволяет обойти сразу пару потенциальных проблем. При подключении обработчика обычным методом .bind() пользо-

ватель сможет щелкнуть на изображении еще раз в процессе воспроизведения эффекта растворения, что приведет к повторному вызову обработчика события. Кроме того, так как для отображения увеличенных версий изображений используется один и тот же элемент, при каждом его отображении будет производиться подключение обработчика. Если не выполнять отключение обработчика вручную, со временем это приведет к накоплению целой стопки обработчиков. Метод `.one()` гарантирует, что обработчик будет отключен сразу же после первого его использования.

Отображение кнопки закрытия

Этого достаточно для удаления увеличенного изображения, но мы никак не показали пользователю, что щелчок на изображении приведет к его удалению. Мы можем проинформировать его об этом, снабдив увеличенное изображение кнопкой Close (закрыть). Создание кнопки аналогично определению других **одиночных элементов** (появляющихся на странице в единственном экземпляре), и для этого мы можем использовать функцию, созданную ранее:

```
var $closeButton = createControl('images/close.gif')
  .addClass('enlarged-control')
  .appendTo('body');
```

Когда на центральном изображении выполняется щелчок мышью и на странице отображается его увеличенная версия, нам необходимо позиционировать и отобразить кнопку:

```
$closeButton.css({
  'left': ($('body').width() - 360) / 2,
  'top' : 100
}).show();
```

Координаты кнопки Close совпадают с координатами увеличенного изображения, потому что совпадают их верхние левые углы, как показано на рис. 9.17.

У нас уже имеется реализованный механизм сокрытия изображения в ответ на щелчок мышью. Обычно в подобных ситуациях можно было бы положиться на то, что **событие всплывет** и щелчок на кнопке Close вызовет тот же эффект. Однако в данном случае кнопка Close не является **элементом-потомком** изображения, несмотря на то что она выводится поверх его. Мы лишь поместили кнопку Close над изображением, а это означает, что событие щелчка на ней не будет передано элементу изображения. Поэтому мы должны обработать щелчок на кнопке Close самостоятельно:

```
// Центральное изображение; увеличенная версия.
$covers.eq(1)
  .css('left', spacing)
  .click(function(event) {
    $enlargedCover.attr('src', $(this).attr('href'))
```

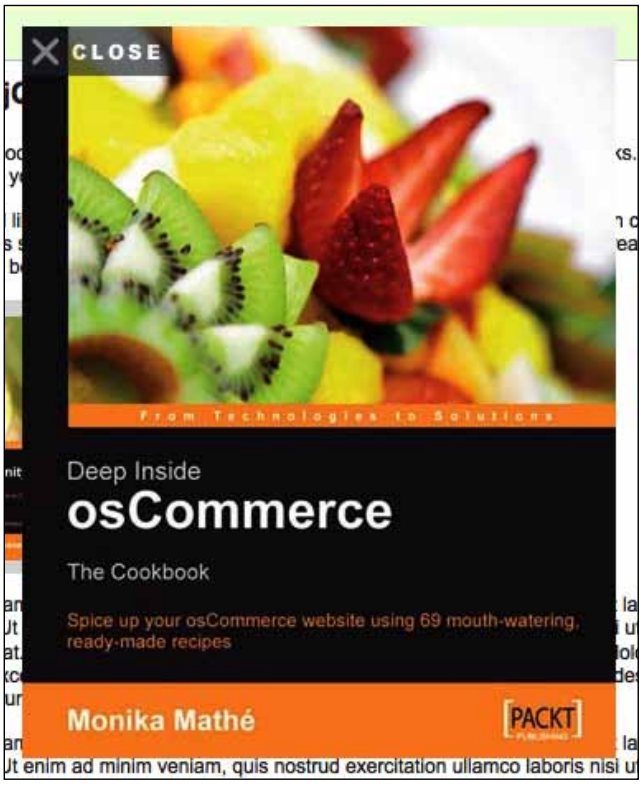


Рис. 9.17. *Верхний левый угол кнопки Close (закрыть) совпадает с верхним левым углом изображения*

```
    .css({
      'left': ($('body').width() - 360) / 2,
      'top' : 100,
      'width': 360,
      'height': 444
    })
    .show()
    .one('click', function() {
      $closeButton.unbind('click').hide();
      $enlargedCover.fadeOut();
    });
  $closeButton
    .css({
      'left': ($('body').width() - 360) / 2,
      'top' : 100
    })
    .show()
    .click(function() {
      $enlargedCover.click();
```

```
      });
    event.preventDefault();
  })
  .hover(function() {
    $enlargeRollover.appendTo(this).show();
  }, function() {
    $enlargeRollover.hide();
  });
```

При отображении кнопки Close мы подключаем к ней обработчик события `click`. Но все, что требуется от данного обработчика, – это просто вызвать обработчик события `click`, который уже подключен к увеличенному изображению. Однако нам необходимо изменить этот обработчик и скрыть в нем кнопку Close. Одновременно мы отключим обработчик с целью предотвращения их возможного накопления.

Отображение дополнительной информации

В исходной разметке HTML у нас имеется информация о стоимости книг, и мы можем отобразить эту дополнительную информацию поверх увеличенного изображения обложки книги. Мы воспользуемся приемом, который был разработан для кнопки Close, но на этот раз для отображения текстовой информации, а не изображения.

И снова в самом начале программного кода JavaScript мы создаем **одиночный элемент:**

```
var $priceBadge = $('<div/>')
  .addClass('enlarged-price')
  .css('opacity', 0.6)
  .css('display', 'none')
  .appendTo('body');
```

Так как ценник будет частично прозрачным, лучше использовать высококонтрастную комбинацию цветов шрифта и фона:

```
.enlarged-price {
  background-color: #373c40;
  color: #fff;
  width: 80px;
  padding: 5px;
  font-size: 18px;
  font-weight: bold;
  text-align: right;
  position: absolute;
  z-index: 6;
}
```

Прежде чем мы сможем отобразить ценник, нам необходимо заполнить его действительной информацией из разметки HTML. Внутри обработчика события `click` центрального изображения ключевое слово `this` ссылается на элемент ссылки. Поскольку цена определяется элементом

, находящимся внутри ссылки, получить текст не представляет труда:

```
var price = $(this).find('.price').text();
```

Теперь вместе с увеличенной версией изображения обложки можно отобразить и ценник:

```
$priceBadge.css({
  'right': ($('body').width() - 360) / 2,
  'top' : 100
}).text(price).show();
```

Этот вызов отобразит ценник в правом верхнем углу увеличенного изображения, как показано на рис. 9.18.

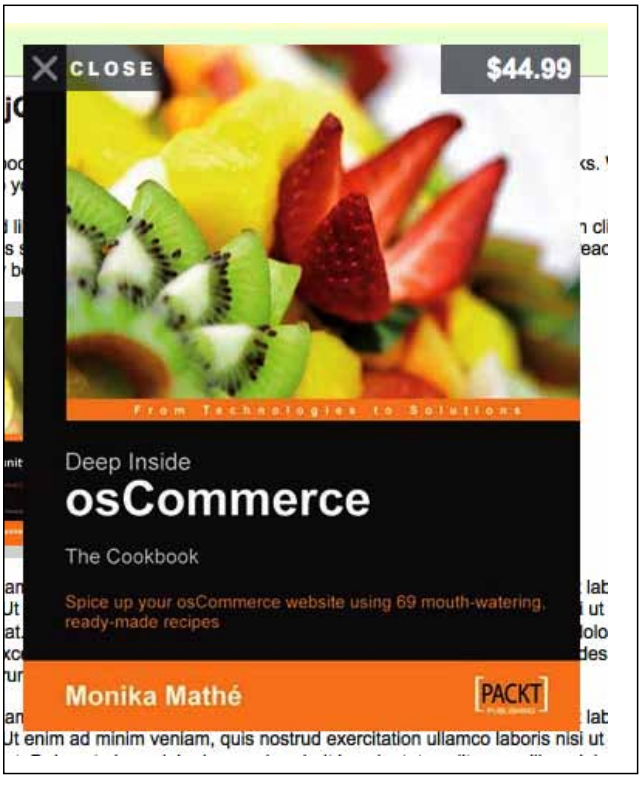


Рис. 9.18. *Ценник отображается в правом верхнем углу увеличенного изображения*

Как только мы поместим вызов метода `$priceBadge.hide();` в обработчик события `click` увеличенного изображения, чтобы скрыть ценник в нужный момент времени, работу можно будет считать законченной.

Анимация увеличения изображения

Сейчас, если пользователь щелкнет на центральном изображении, увеличенная версия просто появится в центре страницы без каких-либо эффектов. Чтобы исправить положение и реализовать плавное изменение размеров изображения от миниатюрного представления до полноразмерной версии, мы можем воспользоваться встроенными в библиотеку jQuery средствами воспроизведения анимационных эффектов.

Для этого необходимо определить начальные координаты анимационного эффекта, то есть координаты центрального изображения. Для вычисления этих координат необходимо выполнить обход дерева DOM с использованием обычного JavaScript, но библиотека jQuery обеспечивает более простой способ. Метод `.offset()` возвращает объект, содержащий координаты `top` и `left` элемента относительно страницы. Затем мы можем вставить в этот объект значения свойств `width` и `height` изображения и получить информацию о позиционировании в виде аккуратного пакета.

```
var startPos = $(this).offset();
startPos.width = $(this).width();
startPos.height = $(this).height();
```

Теперь из этих координат можно легко вычислить конечные координаты. Мы сохраним их в виде такого же объекта.

```
var endPos = {};
endPos.width = startPos.width * 3;
endPos.height = startPos.height * 3;
endPos.top = 100;
endPos.left = ($('body').width() - endPos.width) / 2;
```

Затем можно использовать эти два объекта как **отображения** атрибутов CSS для передачи таким методам, как `.css()` и `.animate()`.

```
$enlargedCover.attr('src', $(this).attr('href'))
  .css(startPos)
  .show()
  .animate(endPos, 'normal', function() {
    $enlargedCover
      .one('click', function() {
        $closeButton.unbind('click').hide();
        $priceBadge.hide();
        $enlargedCover.fadeOut();
      });
    $closeButton
      .css({
        'left': endPos.left,
        'top' : endPos.top
      })
```

```
        .show()
        .click(function() {
          $enlargedCover.click();
        });
      $priceBadge
        .css({
          'right': endPos.left,
          'top' : endPos.top
        })
        .text(price)
        .show();
  });
```

Обратите внимание, что кнопку Close и ценник нельзя отображать, пока не завершится воспроизведение анимационного эффекта, поэтому мы переместили соответствующий программный код внутрь функции обратного вызова метода `.animate()`. Кроме того, мы воспользовались этим обстоятельством, чтобы упростить вызовы метода `.css()` для обоих элементов, повторно применив информацию о позиционировании, вычисленную для увеличенного изображения.

Теперь изображение плавно увеличивает свои размеры, как показано на рис. 9.19 – 9.22.

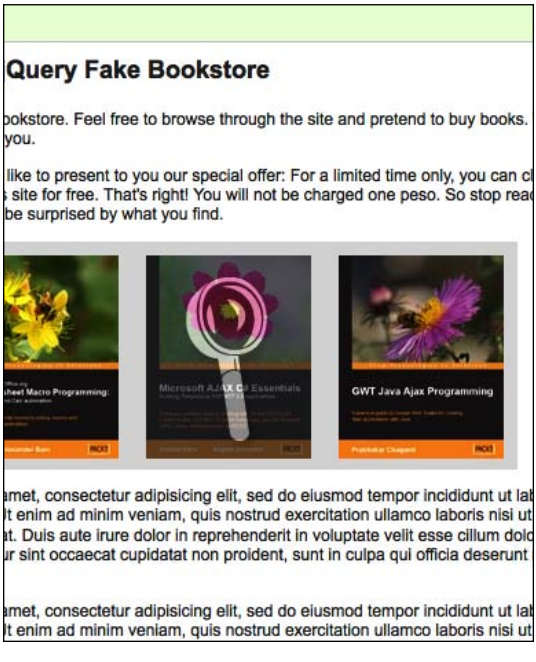


Рис. 9.19. *Вид страницы перед щелчком на центральном изображении*

Рис. 9.20. *Вид страницы после щелчка на центральном изображении в начале воспроизведения эффекта*

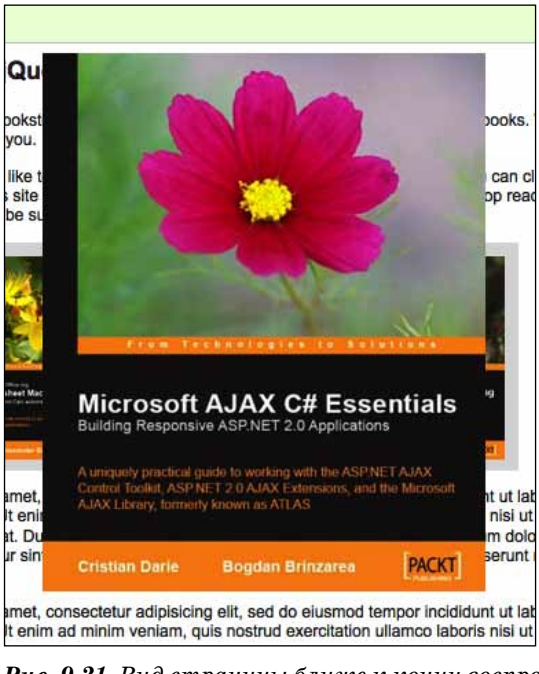


Рис. 9.21. *Вид страницы ближе к концу воспроизведения эффекта*

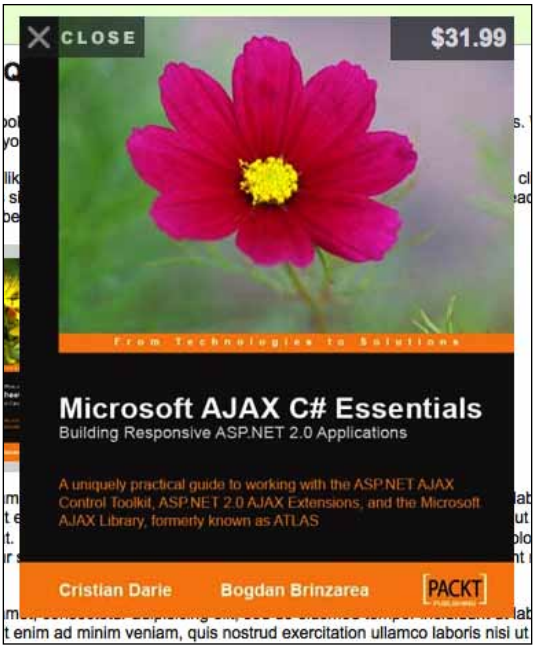


Рис. 9.22. *Вид страницы по окончании воспроизведения эффекта*

Задержка воспроизведения анимационных эффектов до окончания загрузки изображения

Анимационный эффект воспроизводится без проблем, но он зависит от скорости соединения с сайтом. Если для загрузки увеличенного изображения требуется некоторое время, то в первые мгновения воспроизведения анимационного эффекта можно наблюдать красный крест как признак испорченного изображения или прежнее изображение. Мы можем сделать эффект более элегантным, если перед запуском анимационного эффекта дождемся, чтобы изображение полностью загрузилось:

```
$enlargedCover.attr('src', $(this).attr('href'))
  .css(startPos)
  .show();

var performAnimation = function() {
  $enlargedCover.animate(endPos, 'normal', function() {
    $enlargedCover.one('click', function() {
      $closeButton.unbind('click').hide();
      $priceBadge.hide();
      $enlargedCover.fadeOut();
    });
    $closeButton
      .css({
```

```
        'left': endPos.left,
        'top' : endPos.top
      })
      .show()
      .click(function() {
        $enlargedCover.click();
      });
    $priceBadge
      .css({
        'right': endPos.left,
        'top' : endPos.top
      })
      .text(price)
      .show();
  });
};
if ($enlargedCover[0].complete) {
  performAnimation();
}
else {
  $enlargedCover.bind('load', performAnimation);
}
```

Рассмотрим два возможных сценария: изображение может быть получено практически мгновенно (возможно, благодаря работе механизма кэширования), и для его загрузки требуется некоторое время. В первом случае атрибут `complete` изображения будет иметь значение `true`, поэтому мы сразу можем вызвать нашу новую функцию `performAnimation()`. Во втором случае необходимо подождать, пока изображение загрузится полностью, и только потом вызывать функцию `performAnimation()`. Это редкий случай, когда предпочтительнее использовать стандартное событие DOM `load`, а не событие `ready`, предоставляемое библиотекой jQuery. Событие `load` возбуждается окном, изображением или фреймом, когда будет загружено все содержимое, поэтому мы можем дождаться появления этого события, чтобы обеспечить корректное отображение изображения. Анимационный эффект воспроизводится, только когда запускается обработчик события.

Примечание

Для большей ясности при подключении обработчика события вместо сокращенного метода `.load()` мы использовали вызов `.bind('load')`, поскольку существует еще один метод `.load()`, который является методом поддержки технологии AJAX. Эти два способа являются взаимозаменяемыми.

Броузеры Internet Explorer и Firefox по-разному обрабатывают ситуацию, когда изображение уже находится в кэше броузера. Так, Firefox немедленно возбуждает событие `load`, а Internet Explorer никогда не возбудит это событие, потому что загрузка фактически уже была вы-

полнена. Проверка атрибута `complete` компенсирует это различие в реализациях.

Добавление индикатора загрузки

Но теперь может возникнуть неприятная ситуация в медленных сетях, когда загрузка изображения занимает некоторое время. Во время загрузки изображения у пользователя может сложиться впечатление, что страница ничего не делает. Как и при загрузке заголовков новостей, мы должны показать пользователю, что выполняется некоторая операция, отображая **индикатор загрузки**.

В качестве индикатора будет использоваться другое **одиночное** изображение, которое будет отображаться в нужный момент времени:

```
var $waitThrobber = $('<img/>')
  .attr('src', 'images/wait.gif')
  .addClass('control')
  .css('z-index', 4)
  .hide();
```

В качестве этого индикатора мы будем использовать анимированное изображение в формате GIF, показанное на рис. 9.23, потому что движущееся изображение укрепит пользователя в мнении, что что-то происходит.

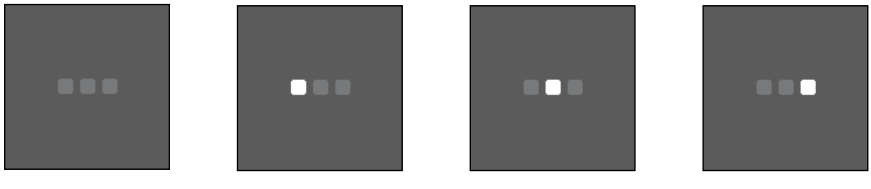


Рис. 9.23. *Индикатор загрузки*

Теперь, когда элемент уже определен, для обслуживания индикатора загрузки нам потребуется добавить всего две строки программного кода. В самом начале обработчика события `click` для центрального изображения перед выполнением каких-либо действий нам нужно отобразить индикатор:

```
$waitThrobber.appendTo(this).show();
```

А в начале функции `performAnimation()`, когда точно известно, что изображение уже загружено, мы скрываем индикатор:

```
$waitThrobber.hide();
```

Добавив индикатор загрузки, мы закончили реализацию функциональности ярлыка, отвечающего за увеличение изображения. Анимированное изображение выводится поверх левого верхнего угла изображения обложки, как показано на рис. 9.24.

Рис. 9.24. *Индикатор загрузки выводится в левом верхнем углу изображения обложки*

Окончательная версия

В этой главе была представлена лишь малая часть возможностей по динамическому отображению информации в Сети с использованием анимированных изображений и приемов прокрутки текста. Ниже приводится полная реализация карусели изображений:

```
$(document).ready(function() {
  var spacing = 140;

  function createControl(src) {
    return $('<img/>')
      .attr('src', src)
      .addClass('control')
      .css('opacity', 0.6)
      .css('display', 'none');
  }

  var $leftRollover = createControl('images/left.gif');
  var $rightRollover = createControl('images/right.gif');
  var $enlargeRollover = createControl('images/enlarge.gif');
  var $enlargedCover = $('<img/>')
    .addClass('enlarged')
    .hide()
    .appendTo('body');
  var $closeButton = createControl('images/close.gif')
    .addClass('enlarged-control')
    .appendTo('body');
  var $priceBadge = $('<div/>')
    .addClass('enlarged-price')
    .css('opacity', 0.6)
    .css('display', 'none')
    .appendTo('body');
  var $waitThrobber = $('<img/>')
    .attr('src', 'images/wait.gif')
    .addClass('control')
    .css('z-index', 4)
```

```
  .hide();

$('#featured-books').css({
  'width': spacing * 3,
  'height': '166px',
  'overflow': 'hidden'
}).find('.covers a').css({
  'float': 'none',
  'position': 'absolute',
  'left': 1000
});

var setUpCovers = function() {
  var $covers = $('#featured-books .covers a');

  $covers.unbind('click mouseenter mouseleave');

// Левое изображение – сдвинуть вправо
// (чтобы просмотреть изображения слева).
  $covers.eq(0)
    .css('left', 0)
    .click(function(event) {
      $covers.eq(0).animate({'left': spacing}, 'fast');
      $covers.eq(1).animate({'left': spacing * 2}, 'fast');
      $covers.eq(2).animate({'left': spacing * 3}, 'fast');
      $covers.eq($covers.length - 1)
        .css('left', -spacing)
        .animate({'left': 0}, 'fast', function() {
          $(this).prependTo('#featured-books .covers');
          setUpCovers();
        });

      event.preventDefault();
    }).hover(function() {
      $leftRollover.appendTo(this).show();
    }, function() {
      $leftRollover.hide();
    });

// Правое изображение – сдвинуть влево
// (чтобы просмотреть изображения справа).
$covers.eq(2)
  .css('left', spacing * 2)
  .click(function(event) {
    $covers.eq(0)
      .animate({'left': -spacing}, 'fast', function() {
        $(this).appendTo('#featured-books .covers');
        setUpCovers();
      });
    $covers.eq(1).animate({'left': 0}, 'fast');
    $covers.eq(2).animate({'left': spacing}, 'fast');
```

```
    $covers.eq(3)
      .css('left', spacing * 3)
      .animate({'left': spacing * 2}, 'fast');

    event.preventDefault();
  }).hover(function() {
    $rightRollover.appendTo(this).show();
  }, function() {
    $rightRollover.hide();
  });

// Центральное изображение; увеличенная версия.
$covers.eq(1)
  .css('left', spacing)
  .click(function(event) {
    $waitThrobber.appendTo(this).show();
    var price = $(this).find('.price').text();
    var startPos = $(this).offset();
    startPos.width = $(this).width();
    startPos.height = $(this).height();
    var endPos = {};
    endPos.width = startPos.width * 3;
    endPos.height = startPos.height * 3;
    endPos.top = 100;
    endPos.left = ($('body').width() - endPos.width) / 2;

    $enlargedCover.attr('src', $(this).attr('href'))
      .css(startPos)
      .show();
    var performAnimation = function() {
      $waitThrobber.hide();
      $enlargedCover.animate(endPos, 'normal',
        function() {
          $enlargedCover.one('click', function() {
            $closeButton.unbind('click').hide();
            $priceBadge.hide();
            $enlargedCover.fadeOut();
          });
          $closeButton
            .css({
              'left': endPos.left,
              'top' : endPos.top
            })
            .show()
            .click(function() {
              $enlargedCover.click();
            });
          $priceBadge
            .css({
              'right': endPos.left,
              'top' : endPos.top
```

```
            })
            .text(price)
            .show();
        });
      };
      if ($enlargedCover[0].complete) {
        performAnimation();
      }
      else {
        $enlargedCover.bind('load', performAnimation);
      }
      event.preventDefault();
    })
    .hover(function() {
      $enlargeRollover.appendTo(this).show();
    }, function() {
    $enlargeRollover.hide();
  });
};
setUpCovers();
});
```

В заключение

В этой главе мы рассмотрели элементы страницы, которые изменяются с течением времени либо сами по себе, либо в ответ на действия пользователя. Показанные приемы **перемещения и прокрутки** действительно могут вывести присутствие в Интернете на современный уровень, отличный от того, который возможен при использовании традиционных приемов. Мы изучили способы представления на странице информации из рассылки, поставляемой в формате XML, а также приемы прокрутки изображений с возможностью приостановки. Помимо способа отображения набора изображений в виде галереи с возможностью навигации по ней мы обсудили прием увеличения размеров изображения для более детального знакомства с ним с применением анимационного эффекта и предоставлением элементов управления пользовательского интерфейса ненавязчивым способом.

Эти приемы могут объединяться в самые разные комбинации для того, чтобы вдохнуть жизнь в статичные страницы и одновременно повысить удобство наших веб-приложений. Благодаря возможностям библиотеки jQuery можно без усилий создавать анимационные эффекты, добиться которых иными способами было бы довольно трудно.

10

Использование модулей расширения

На протяжении всей книги мы рассматривали множество способов использования библиотеки jQuery для решения разнообразных задач. Однако относительно неизведанным остался еще один аспект – возможность расширения библиотеки jQuery. Помимо мощного ядра библиотека обладает элегантной **расширяемой архитектурой**, позволяющей разработчикам расширять jQuery, делая ее еще более многофункциональной библиотекой.

Сообществом пользователей jQuery по мере своего развития были созданы сотни модулей расширения, от маленьких вспомогательных селекторов до полномасштабных элементов пользовательского интерфейса. Мы уже обсуждали возможности модулей расширения и даже разработали один в главе 7. В этой главе мы узнаем, как находить расширения, созданные другими разработчиками, и как добавлять их в свои страницы. Мы исследуем популярный модуль расширения Form и официальную библиотеку расширений jQuery UI, а затем перечислим и дадим краткое описание некоторых других популярных расширений, «рекомендуемых авторами».

Поиск расширений и получение справочной информации

Веб-сайт jQuery предоставляет огромный репозиторий модулей расширения, расположенный по адресу *http://plugins.jquery.com/*, с информацией об их рейтингах, о вышедших версиях и с сообщениями об ошибках. Репозиторий **Plugin Repository** является отличной отправной точкой для поиска документации. Каждое расширение, включенное в репозиторий, доступно для загрузки в виде архива `.zip`, причем для многих из них имеются ссылки на демонстрационные примеры, примеры программного кода и обучающие руководства, которые помогут приступить к работе с ними.

Еще больше расширений можно найти в универсальных репозиториях, таких как *http://github.com/*, и в блогах разработчиков расширений.

Если на возникающие вопросы не удается найти ответы ни в репозитории Plugin Repository, ни на веб-сайте автора, ни в комментариях внутри программного кода расширения, всегда можно обратиться к группе jQuery Google Group, по адресу: *http://groups.google.com/group/jquery-en/*. Многие авторы расширений присутствуют в списках этой группы и всегда готовы помочь в решении любых проблем, с которыми сталкиваются начинающие пользователи.

Как использовать расширения

Использовать расширение для библиотеки jQuery очень просто. В первую очередь его необходимо подключить в разделе `<head>` документа, но только после подключения основного файла jQuery:

```
<head>
  <meta http-equiv="Content-Type"
    content="text/html; charset=utf-8"/>
  <script src="jquery.js" type="text/javascript"></script>
  <script src="jquery.plugin.js"
    type="text/javascript"></script>
  <script src="custom.js" type="text/javascript"></script>
  <title>Example</title>
</head>
```

После этого остается лишь задействовать в своем сценарии JavaScript методы, которые создаются или расширяются модулем расширения. Часто бывает достаточно добавить единственную строку в вызов метода `$(document).ready()`, чтобы вызвать некоторое действие:

```
$(document).ready(function() {
  $('#myID').somePlugin();
});
```

Многие расширения обеспечивают значительную гибкость, позволяя указывать ряд необязательных параметров, благодаря которым можно изменять поведение расширений. Мы можем настраивать их под свои нужды обычно за счет передачи **отображения** в виде аргумента метода:

```
$(document).ready(function() {
  $('#myID').somePlugin ({
    send: true,
    message: 'This plugin is great!'
  });
});
```

Синтаксис вызова методов расширений jQuery, как правило, очень близок к синтаксису вызова методов, входящих в ядро библиотеки. Теперь,

когда мы узнали, как подключить расширение к веб-странице, познакомимся поближе с парой популярных расширений.

Расширение Form

Расширение Form – потрясающий пример сценария, делающего решение трудных и сложных задач до банальности простым. Файл расширения, а также подробная документация доступны по адресу: *http://malsup/com/jquery/form/*.

Основой расширения является метод `.ajaxForm()`. Он позволяет превратить обычную форму в форму с поддержкой AJAX всего одной простой строчкой кода:

```
$(document).ready(function() {
  $('#myForm').ajaxForm();
});
```

Этот пример подготовит форму `<form id="myForm">` к отправке данных без необходимости полного обновления текущей страницы. Уже сама по себе эта возможность является привлекательной, но ее истинная мощь обеспечивается за счет отображения параметров, которые можно передавать методу. Например, следующий фрагмент вызывает метод `.ajaxForm()` с параметрами `target`, `beforeSubmit` и `success`:

```
$(document).ready(function() {
  function validateForm() {
    // здесь можно поместить программный код, выполняющий проверку формы
    // мы можем вернуть значение false,
    // чтобы отменить операцию отправки данных
  };
  $('#test-form').ajaxForm({
    target: '#log',
    beforeSubmit: validateForm,
    success: function() {
      alert('Thanks for your comment!');
    }
  });
});
```

Параметр `target` определяет элементы (в данном случае элемент со значением атрибута `id="log"`), которые будут обновлены данными, полученными от сервера.

Параметр `beforeSubmit` позволяет определить операции, которые будут выполняться перед отправкой формы. В этом примере параметр ссылается на функцию `validateForm()`. Если функция вернет значение `false`, форма не будет отправлена на сервер.

Параметр `success` позволяет определить операции, которые будут выполняться после успешной отправки формы. В данном примере про-

сто выводится сообщение, извещающее пользователя о том, что форма была отправлена.

В число других параметров, которые можно передавать методу .ajaxForm() и которые напоминают параметр .ajaxSubmit(), входят:

- url: адрес URL, по которому должны отправляться данные формы, если он отличается от указанного в атрибуте action формы.
- type: метод отправки формы – или GET, или POST. По умолчанию используется метод, указанный в атрибуте method формы, или метод GET в случае его отсутствия.
- dataType: ожидаемый тип данных ответа сервера. Возможные значения: null, xml, script и json. По умолчанию используется значение null (ответ в формате HTML).
- resetForm: тип *Boolean*; значение по умолчанию false. Если в этом параметре передать значение true, то после успешной отправки все поля формы будут сбрасываться в значения по умолчанию.
- clearForm: тип *Boolean*; значение по умолчанию false. Если в этом параметре передать значение true, то после успешной отправки все поля формы будут очищены.

Расширение Form предоставляет несколько других методов, оказывающих помощь в обработке форм и их данных. Поближе познакомиться с этими методами, а также с демонстрационными примерами можно на странице *http://www.malsup.com/jquery/form/*.

Советы и рекомендации

Обычно метод .ajaxForm() удобнее, чем метод .ajaxSubmit(), за счет дополнительной гибкости. Когда желательно доверить расширению управление подключением всех обработчиков событий, а также вызывать метод .ajaxSubmit() в нужный момент времени, лучше использовать метод .ajaxForm(). Когда необходимо иметь более тонкий контроль над обработкой события submit, рекомендуется использовать метод .ajaxSubmit().

Оба метода, .ajaxForm() и .ajaxSubmit(), по умолчанию используют значения атрибутов action и method в разметке формы. Расширение будет действовать в соответствии с нашими ожиданиями, не требуя производить какие-либо дополнительные манипуляции, при условии использования корректной разметки формы. В качестве дополнительной выгоды мы автоматически получаем следование принципу **прогрессивного улучшения** – при отсутствии поддержки JavaScript форма сохраняет свою функциональность.

Обычно, если для отправки формы используется элемент с атрибутом name, вместе с формой на сервер отправляются значения атрибутов name и value этого элемента. Метод .ajaxForm() в отношении таких элементов является **активным** – он добавляет обработчики события click ко всем

элементам <input type="submit">, благодаря чему он знает, какой элементов инициировал отправку формы. Метод .ajaxSubmit(), напротив, является **реактивным** и не имеет возможности получить эту информацию. Он не фиксирует информацию о том, какой элемент вызвал отправку формы. То же самое относится и к элементам <input type="image">: метод .ajaxForm() обрабатывает их, а метод .ajaxSubmit() игнорирует.

Если вместе с формой не выполняется выгрузка файла на сервер, методы .ajaxForm() и .ajaxSubmit() передают свои параметры методу $.ajax(), который является частью ядра библиотеки jQuery. Благодаря этому с помощью расширения Form можно передавать методу $.ajax() любые допустимые параметры. Применяя данную особенность, можно еще больше повысить устойчивость приемов использования технологии AJAX, допустим так:

```
$(document).ready(function() {
  $(#myForm).ajaxForm({
    timeout: 2000,
    error: function (xml, status, e) {
      alert(e.message);
    }
  });
});
```

Когда не требуется точная настройка поведения методов .ajaxForm() и .ajaxSubmit(), вместо отображения с параметрами им можно передавать функцию. Поскольку такая функция будет интерпретироваться как обработчик случая успешного получения ответа, мы можем выполнять обработку ответа, полученного от сервера, например:

```
$(document).ready(function() {
  $(#myForm).ajaxForm(function(responseText) {
    alert(responseText);
  });
});
```

Библиотека расширений jQuery UI

Тогда как расширение Form делает что-то одно и делает это прекрасно, расширение **jQuery UI** решает весьма широкий круг задач (и хорошо с ними справляется). В действительности jQuery UI – это не столько расширение, сколько целая библиотека расширений.

Во главе с Полом Бакаусом (Paul Bakaus) группа разработчиков jQuery UI создала несколько **компонентов взаимодействий** и полноценных **элементов пользовательского интерфейса (виджетов)**, чтобы помочь делать веб-приложения похожими на обычные приложения. В число компонентов взаимодействий входят методы перетаскивания элементов, их сортировки и масштабирования. Текущая стабильная версия

библиотеки включает такие виджеты, как сворачиваемая панель, календарь, диалог, движок и вкладки; еще больше элементов находится в активной разработке. Дополнительно к базовым анимационным эффектам jQuery библиотека jQuery UI предоставляет обширный набор улучшенных эффектов.

В рамках одной главы невозможно полностью описать все возможности библиотеки UI, поэтому мы ограничимся изучением визуальных эффектов, компонента Sortable и виджета Dialog. Исходные тексты, документация и демонстрационные примеры для всех модулей библиотеки jQuery UI доступны по адресу: *http://ui.jquery.com/*.

Эффекты

Модуль эффектов библиотеки jQuery UI поставляется в виде базового файла и множества отдельных файлов эффектов. Базовый файл включает реализацию анимационных эффектов изменения цвета и классов CSS, а также улучшенную реализацию переходов.

Эффекты управления цветом

При наличии в документе ссылки на базовый файл эффектов мы получаем в свое распоряжение расширенный метод `.animate()`, способный принимать дополнительные свойства стиля, такие как `borderTopColor`, `backgroundColor` и `color`. Например, он позволяет постепенно изменить черный цвет текста на белом фоне до белого цвета текста на черном фоне:

```
$(document).ready(function() {
  $('#mydiv').animate({
    color: '#fff',
    backgroundColor: '#000'
  }, 'slow');
});
```

Примерно на середине воспроизведения эффекта элемент `<div>` выглядит, как показано на рис. 10.1.

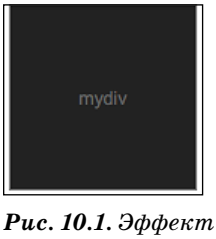


Рис. 10.1. *Эффект одновременного изменения цвета текста и фона*

Элемент выглядит именно так, как и ожидалось; текст постепенно становится белым, а фон – черным.

Эффекты управления классами

Три метода управления классами, с которыми нам приходилось работать в предыдущих главах, `.addClass()`, `.removeClass()` и `.toggleClass()`, теперь принимают второй необязательный аргумент, определяющий продолжительность воспроизведения эффекта. Благодаря этому можно вызывать их как `.addClass('highlight', 'fast')`, или `.removeClass('highlight', 'slow')`, или `.toggleClass('highlight', 1000)`.

Улучшенная реализация переходов

Улучшенные функции реализации **переходов** позволяют изменять скорость и дистанцию в различных точках выполнения переходов в процессе воспроизведения эффектов. Например, функция `easeInQuart` заканчивает воспроизведение эффекта со скоростью в четыре раза большей, чем в самом начале. Любым методам воспроизведения эффектов, входящим в состав библиотеки jQuery или jQuery UI, мы можем передавать собственные функции реализации переходов. Делать это можно, передавая либо дополнительный аргумент, либо дополнительный параметр в **отображении с параметрами** в зависимости от используемого синтаксиса. Например, в предыдущем примере воспроизведения эффекта изменения цвета можно было бы указать функцию `easeInQuart` в виде дополнительного аргумента:

```
$(document).ready(function() {
  $('#mydiv').animate({
    color: '#fff',
    backgroundColor: '#000'
  }, 'slow', 'easeInQuart');
});
```

Или в виде дополнительного параметра во втором отображении:

```
$(document).ready(function() {
  $('#mydiv').animate({
    color: '#fff',
    backgroundColor: '#000'
  }, {
    duration: 'slow',
    easing: 'easeInQuart'
  });
});
```

Демонстрационные примеры всего набора функций реализации переходов можно найти по адресу: *http://gsgd.co.uk/sandbox/jquery/easing/*.

Дополнительные эффекты

В отдельных файлах эффектов реализованы различные дополнительные варианты переходов, каждый из которых может быть выполнен с помощью метода `.effect()`, а также присутствуют расширенные вер-

сии методов jQuery, таких как `.show()`, `.hide()` и `.toggle()`. Например, эффект взрыва, который скрывает элемент, взрывая его на заданное количество фрагментов, можно реализовать с помощью метода `.effect()`:

```
$(document).ready(function() {
  $('#explode').effect('explode', {pieces: 16}, 800);
});
```

Или с помощью метода `.hide()`:

```
$(document).ready(function() {
  $('#explode').hide('explode', {pieces: 16}, 800);
});
```

В любом случае этот эффект скроет прямоугольник, который изначально выглядит, как изображено на рис. 10.2.

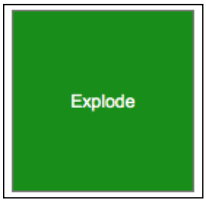


Рис. 10.2. *Вид прямоугольника до воспроизведения эффекта*

В середине воспроизведения эффекта прямоугольник выглядит, как показано на рис. 10.3.

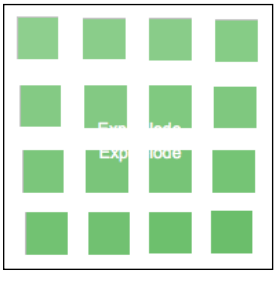


Рис. 10.3. *Эффект взрыва в середине воспроизведения*

В конце воспроизведения эффекта прямоугольник окажется скрытым.

Компоненты взаимодействий

В число **компонентов взаимодействий** библиотеки jQuery UI входит компонент **Sortable**, который способен преобразовать практически любую группу элементов в список, дающий возможность перетаскивать его элементы. На рис. 10.4 изображен неупорядоченный список элементов, к которым были применены некоторые стили CSS.

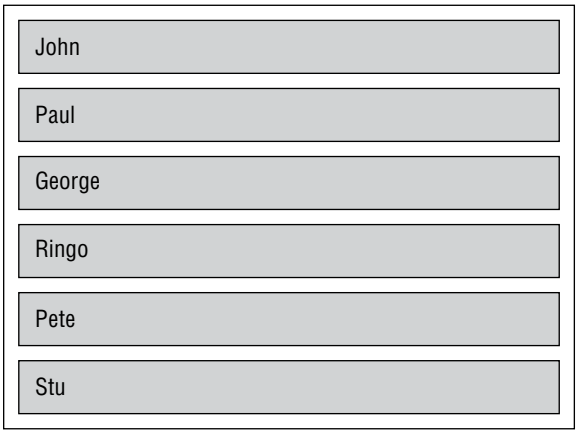


***Рис. 10.4.** Первоначальный вид списка*

Разметка HTML этого списка выглядит очень просто:

```
<ul id="sort-container">
  <li>John</li>
  <li>Paul</li>
  <li>George</li>
  <li>Ringo</li>
  <li>Pete</li>
  <li>Stu</li>
</ul>
```

Теперь, чтобы сделать данный список сортируемым, достаточно написать следующий программный код:

```
$(document).ready(function() {
  $('#sort-container').sortable();
});
```

Эта единственная строка в вызове `$(document).ready()` позволяет буксировать мышью любой элемент списка и перемещать его в любую позицию в пределах списка, как показано на рис. 10.5.

Мы можем расширить возможности пользователя при взаимодействии с интерфейсом путем передачи методу `.sortable()` дополнительных параметров. У этого метода имеется более тридцати возможных параметров, однако в нашем примере мы воспользуемся лишь некоторыми из них:

```
$(document).ready(function() {
  $('#sort-container').sortable({
    opacity: .5,
    cursor: 'move',
    axis: 'y'
  });
});
```

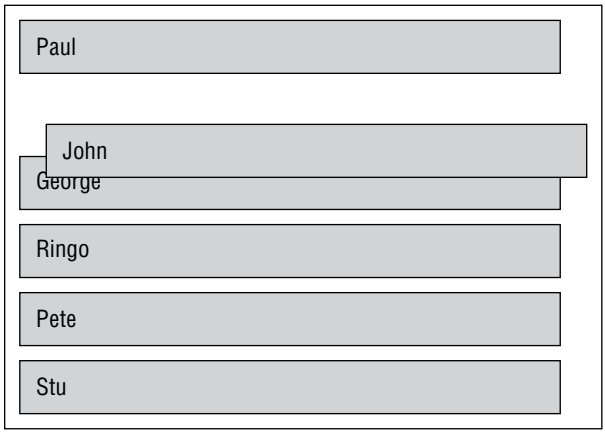


Рис. 10.5. *Любой элемент может быть перемещен в любую позицию в пределах списка*

Имена первых двух параметров, `opacity` и `cursor`, говорят сами за себя. Третий параметр, `axis`, ограничивает возможность перемещения элементов в процессе сортировки определенной осью координат (в данном случае осью y), как показано на рис. 10.6.

Как видно по более светлому цвету фона перемещаемого элемента на рис. 10.6, мы воспользовались также возможностью определить класс `ui-sortable-helper` в таблице стилей CSS, который автоматически применяется к перемещаемым элементам.

За дополнительной информацией обо всех компонентах взаимодействий, входящих в состав библиотеки jQuery UI, обращайтесь по адресу: *http://docs.jquery.com/UI#Interaction*.

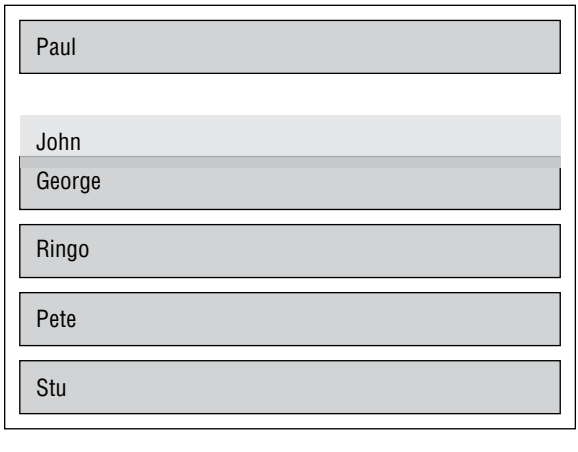


Рис. 10.6. *Параметр axis ограничивает возможность перемещения элемента списка определенной осью координат*

Виджеты

В дополнение к готовым компонентам библиотека jQuery UI включает в себя набор надежных **виджетов** пользовательского интерфейса, которые не требуют дополнительной настройки и выглядят, как привычные элементы интерфейса обычных приложений. Например, виджет **Dialog**, использующий компоненты *draggable* и *resizable*, может применяться для воспроизведения диалогового окна, благодаря чему у нас отпадает необходимость создавать собственные диалоги.

Как и другие виджеты пользовательского интерфейса, виджет Dialog может принимать огромное число параметров. Его подходящим образом названный метод `.dialog()` может также принимать строковые аргументы, которые оказывают влияние на поведение диалога. В самом простом случае метод `.dialog()` преобразует существующий элемент в диалог и отображает его вместе с элементами, содержащимися в нем. Например, для начала возьмем элемент `<div>` с простейшей структурой.

```
<div id="dlg">My Dialog</div>
```

Неудивительно, что этот элемент `<div>` имеет вид простого текстового блока, как показано на рис. 10.7.

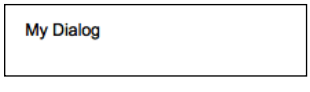


Рис. 10.7. *Простой элемент выглядит как обычный текстовый блок*

Как только броузер создаст дерево DOM, мы можем вызвать простой диалог.

```
$(document).ready(function() {
  $('#dlg').dialog();
});
```

Теперь текст отображается в окне диалога, как показано на рис. 10.8.

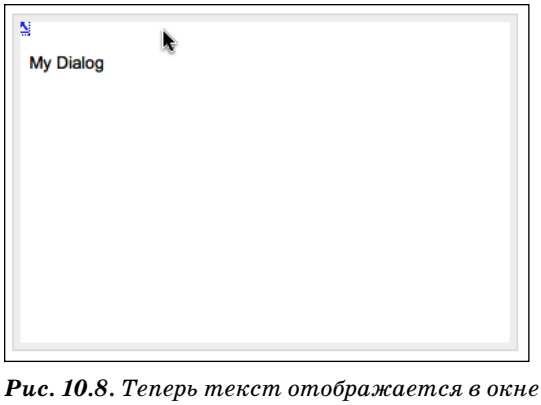


Рис. 10.8. *Теперь текст отображается в окне диалога*

Изменить размеры диалогового окна можно, ухватив и переместив какой-либо его край указателем мыши. Переместить это окно можно, ухватив мышью в пределах верхней области чуть ниже верхней границы. А закрыть – щелкнув на ссылке с изображением крестика в правом верхнем углу.

Однако мы можем улучшить вид диалога, используя стили CSS. Библиотека jQuery UI предоставляет лишь минимальный набор стилей, необходимых для обеспечения функциональности виджетов, и оставляет реализацию внешнего вида за нами. На рис. 10.9 показано, как выглядит диалог после применения **темы** по умолчанию.

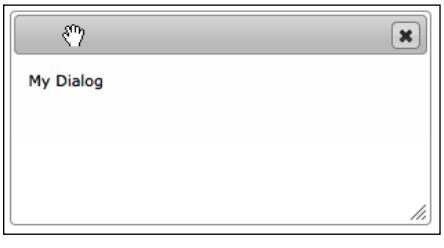


Рис. 10.9. *Вид диалога после применения темы по умолчанию*

Теперь более ясно обозначены различные области диалога, и при наведении на них указатель мыши изменяется, обеспечивая визуальную обратную связь, позволяющую опознать элементы диалога, которые могут использоваться для его перемещения и изменения его размеров.

Как и другие методы библиотеки jQuery UI, метод `.dialog()` может принимать различные параметры. Некоторые параметры определяют внешний вид диалога, другие позволяют возбуждать события. Ниже приводится пример использования некоторых из этих параметров:

```
$(document).ready(function() {
  var $dlg = $('#dlg');
  var dlgText = $dlg.text();
  $dlg.dialog({
    autoOpen: false,
    title: dlgText,
    open: function() {
      $dlg.empty();
    },
    buttons: {
      'add message': function() {
        $dlg.append('<p>Inserted message</p>');
      },
      'erase messages': function() {
        $('p', $dlg).remove();
      }
    }
  });
```

```
    $('#do-dialog').click(function() {
      $dlg.dialog('open');
    });
  });
```

Мы указали, что изначально диалог должен быть скрыт, но должен открываться, когда пользователь щелкнет на кнопке с атрибутом `id="do-dialog"`. Кроме того, мы перенесли исходное текстовое содержимое диалога в область заголовка и добавили две кнопки, одна из них **добавляет текст сообщения**, а другая **стирает сообщения**. Для каждой кнопки определена своя функция, определяющая реакцию на щелчок мышью, одна из них добавляет параграф, а другая удаляет параграфы. После трех щелчков на кнопке add message (добавить сообщение), диалог будет выглядеть, как показано на рис. 10.10.

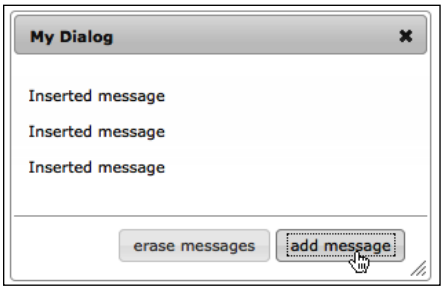


Рис. 10.10. *Вид диалога после трех щелчков на кнопке add message (добавить сообщение)*

Описание других параметров, позволяющих определять внешний вид и поведение диалогов, можно найти по адресу: *http://docs.jquery.com/UI/Dialog/dialog#options*.

jQuery UI ThemeRoller

Одним из недавних дополнений библиотеки jQuery UI является **ThemeRoller** – интерактивный веб-механизм тем для виджетов пользовательского интерфейса. Механизм ThemeRoller позволяет легко и быстро создавать тонконастраиваемые элементы с профессиональным внешним видом. Как уже отмечалось, к только что созданному диалогу мы применяем тему по умолчанию. При отсутствии дополнительных настроек эта тема будет поставляться механизмом ThemeRoller.

Для создания совершенно разных наборов стилей достаточно лишь посетить страницу *http://ui.jquery.com/themeroller/* (рис. 10.11), изменить некоторые параметры по своему желанию и щелкнуть на кнопке Download This Theme (загрузить эту тему). Полученный после этого файл `.zip`, содержащий таблицу стилей и изображений, можно поместить в требуемый каталог. Например, выбрав другие цвета и текстуры, мы за не-

сколько минут можем изменить предыдущий диалог, придав ему внешний вид, показанный на рис. 10.12.

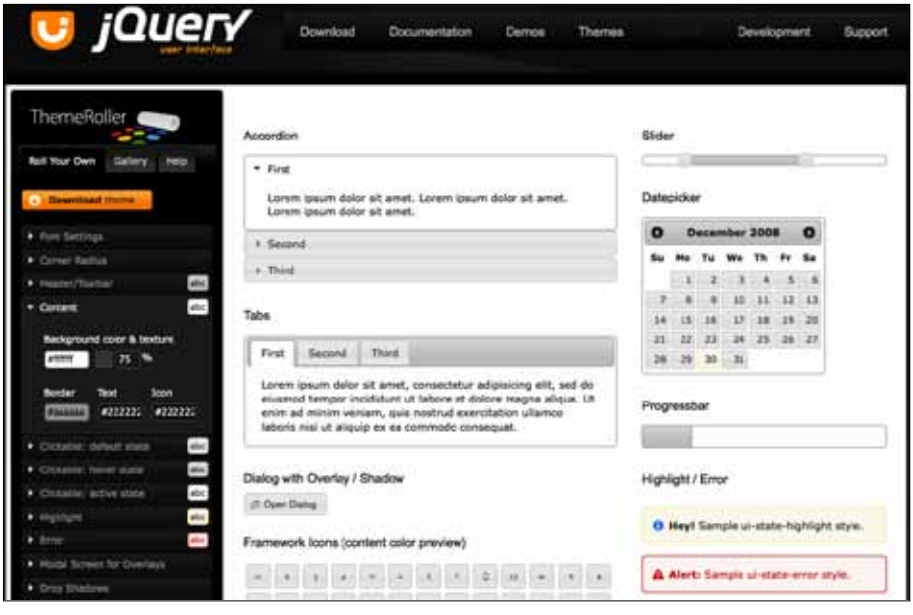


Рис. 10.11. *Домашняя страница проекта ThemeRoller*

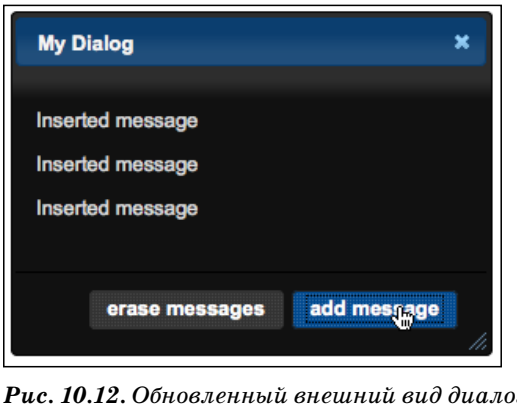


Рис. 10.12. *Обновленный внешний вид диалога*

Другие рекомендуемые расширения

В дополнение к расширениям, описанным в этой и других главах, расширения, перечисленные ниже, рекомендуются авторами не только вследствие их популярности, но и из-за надежности их реализации.

Формы

В главе 8 мы исследовали несколько способов манипулирования формами. Следующие расширения помогут упростить решение подобных задач.

Autocomplete

http://bassistance.de/jquery-plugins/jquery-plugin-autocomplete/

http://plugins.jquery.com/project/autocompletex

Расширение **Autocomplete**, написанное Йерном Заффererом (Jörn Zaefferer), одним из основных разработчиков библиотеки jQuery, обеспечивает вывод списка возможных дополнений текстовой строки по мере ее ввода пользователем, как показано на рис. 10.13.

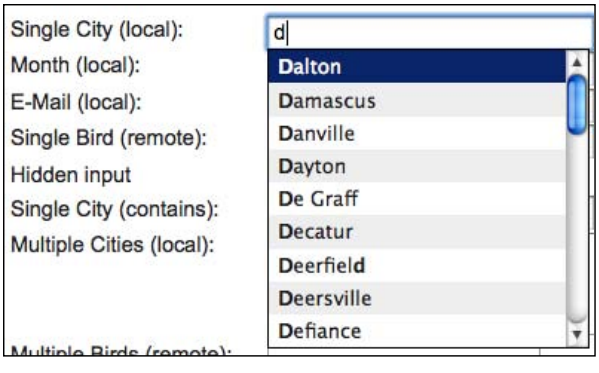


Рис. 10.13. *Расширение Autocomplete в действии*

Validation

http://bassistance.de/jquery-plugins/jquery-plugin-validation/

http://plugins.jquery.com/project/validate

Еще одно расширение Йерна Заффererа (Jörn Zaefferer), **Validation** (рис. 10.14), представляет собой чрезвычайно гибкий инструмент проверки содержимого полей ввода формы на основе разнообразных критериев.

Jeditable

http://www.appelsiini.net/projects/jeditable

http://plugins.jquery.com/project/jeditable

Расширение **Jeditable** (рис. 10.15) преобразует элементы, не являющиеся элементами форм, в поля ввода, доступные для редактирования, когда пользователь выполняет некоторое действие, такое как щелчок или двойной щелчок. Измененное содержимое элемента автоматически отправляется на сервер.

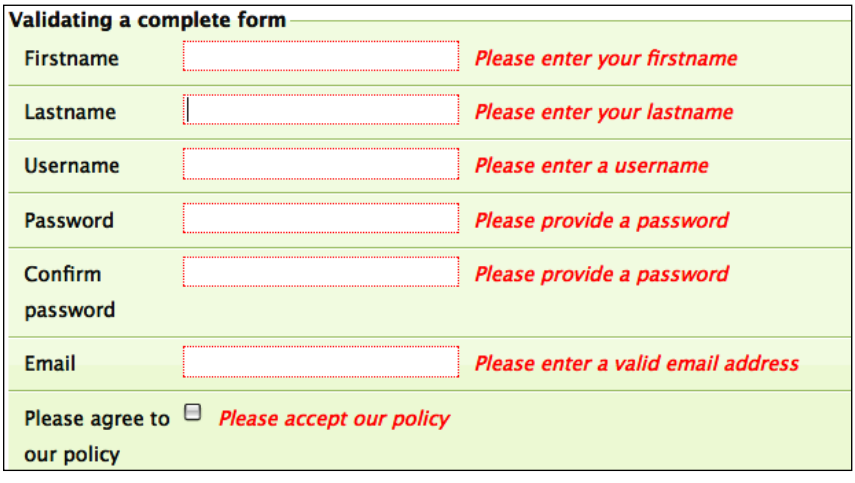


Рис. 10.14. *Расширение Validation в действии*

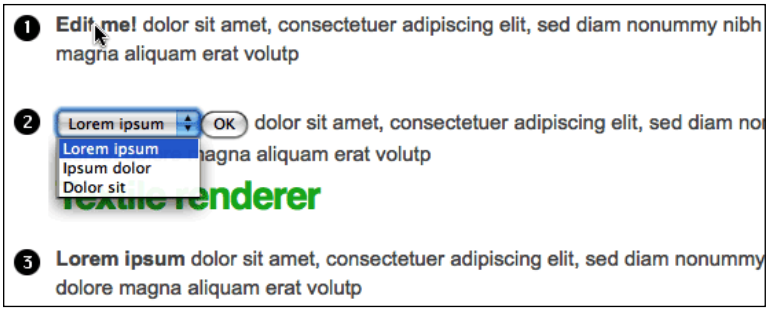


Рис. 10.15. *Расширение Jeditable в действии*

Masked input

http://digitalbush.com/projects/masked-input-plugin/

http://plugins.jquery.com/project/maskedinput

Расширение **Masked Input** (рис. 10.16) позволяет упростить для пользователя ввод таких данных, как даты, номера телефонов или карточек социального страхования, в предопределенных форматах. Расширение автоматически вставляет в поле ввода служебные символы (такие как слеш, при вводе дат) и обеспечивает возможность ввода только определенных символов, соответствующих настройкам расширения.

Таблицы

В главе 7 мы рассматривали приемы упорядочения и оформления табличных данных. Многие разработчики расширений создали свои наборы подпрограмм, чтобы помочь нам в решении этих задач.

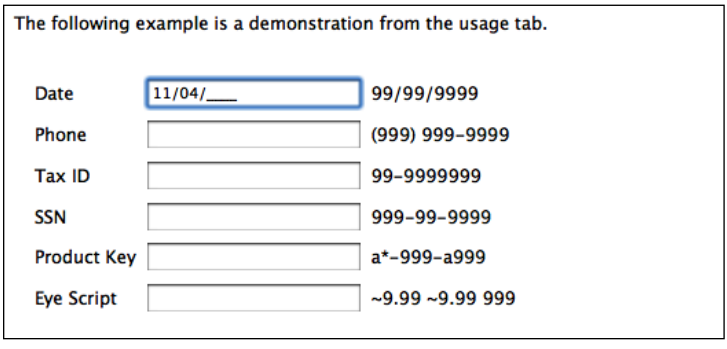


Рис. 10.16. *Расширение Masked Input в действии*

Tablesorter

http://tablesorter.com/

http://plugins.jquery.com/project/tablesorter

Расширение **Tablesorter** (рис. 10.17) может дополнить любую таблицу с элементами `<thead>` и `<tbody>` возможностью сортировки, выполнение которой не требует обновления всей страницы. В число особенностей расширения входят: возможность сортировки сразу по нескольким столбцам, возможность сортировки данных в различных форматах (таких как дата, время, денежные величины, адреса URL), вторичная «скрытая» сортировка и возможность расширения посредством системы виджетов.

Рис. 10.17. *Расширение Tablesorter в действии*

jqGrid

http://www.trirand.com/blog/

http://plugins.jquery.com/project/jqGrids

Элемент управления **jqGrid** (рис. 10.18), обладающий поддержкой AJAX, позволяет разработчикам реализовывать динамическое представление табличных данных в веб-приложениях и манипулировать ими. Он обеспечивает возможность редактирования прямо в ячейках таблицы, выполнения перехода на указанную страницу в таблице, выбора сразу несколько элементов, создания вложенных таблиц и древовидных структур. Обширную документацию к этому расширению можно найти по адресу: *http://www.secondpersonplural.ca/jqgriddocs/*.

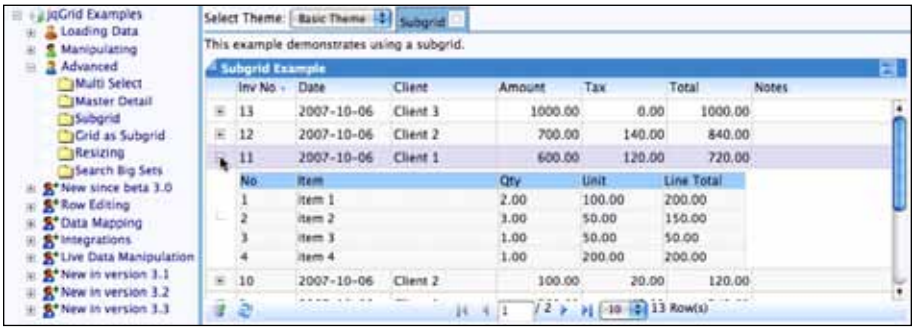


Рис. 10.18. *Элемент управления jqGrid*

Flexigrid

http://code.google.com/p/flexigrid/

http://plugins.jquery.com/project/flexigrid

Как и jqGrid, **Flexigrid** (рис. 10.19) представляет собой компонент многофункциональной таблицы. В число множества его особенностей входят: поддержка формата JSON, разбивка на страницы, функция быстрого поиска, отображение, сокрытие и изменение размеров столбцов, а также сортировка строк.

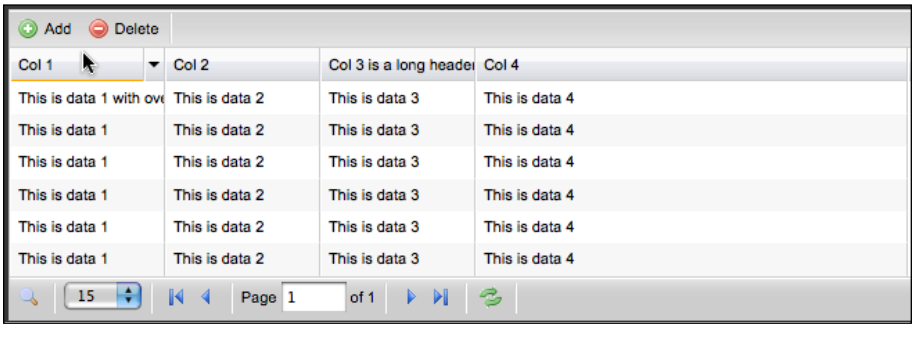


Рис. 10.19. *Элемент управления Flexgrid*

Изображения

Работа с изображениями часто требует значительных вычислительных ресурсов сервера. Однако некоторыми авторами расширений были разработаны способы выполнения некоторых простых манипуляций на языке JavaScript с использованием библиотеки jQuery.

Jcrop

http://deepliquid.com/content/Jcrop.html

http://plugins.jquery.com/project/Jcrop

Расширение **Jcrop** (рис. 10.20) предлагает простой и быстрый способ обработки изображений в веб-приложениях. В число его особенностей входят: сохранение пропорций, возможность указать минимальный и максимальный размер изображения, поддержка перемещения изображения с помощью клавиатуры, возможность подключать свои обработчики событий и определять собственные стили оформления.

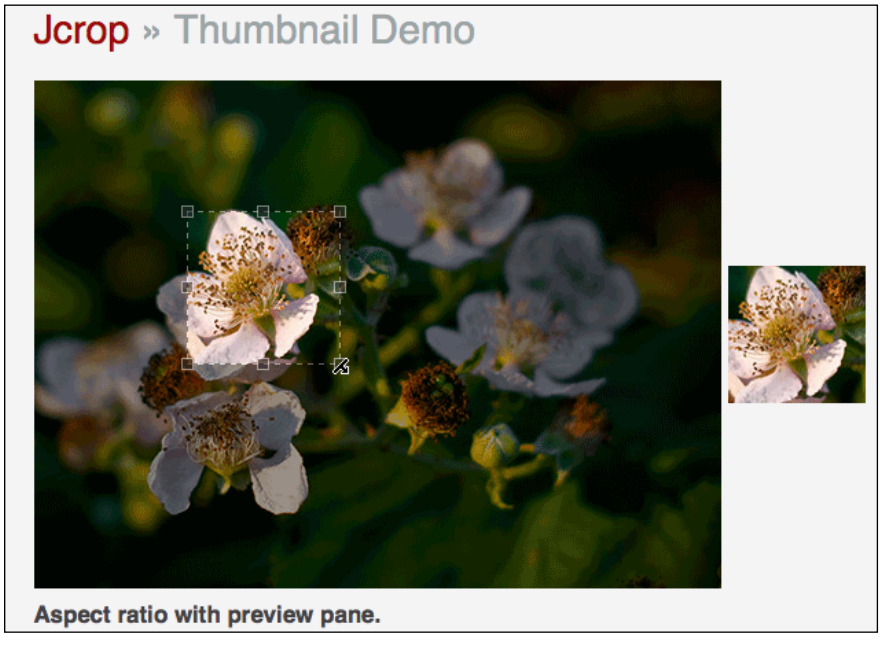


Рис. 10.20. *Расширение Jcorp в действии*

Magnify

http://www.jnathanson.com/index.cfm?page=pages/jquery/magnify/magnify

http://plugins.jquery.com/project/magnify

При наличии маленького и большого изображений с пропорциональными размерами расширение **Magnify** (рис. 10.21) создает «лупу», подобную тем, что обычно используются для создания подробных, увеличенных изображений.

Окна с подсветкой и модальные диалоги

В одном из примеров главы 9 демонстрировалась особенность, часто называемая **lightbox** (окном с подсветкой), которая позволяет отобразить дополнительную информацию поверх страницы без использования всплывающих окон. Перечисленные ниже расширения помогают создавать такие перекрытия.

Рис. 10.21. *Расширение Magnify в действии*

FancyBox

http://fancy.klade.lv/

Эта разновидность окна с подсветкой по своему оформлению напоминает внешний вид окон в Mac OS и обладает привлекательным эффектом отбрасывания тени. Помимо автоматически масштабируемых изображений расширение **FancyBox** (рис. 10.22) способно отображать содержимое строчных элементов или элемента `<iframe>`.

Рис. 10.22. *Расширение FancyBox в действии*

Thickbox

http://jquery.com/demo/thickbox/

Расширение **Thickbox** (рис. 10.23) представляет собой универсальную разновидность окна с подсветкой и способно отображать одно изображение, несколько изображений, содержимое строчных элементов, содержимое элемента `<iframe>`, а также содержимое, получаемое с использованием технологии AJAX в виде гибридного модального диалога.

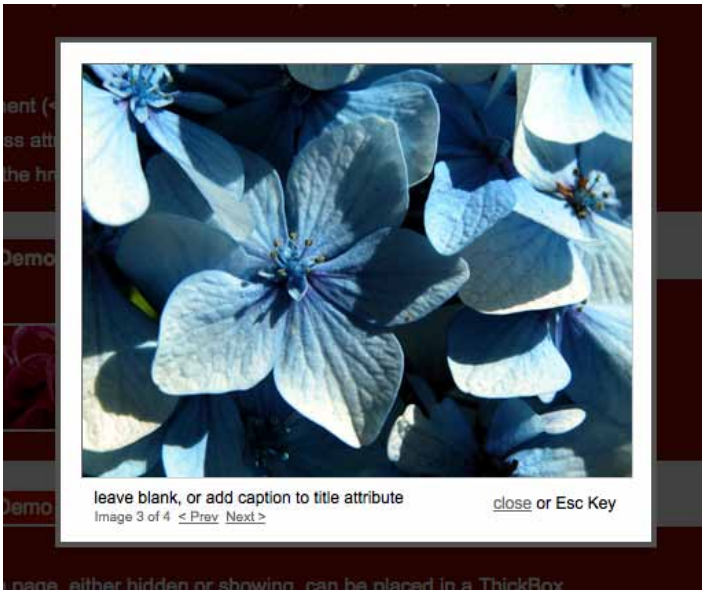


Рис. 10.23. *Расширение Thickbox в действии*

BlockUI

http://malsup.com/jquery/block/

http://plugins.jquery.com/project/blockUI

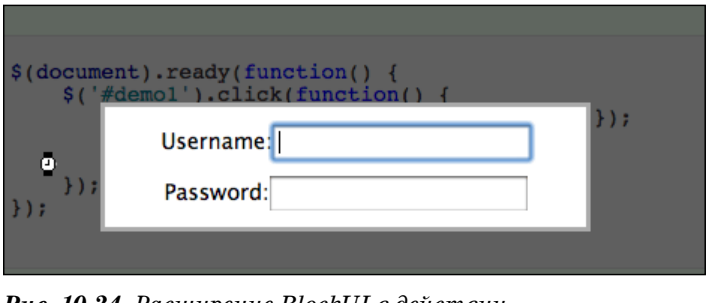


Рис. 10.24. *Расширение BlockUI в действии*

Расширение **BlockUI** (рис. 10.24) имитирует синхронное поведение, не блокируя при этом сам броузер. Активированное расширение не допускает взаимодействия пользователя со страницей (или с ее частью) до тех пор, пока не будет деактивировано.

jqModal

http://dev.iceburg.net/jquery/jqModal/

http://plugins.jquery.com/project/jqModal

Расширение **jqModal** (рис. 10.25) представляет собой легковесную реализацию модального диалога, обладающего большой гибкостью и широкими возможностями. Делая основной упор на расширяемую архитектуру, он оставляет за веб-разработчиком реализацию взаимодействий и оформление внешнего вида.

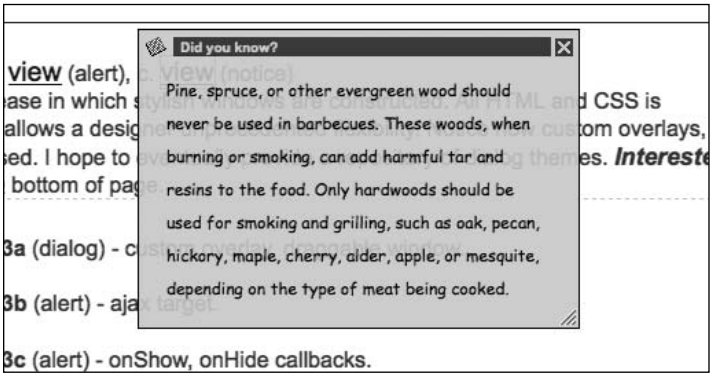


Рис. 10.25. *Расширение jqModal в действии*

Диаграммы

Как и работа с изображениями, создание диаграмм традиционно выполнялось на стороне сервера и требовало привлечения значительных вычислительных ресурсов. Изобретательные программисты разработали несколько способов создания диаграмм в броузере и реализовали эти приемы в виде расширений, перечисленных ниже.

Flot

http://code.google.com/p/flot/

http://plugins.jquery.com/project/flot

Расширение **Flot** (рис. 10.26) использует элемент `<canvas>` для создания графиков из наборов данных и дает возможность манипулировать этими графиками. При подключении дополнительного сценария **Excanvas**, преобразующего инструкции Canvas в частный формат VML, расширение Flot способно также отображать графики в Internet Explorer.

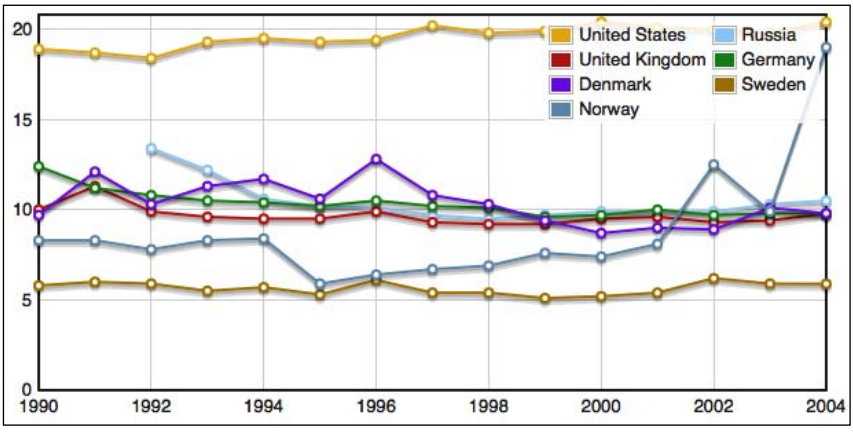


Рис. 10.26. *Расширение Flot в действии*

Sparklines

http://omnipotent.net/jquery.sparkline/

http://plugins.jquery.com/project/sparklines

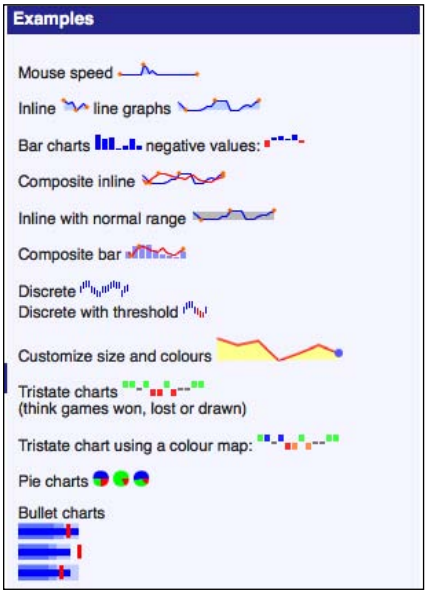


Рис. 10.27. *Расширение Sparklines в действии*

Получивший свое название в честь понятия, популяризованного специалистом в области визуализации данных Эдвардом Тафти (Edward Tufte), расширение **Sparklines** (рис. 10.27) позволяет создавать про-

стые маленькие диаграммы, встраиваемые в строку. Подобно Flot расширение Sparklines использует элемент <canvas> для отображения диаграмм, но преобразование для Internet Explorer реализовано внутри самого расширения, благодаря чему отсутствует необходимость подключать сценарий Excanvas.

События

Мы уже не раз убеждались, что библиотека jQuery обладает богатыми возможностями обработки событий, инициированных пользователем, таких как щелчок мышью или нажатие клавиши на клавиатуре. Несмотря на разнообразие вариантов, доступных в ядре библиотеки, всегда будет возникать необходимость в реализации дополнительных приемов. Следующие расширения позволяют упростить реализацию редко встречающихся сценариев обработки событий.

hoverIntent

http://cherne.net/brian/resources/jquery.hoverIntent.html

http://plugins.jquery.com/project/hoverIntent

Расширение **hoverIntent** предоставляет единственный метод, замещающий метод `.hover()`, когда важно предотвратить случайный запуск анимационного эффекта при перемещении указателя мыши над элементом. Оно пытается определить намерения пользователя, контролируя скорость перемещения указателя мыши. Это расширение особенно эффективно при реализации механизма навигации на основе раскрывающихся меню.

Live query

http://github.com/brandonaaron/livequery/

http://plugins.jquery.com/project/livequery

Подобно методу `.live()` в библиотеке jQuery расширение **Live Query** динамически подключает и поддерживает привязку обработчиков событий к элементам дерева DOM независимо от того, когда эти элементы будут созданы. Расширение предлагает альтернативную реализацию, которая может оказаться предпочтительнее в некоторых ситуациях.

В заключение

В этой главе мы исследовали включения в наши веб-страницы расширений, созданных сторонними разработчиками. Мы близко познакомились с расширениями Form и jQuery UI, а также перечислили некоторые другие расширения. В следующей главе мы воспользуемся преимуществами расширяемой архитектуры библиотеки jQuery и создадим несколько своих собственных расширений.

11

Разработка модулей расширения

Доступные сторонние расширения обеспечивают широкие возможности при создании наших сценариев, но иногда необходимо добиться чего-то большего. При написании программного кода, который могли бы применять другие разработчики или даже мы сами, может появиться потребность оформить его в виде расширения. К счастью, процесс создания расширений ненамного сложнее, чем написание программного кода, использующего его.

В этой главе мы рассмотрим, как создавать различные расширения, от простых до сложных. Начнем мы с расширений, которые реализуют новые глобальные функции, затем перейдем к реализации различных методов объекта jQuery. Кроме того, мы дополним механизм селекторов jQuery новыми выражениями и закончим некоторыми советами, как подготовить расширение к передаче другим разработчикам.

Добавление новых глобальных функций

Некоторые из встроенных возможностей библиотеки jQuery реализованы в виде так называемых **глобальных функций**. Как мы уже видели, в действительности эти функции являются **методами** объекта jQuery, но с практической точки зрения они являются функциями внутри **пространства имен** jQuery.

Яркий пример таких функций – функция `$.ajax()`. Все, что делает функция `$.ajax()`, может быть реализовано в виде обычной глобальной функции с именем `ajax()`, но в этом случае будет существовать вероятность возникновения конфликта имен с другими функциями. Если мы поместим функцию в пространство имен jQuery, нам нужно будет побеспокоиться лишь о конфликтах имен с другими методами объекта jQuery.

Чтобы добавить новую функцию в пространство имен jQuery, достаточно просто определить новую функцию как **свойство** объекта jQuery:

```
jQuery.globalFunction = function() {
```

```
  alert('This is a test. This is only a test.');
};
```

Теперь в программном коде, использующем это расширение, достаточно записать:

```
jQuery.globalFunction();
```

Кроме того, можно воспользоваться псевдонимом $ и записать:

```
$.globalFunction();
```

Этот вызов будет работать точно так же, как и предыдущий, и отобразит диалог с текстом.

Добавление нескольких функций

Если в расширении требуется реализовать более одной глобальной функции, их можно объявить независимо друг от друга:

```
jQuery.functionOne = function() {
  alert('This is a test. This is only a test.');
};
jQuery.functionTwo = function(param) {
  alert('The parameter is "' + param + '".');
};
```

Теперь, когда оба метода определены, их можно вызвать обычным способом:

```
$.functionOne();
$.functionTwo('test');
```

Для определения функций допускается также использовать альтернативный синтаксис, использующий функцию $.extend():

```
jQuery.extend({
  functionOne: function() {
    alert('This is a test. This is only a test.');
  },
  functionTwo: function(param) {
    alert('The parameter is "' + param + '".');
  }
});
```

Этот фрагмент даст тот же самый результат.

Однако здесь существует риск загрязнения пространства имен другого рода. Хотя использование пространства имен jQuery ограждает нас от конфликтов имен с большинством функций JavaScript и переменных, остается еще вероятность конфликта с именами функций, определяемыми другими расширениями. Лучший способ избежать этого – инкапсулировать все глобальные функции расширения в объект:

```
jQuery.myPlugin = {
```

```
  functionOne: function() {
    alert('This is a test. This is only a test.');
  },
  functionTwo: function(param) {
    alert('The parameter is "' + param + '".');
  }
};
```

При таком подходе по существу создается еще одно пространство имен с именем `jQuery.myPlugin`, куда помещаются глобальные функции. Несмотря на то, что мы все еще неофициально называем эти функции «глобальными», теперь они стали методами объекта `myPlugin`, который в свою очередь является свойством глобального объекта jQuery. Следовательно, мы теперь должны включать имя расширения в вызовы функций:

```
$.myPlugin.functionOne();
$.myPlugin.functionTwo('test');
```

Применяя эту методику (когда достаточно обеспечить уникальность имени расширения), мы можем полностью защитить себя от конфликтов имен с другими глобальными функциями.

Какой в этом смысл?

Итак, мы познакомились с основами разработки расширений. Сохранив наши функции в файле с именем `jquery.myplugin.js`, мы можем подключить его к странице и использовать функции в других сценариях. Но чем он отличается от любого другого файла JavaScript, который можно было бы создать и подключить к странице?

Мы уже обсуждали преимущества включения нашего программного кода внутрь объекта `jQuery`. Однако оформление нашей собственной библиотеки функций в виде расширения для jQuery дает еще одно преимущество: так как расширение предполагает подключение библиотеки jQuery, мы можем ее использовать в своих функциях.

Примечание

Даже если библиотека jQuery будет подключена, мы не должны исходить из предположения, что псевдоним `$` будет доступен. Напомним, что метод `$.noConflict()` может отдать контроль над этим псевдонимом другой библиотеке. Учитывая сказанное, наши расширения всегда должны вызывать функции из библиотеки jQuery как методы объекта `jQuery`, или сами определять псевдоним `$`, как будет описано ниже.

Создание вспомогательного метода

Многие глобальные функции, предоставляемые ядром библиотеки jQuery, являются **вспомогательными методами**, то есть они обеспечивают сокращенные способы решения задач, которые встречаются достаточно часто, но которые несложно решить вручную. Хорошими при-

мерами таких методов являются функции для работы с массивами `$.each()`, `$.map()` и `$.grep()`. Чтобы показать, как создаются вспомогательные методы, мы добавим в их число новую функцию `$.sum()`.

Наш новый метод будет принимать массив, находить сумму значений элементов этого массива и возвращать ее в виде результата. Реализация нашего расширения получается очень короткой:

```
jQuery.sum = function(array) {
  var total = 0;
  jQuery.each(array, function(index, value) {
    total += value;
  });

  return total;
};
```

Обратите внимание, что здесь для обхода элементов массива мы использовали метод `$.each()`. Мы могли бы использовать простой цикл `for()`, но, так как мы уверены, что библиотека jQuery будет подключена к странице перед нашим расширением, мы можем использовать удобные для нас методы.

Для проверки расширения мы создадим простую страницу, в которой будут отображаться входные и выходные значения функции:

```
<body>
  <p>Array contents:</p>
  <ul id="array-contents"></ul>
  <p>Array sum:</p>
  <div id="array-sum"></div>
</body>
```

Теперь напишем короткий сценарий, добавляющий в страницу значения элементов массива и их сумму.

```
$(document).ready(function() {
  var myArray = [52, 97, 0.5, -22];
  $.each(myArray, function(index, value) {
    $('#array-contents').append('<li>' + value + '</li>');
  });
  $('#array-sum').append($.sum(myArray));
});
```

Открыв страницу в броузере (рис. 11.1), можно убедиться, что наше расширение работает без ошибок.

Теперь мы знаем, что расширения обеспечивают защиту пространства имен и гарантируют доступность библиотеки jQuery. Хотя это всего лишь организационная мера. Чтобы получить в свои руки истинную мощь расширений jQuery, нам необходимо научиться создавать новые **методы** для отдельных экземпляров объекта jQuery.

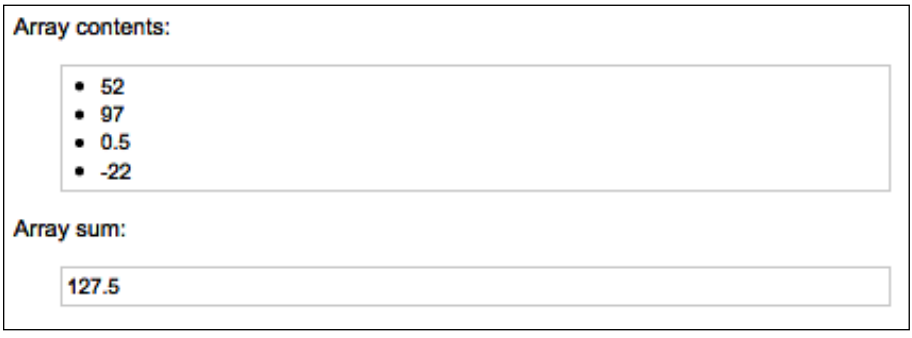


Рис. 11.1. *Результат работы расширения*

Добавление методов объекта jQuery

Большая часть функциональных возможностей jQuery предоставляется в виде методов объекта, и именно здесь расширения проявляются во всем своем блеске. Всякий раз, когда возникает необходимость написать функцию, которая оперировала бы с фрагментом дерева DOM, уместнее всего, наверное, создавать метод объекта.

Мы уже знаем, что для добавления глобальных функций требуется дополнить объект `jQuery` новыми методами. Добавление методов экземпляра производится похожим способом, только в этом случае необходимо дополнить объект `jQuery.fn`:

```
jQuery.fn.myMethod = function() {
  alert('Nothing happens.');
}
```

Примечание

Имя `jQuery.fn` – это псевдоним имени `jQuery.prototype`, предоставленный для обеспечения краткой формы записи.

В итоге мы получаем возможность вызывать новый метод из наших сценариев, используя любой селектор:

```
$('div').myMethod();
```

При вызове нашего метода будет отображен диалог с сообщением. Однако в данном случае мы могли бы оформить метод как глобальную функцию, потому что метод никак не использует узлы дерева DOM, соответствующие селектору. Разумная реализация метода должна выполнять некоторые операции с его **контекстом**.

Контекст методов объекта

Внутри любого метода расширения ключевое слово `this` ссылается на текущий экземпляр объекта jQuery. Благодаря этому мы можем вызы-

вать любые встроенные методы jQuery относительно ссылки this или извлекать из него узлы дерева DOM и работать с ними:

```
jQuery.fn.showAlert = function() {
  alert('You selected ' + this.length + ' elements.');
}
```

Чтобы исследовать, какие операции можно выполнять с контекстом объекта, мы напишем небольшое расширение, оперирующее классами элементов в наборе. Наш новый метод будет принимать два имени классов, и при каждом вызове будет заменять один класс другим в каждом элементе.

```
jQuery.fn.swapClass = function(class1, class2) {
  if (this.hasClass(class1)) {
    this.removeClass(class1).addClass(class2);
  }
  else if (this.hasClass(class2)) {
    this.removeClass(class2).addClass(class1);
  }
};
```

Сначала метод проверяет наличие класса class1 в элементе и, если он присутствует, замещает его классом class2. В противном случае проверяется наличие класса class2 и, если он присутствует, замещается классом class1. Если элемент не имеет ни одного из указанных классов, метод ничего не делает.

Чтобы проверить метод, нам потребуется разметка HTML для экспериментов:

```
<ul>
  <li>Lorem ipsum dolor sit amet</li>
  <li class="this">Consectetur adipisicing elit</li>
  <li>Sed do eiusmod tempor incididunt ut labore</li>
  <li class="that">Magna aliqua</li>
  <li class="this">Ut enim ad minim veniam</li>
  <li>Quis nostrud exercitation ullamco</li>
  <li>Laboris nisi ut aliquip ex ea commodo</li>
  <li class="that">Duis aute irure dolor</li>
</ul>
<input type="button" value="Swap classes" id="swap" />
```

Класс this обеспечивает отображение текста жирным шрифтом, а класс that – курсивным, как показано на рис. 11.2.

Теперь вызовем наш метод по щелчку на кнопке:

```
$(document).ready(function() {
  $('#swap').click(function() {
    $('li').swapClass('this', 'that');
    return false;
  });
});
```

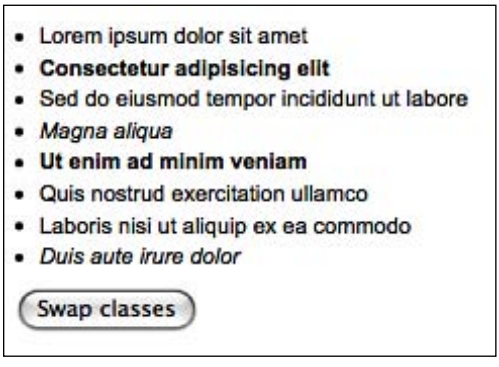


Рис. 11.2. *Страница в первоначальном виде*

Что-то пошло не так. После щелчка на кнопке ко всем строкам был применен класс `that`, как показано на рис. 11.3.

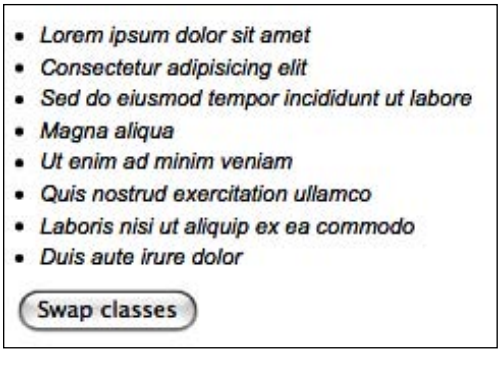


Рис. 11.3. *Вид страницы после щелчка на кнопке*

Не следует забывать, что селектор jQuery может отобрать ноль, один или более элементов. При проектировании методов расширений мы должны учитывать все возможные варианты. В данном случае мы вызвали метод `.hasClass()`, который проверил только первый элемент в множестве, а нам требуется проверить каждый элемент в отдельности и выполнить необходимые операции над ним.

Самый простой способ гарантировать корректную работу независимо от количества совпавших элементов – произвести вызов метода `.each()` в контексте метода, который выполнит **явные итерации**, что позволит обеспечить непротиворечивость взаимодействия между расширением и встроенными методами. Внутри вызова метода `.each()` ключевое слово `this` будет поочередно ссылаться на каждый элемент DOM, поэтому мы сможем откорректировать свой программный код так, чтобы он отдельно проверял наличие классов и применял их к каждому элементу в наборе.

```
jQuery.fn.swapClass = function(class1, class2) {
  this.each(function() {
    var $element = jQuery(this);
    if ($element.hasClass(class1)) {
      $element.removeClass(class1).addClass(class2);
    }
    else if ($element.hasClass(class2)) {
      $element.removeClass(class2).addClass(class1);
    }
  });
};
```

Примечание

Внимание! Внутри метода объекта ключевое слово `this` ссылается на **объект jQuery**, но внутри функции, вызываемой методом `.each()`, оно ссылается на **элемент DOM**.

Теперь после щелчка на кнопке операция замены классов не затронет элементы, не имеющие ни одного из двух классов, как показано на рис. 11.4.

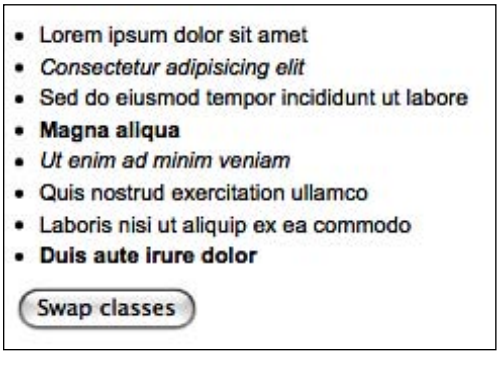


Рис. 11.4. *Теперь операция замены классов не затрагивает элементы, не имеющие ни одного из двух классов*

Объединение методов в цепочки

В дополнение к явной итерации пользователи jQuery должны иметь возможность объединять методы в **цепочки**. Это означает, что все методы расширения должны возвращать объект jQuery при условии, что они явно не предназначены для получения иной информации. Как правило, возвращаемый объект представлен ссылкой `this`. В случае использования метода `.each()` для выполнения итераций через объект `this` мы можем просто вернуть его результат:

```
jQuery.fn.swapClass = function(class1, class2) {
  return this.each(function() {
    var $element = jQuery(this);
```

```
    if ($element.hasClass(class1)) {
      $element.removeClass(class1).addClass(class2);
    }
    else if ($element.hasClass(class2)) {
      $element.removeClass(class2).addClass(class1);
    }
  });
};
```

Ранее, когда мы вызывали метод `.swapClass()`, нам приходилось писать новую инструкцию, чтобы выполнить какие-либо другие операции с элементами. Однако, добавляя инструкцию `return`, мы получаем возможность объединять метод нашего расширения в цепочки с другими встроенными методами, как показывает вид страницы на рис. 11.5.

```
$(document).ready(function() {
  $('#swap').click(function() {
    $('li')
      .swapClass('this', 'that')
      .css('text-decoration', 'underline');
    return false;
  });
});
```

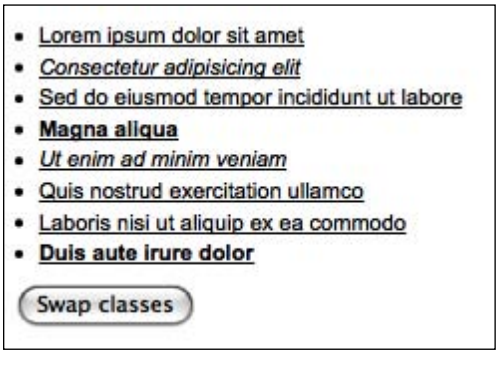


***Рис. 11.5.** Теперь метод нашего расширения может объединяться в цепочки с другими встроенными методами*

Методы обхода дерева DOM

В некоторых случаях бывает необходимо, чтобы метод расширения изменял элементы DOM, на которые ссылается объект jQuery. Например, представим, что нам требуется добавить метод, выполняющий обход дерева DOM и ищущий для элементов в наборе родителей их родителей:

```
jQuery.fn.grandparent = function() {
  var grandparents = [];
```

```
    this.each(function() {
      grandparents.push(this.parentNode.parentNode);
    });
    grandparents = jQuery.unique(grandparents);
    return this.setArray(grandparents);
  };
```

Этот метод создает новый массив `grandparents` и заполняет его, выполняя обход всех элементов, на которые ссылается текущий объект jQuery. Метод получает ссылки на предков с помощью стандартного свойства DOM `.parentNode` и помещает их в массив `grandparents`. Затем с помощью метода `$.unique()` из этого массива удаляются повторяющиеся ссылки. В заключение благодаря вызову внутреннего метода `.setArray()` библиотеки jQuery набор элементов, соответствующих селектору, замещается новым массивом. Теперь мы можем отыскивать прародителей элементов в наборе и манипулировать единственным вызовом метода.

Чтобы проверить метод в действии, подготовим глубоко вложенную структуру элементов `<div>`:

```
<div>Deserunt mollit anim id est laborum</div>
<div>Ut enim ad minim veniam
  <div>Quis nostrud exercitation
    <div>Ullamco laboris nisi
      <div>Ut aliquip ex ea</div>
      <div class="target">Commodo consequat
        <div>Lorem ipsum dolor sit amet</div>
      </div>
    </div>
  </div>
    <div>Duis aute irure dolor</div>
    <div>In reprehenderit
      <div>In voluptate</div>
      <div>Velit esse
        <div>Cillum dolore</div>
        <div class="target">Fugiat nulla pariatur</div>
      </div>
      <div>Excepteur sint occaecat cupidatat</div>
    </div>
  </div>
  <div>Non proident</div>
</div>
<div>Sunt in culpa qui officia</div>
```

Мы выделили целевые элементы (`<div class="target">`), добавив оформление текста жирным шрифтом, как показано на рис. 11.6.

Теперь можно попытаться отыскать прародителей элементов, используя новый метод:

```
$(document).ready(function() {
  $('.target').grandparent().addClass('highlight');
});
```

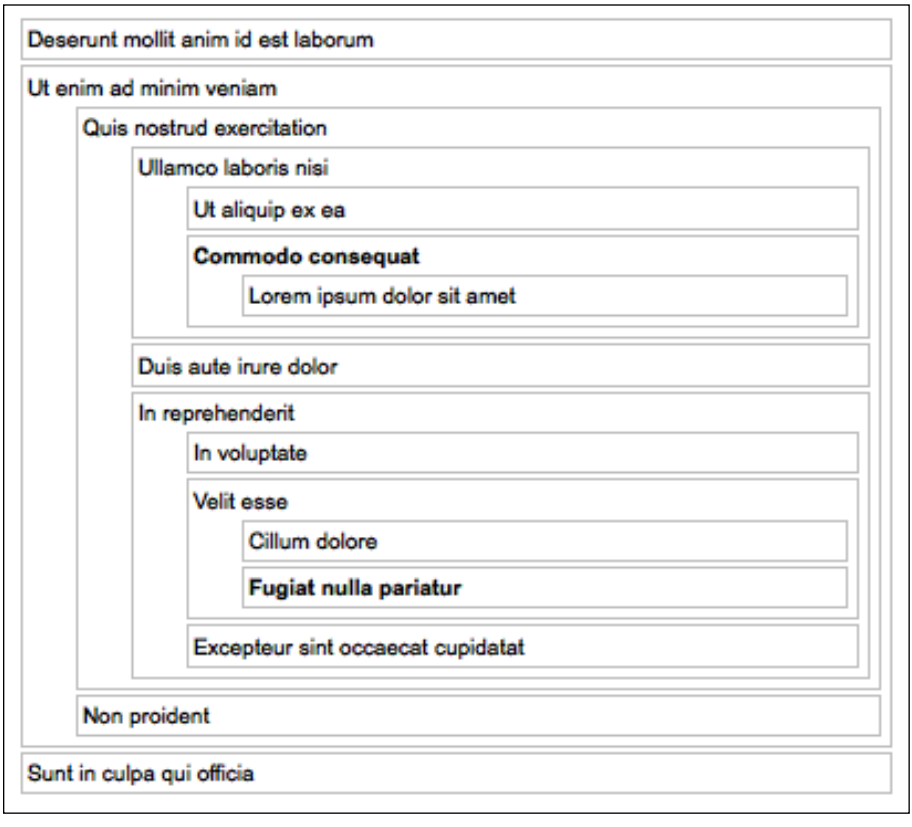


Рис. 11.6. *Целевые элементы выделены жирным шрифтом*

Добавление класса `highlight` позволяет убедиться, что оба прародителя были найдены, как показано на рис. 11.7.

Однако этот метод является **деструктивным**. Как побочный эффект происходит изменение фактического объекта jQuery, что станет очевидным, если сохранить объект jQuery в переменной:

```
$(document).ready(function() {
  var $target = $('.target');
  $target.grandparent().addClass('highlight');
  $target.hide();
});
```

Данный программный код должен выделить прародителей элементов и затем скрыть целевые элементы. Однако в действительности невидимыми становятся элементы-прародители, как показано на рис. 11.8.

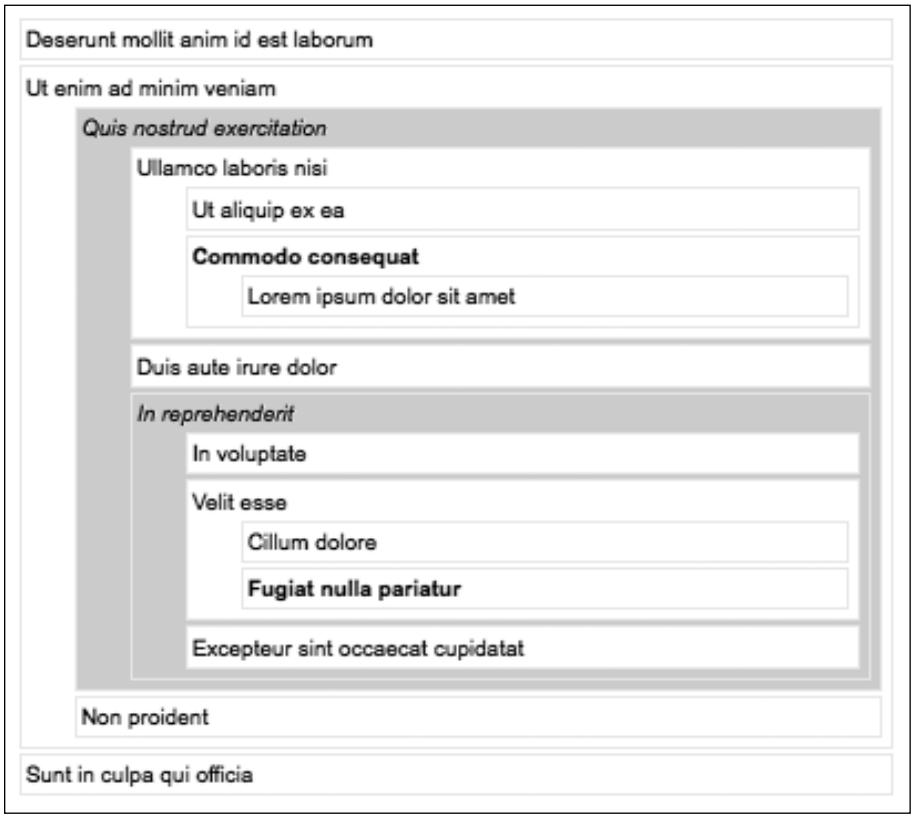


Рис. 11.7. *Были найдены оба прародителя целевых элементов*

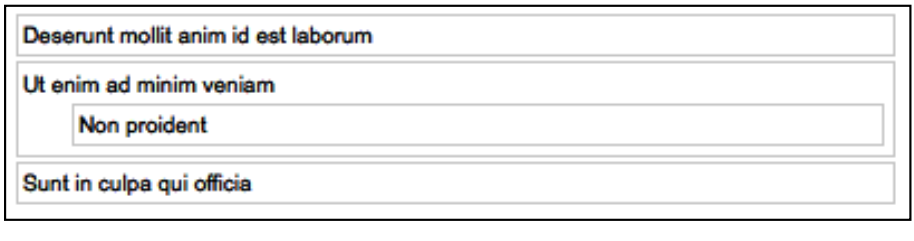


Рис. 11.8. *Невидимыми становятся прародители целевых элементов*

Объект jQuery сначала сохраняется в переменной `$target`, а затем ссылки в нем замещаются ссылками на прародителей. Во избежание этого нам необходимо сделать метод **недеструктивным;** последнее возможно осуществить, если сохранять каждый объект во внутреннем **стеке** библиотеки jQuery.

```
jQuery.fn.grandparent = function() {
  var grandparents = [];
  this.each(function() {
    grandparents.push(this.parentNode.parentNode);
  });
  grandparents = jQuery.unique(grandparents);
  return this.pushStack(grandparents);
};
```

Вызывая `.pushStack()` вместо `.setArray()`, мы создаем новый объект jQuery и не изменяем прежний. Теперь объект `$target` не модифицируется, и невидимыми становятся оригинальные целевые объекты, как показано на рис. 11.9.

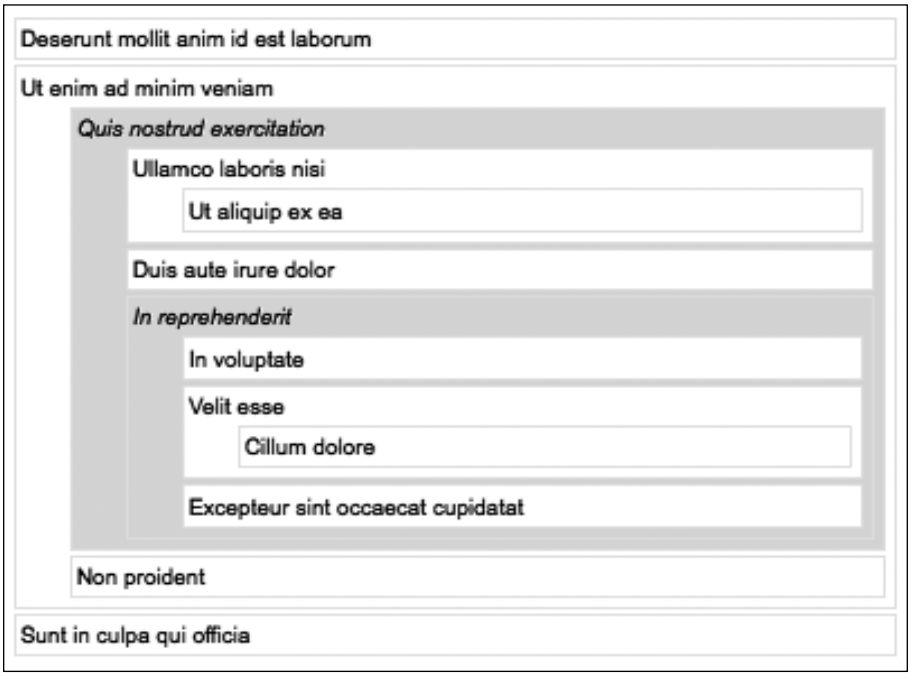


Рис. 11.9. *Теперь невидимыми становятся целевые элементы*

Дополнительно использование метода `.pushStack()` позволяет применять методы `.end()` и `.andSelf()` совместно с нашим расширением, благодаря чему мы обретаем возможность составлять цепочки методов:

```
$(document).ready(function() {
  $('.target').grandparent().andSelf().addClass('highlight');
});
```

Примечание

Методы навигации по дереву DOM, такие как `.children()`, были деструктивными операциями в версии jQuery 1.0, но в версии 1.1 они стали недеструктивными.

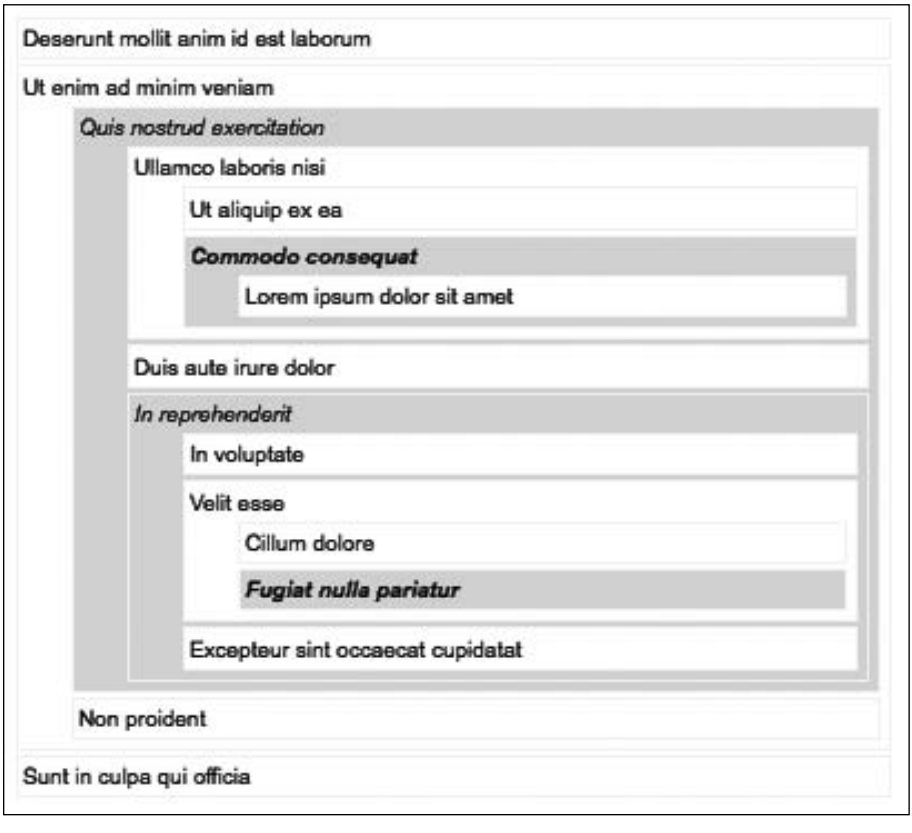


Рис. 11.10. *Использование метода .pushStack() позволяет также применять методы .end() и .andSelf()*

Добавление новых сокращенных методов

Многие из методов, включенных в состав библиотеки jQuery, являются **сокращенными версиями** других базовых методов. Например, большинство **методов подключения обработчиков событий** являются сокращенными версиями метода `.bind()` или `.trigger()`, а многие методы поддержки технологии AJAX вызывают функцию `$.ajax()`. Эти сокращенные версии делают удобным использование функциональных возможностей, работа с которыми напрямую осложнилась бы необходимостью указывать множество параметров.

Библиотека должна поддерживать тонкий баланс между удобством и сложностью. Каждый метод, добавляемый в библиотеку, не только помогает разработчикам быстрее писать определенные фрагменты программного кода, но и увеличивает общий размер кодовой базы и может привести к снижению производительности. По этой причине многие методы, реализующие сокращенную форму доступа к встроенным функциональным возможностям, оформлены в виде расширений, благодаря чему мы можем для каждого проекта отбирать необходимые расширения и отказываться от ненужных.

Обнаружение в программном коде многократного повторения некоторой идиомы может побудить к созданию сокращенного метода. Например, предположим, что нам приходится часто применять анимационный эффект, представляющий собой комбинацию встроенных эффектов свертывания и изменения прозрачности. Объединение этих эффектов означает одновременное изменение высоты и прозрачности элемента. Такой новый эффект легко можно реализовать с помощью метода `.animate()`:

```
.animate({height: 'hide', opacity: 'hide'});
```

Мы можем создать три сокращенных метода, реализующих анимационные эффекты для отображения и сокрытия элементов:

```
jQuery.fn.slideFadeOut = function() {
  return this.animate({
    height: 'hide',
    opacity: 'hide'
  });
};

jQuery.fn.slideFadeIn = function() {
  return this.animate({
    height: 'show',
    opacity: 'show'
  });
};

jQuery.fn.slideFadeToggle = function() {
  return this.animate({
    height: 'toggle',
    opacity: 'toggle'
  });
};
```

Теперь можно вызвать метод `.slideFadeOut()` и затем переключать анимационный эффект, когда в этом возникнет необходимость. Так как внутри методов расширений ключевое слово `this` ссылается на текущий объект jQuery, эффект будет применяться сразу ко всем элементам в наборе.

Для полноты наши новые методы должны принимать те же параметры, что и встроенные сокращенные методы. В частности, такие методы, как `.fadeIn()`, могут принимать значение скорости и функцию обратного вызова. Так как метод `.animate()` также принимает эти параметры, обеспечить их поддержку не представляет труда. Мы просто должны принять эти параметры и передать их методу `.animate()`.

```
jQuery.fn.slideFadeOut = function(speed, callback) {
  return this.animate({
    height: 'hide',
    opacity: 'hide'
  }, speed, callback);
};

jQuery.fn.slideFadeIn = function(speed, callback) {
  return this.animate({
    height: 'show',
    opacity: 'show'
  }, speed, callback);
};

jQuery.fn.slideFadeToggle = function(speed, callback) {
  return this.animate({
    height: 'toggle',
    opacity: 'toggle'
  }, speed, callback);
};
```

Теперь у нас имеются собственные сокращенные методы, которые действуют, как и их встроенные сородичи. Для демонстрации этих методов подготовим простую страницу HTML:

```
<body>
  <p>Lorem ipsum dolor sit amet, consectetur adipisicing
    elit, sed do eiusmod tempor incididunt ut labore et
    dolore magna aliqua. Ut enim ad minim veniam, quis
    nostrud exercitation ullamco laboris nisi ut aliquip
    ex ea commodo consequat. Duis aute irure dolor in
    reprehenderit in voluptate velit esse cillum dolore eu
    fugiat nulla pariatur. Excepteur sint occaecat
    cupidatat non proident, sunt in culpa qui officia
    deserunt mollit anim id est laborum.</p>
  <div class="controls">
    <input type="button" value="Slide and fade out"
      id="out" />
    <input type="button" value="Slide and fade in" id="in" />
    <input type="button" value="Toggle" id="toggle" />
  </div>
</body>
```

Наш сценарий будет просто вызывать новые методы, когда пользователь будет щелкать на кнопках:

```
$(document).ready(function() {
  $('#out').click(function() {
    $('p').slideFadeOut('slow');
    return false;
  });
  $('#in').click(function() {
    $('p').slideFadeIn('slow');
    return false;
  });
  $('#toggle').click(function() {
    $('p').slideFadeToggle('slow');
    return false;
  });
});
```

И анимационные эффекты будут воспроизводиться, как и ожидалось (рис. 11.11).

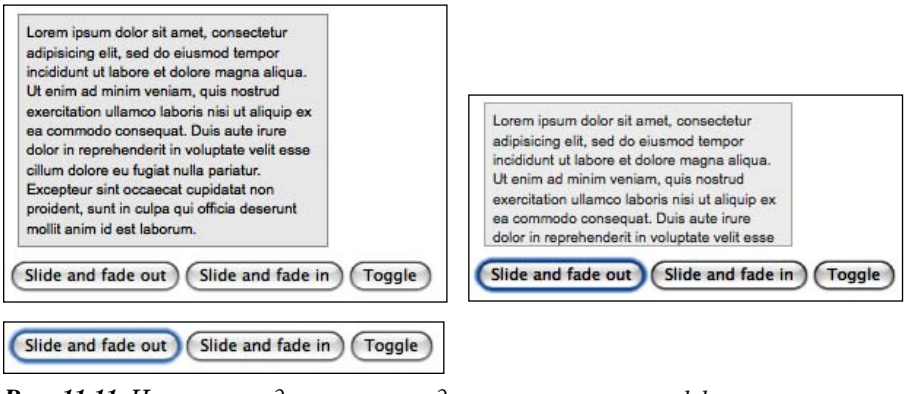


Рис. 11.11. *Новые методы воспроизводят анимационные эффекты*

Параметры методов

К настоящему моменту мы познакомились с несколькими примерами методов расширений, одни из которых принимают явные параметры, а другие – нет. Как мы уже знаем, внутри методов всегда доступно ключевое слово `this`, обеспечивающее контекст метода, но мы можем также передавать методам дополнительную информацию, чтобы оказать влияние на выполняемые им операции. До сих пор список параметров был невелик, но он может быть намного больше. Существует несколько приемов, которые можно использовать для управления параметрами нашего метода и для облегчения жизни тем, кто будет применять наши расширения.

В качестве примера создадим метод расширения, который реализует эффект отбрасывания тени блоком текста. Наш прием будет похож на эффект, применявшийся для изменения прозрачности при прокручивании новостей в главе 9: мы будем использовать несколько полупрозрачных элементов, накладывающихся друг на друга.

```
jQuery.fn.shadow = function() {
  return this.each(function() {
    var $originalElement = jQuery(this);
    for (var i = 0; i < 5; i++) {
      $originalElement
        .clone()
        .css({
          position: 'absolute',
          left: $originalElement.offset().left + i,
          top: $originalElement.offset().top + i,
          margin: 0,
          zIndex: -1,
          opacity: 0.1
        })
      .appendTo('body');
    }
  });
};
```

Данный метод создает несколько копий каждого элемента и изменяет их прозрачности. Копии позиционируются по абсолютным координатам и имеют различные смещения относительно оригинального элемента.

Как обычно, для проверки метода подготовим простую разметку HTML, которая отображается, как показано на рис. 11.12:

```
<body>
  <h1>The quick brown fox jumps over the lazy dog.</h1>
</body>
```

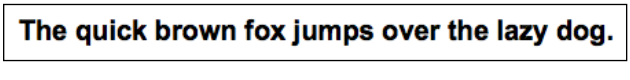


Рис. 11.12. *Первоначальный вид блока текста*

В настоящий момент наше расширение не принимает никаких параметров, поэтому вызов метода выглядит очень просто:

```
$(document).ready(function() {
  $('h1').shadow();
});
```

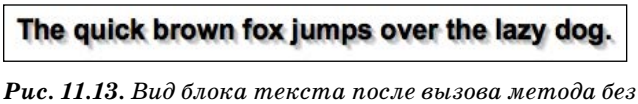


Рис. 11.13. *Вид блока текста после вызова метода без параметров*

Простые параметры

Теперь можно придать методу расширения дополнительную гибкость. В процессе работы метод опирается на несколько числовых значений, которые пользователь может пожелать изменить. Мы можем превратить эти значения в параметры, чтобы их можно было изменять в случае необходимости.

```
jQuery.fn.shadow = function(slices, opacity, zIndex) {
  return this.each(function() {
    var $originalElement = jQuery(this);
    for (var i = 0; i < slices; i++) {
      $originalElement
        .clone()
        .css({
          position: 'absolute',
          left: $originalElement.offset().left + i,
          top: $originalElement.offset().top + i,
          margin: 0,
          zIndex: zIndex,
          opacity: opacity
        })
        .appendTo('body');
    }
  });
};
```

Теперь при вызове метода мы должны указывать эти три значения.

```
$(document).ready(function() {
  $('h1').shadow(10, 0.1, -1);
});
```

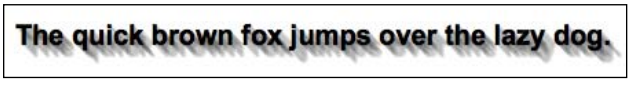


Рис. 11.14. *Вид блока текста после вызова метода с параметрами*

Наши новые параметры действуют, как и предполагалось, – тень стала длиннее за счет увеличения числа копий в два раза. Но интерфейс метода еще далек от идеала. Эти три числа легко перепутать, а порядок их следования невозможно вывести логически. Было бы совсем неплохо, если бы параметры имели подписи как для тех, кто пишет вызов метода, так и для тех, кто читает программный код.

Отображения параметров

Мы видели в библиотеке jQuery множество примеров методов, которые принимают **отображения** в качестве параметров. Они обеспечивают более дружественный для пользователя способ представления пара-

метров расширения по сравнению с простым списком аргументов. Отображение позволяет указывать имя каждого параметра, а также делает порядок следования параметров несущественным. Кроме того, отображения дают нам возможность имитировать в наших расширениях прикладной интерфейс библиотеки jQuery, последнее, в свою очередь, приводит к улучшению непротиворечивости, а, следовательно, и удобства использования.

```
jQuery.fn.shadow = function(opts) {
  return this.each(function() {
    var $originalElement = jQuery(this);
    for (var i = 0; i < opts.slices; i++) {
      $originalElement
      .clone()
        .css({
          position: 'absolute',
          left: $originalElement.offset().left + i,
          top: $originalElement.offset().top + i,
          margin: 0,
          zIndex: opts.zIndex,
          opacity: opts.opacity
        })
        .appendTo('body');
    }
  });
};
```

В нашем обновленном интерфейсе метода мы лишь изменили способ определения параметров: каждый параметр определяется не как отдельная переменная, а как свойство аргумента `opts`.

Теперь при вызове метода требуется указывать не три отдельных числа, а отображение значений:

```
$(document).ready(function() {
  $('h1').shadow({
    slices: 5,
    opacity: 0.25,
    zIndex: -1
  });
});
```

Рис. 11.15. *Вид блока текста после вызова метода с измененным интерфейсом*

Назначение каждого параметра теперь очевидно, достаточно лишь взглянуть на вызов метода.

Значения параметров по умолчанию

По мере роста числа параметров метода становится все менее вероятно, что нам всегда будет требоваться указывать значения для каждого из них. Набор разумных **значений по умолчанию** может сделать интерфейс расширения более удобным. К счастью, использование отображений для передачи параметров помогает решить и эту задачу – достаточно просто заменить любые отсутствующие элементы их значениями по умолчанию.

```
jQuery.fn.shadow = function(options) {
  var defaults = {
    slices: 5,
    opacity: 0.1,
    zIndex: -1
  };
  var opts = jQuery.extend(defaults, options);
  return this.each(function() {
    var $originalElement = jQuery(this);
    for (var i = 0; i < opts.slices; i++) {
      $originalElement
        .clone()
        .css({
          position: 'absolute',
          left: $originalElement.offset().left + i,
          top: $originalElement.offset().top + i,
          margin: 0,
          zIndex: opts.zIndex,
          opacity: opts.opacity
        })
        .appendTo('body');
    }
  });
};
```

Здесь внутри метода мы определили еще одно отображение с именем `defaults`. Вспомогательная функция `$.extend()` позволяет взять отображение `options`, полученное как аргумент, и использовать его для переопределения значений по умолчанию в отображении `defaults` без изменения при этом отсутствующих элементов.

Мы по-прежнему передаем отображение нашему методу, но теперь мы можем указывать только те параметры, значения которых должны отличаться от значений по умолчанию:

```
$(document).ready(function() {
  $('h1').shadow({
    opacity: 0.05
  });
});
```

Рис. 11.16. *Вид блока текста после вызова метода с неполным набором параметров*

Параметры, отсутствующие в вызове метода, получают значения по умолчанию. Метод `$.extend()` способен принимать даже пустые значения, поэтому, если значения по умолчанию вполне устраивают, наш метод может вызываться очень просто без возникновения ошибок:

```
$(document).ready(function() {
  $('h1').shadow();
});
```

Функции обратного вызова

Безусловно, некоторые параметры методов могут быть куда сложнее, чем простые числовые значения. Одним из распространенных типов параметров, которые нам приходилось видеть в библиотеке jQuery, является **функция обратного вызова**. Функции обратного вызова могут существенно увеличивать гибкость расширения, не требуя значительных усилий при разработке расширения.

Чтобы задействовать функцию обратного вызова в нашем методе, нужно просто принять объект функции в виде параметра и вызвать эту функцию внутри реализации метода. В качестве примера расширим наш метод реализации эффекта отбрасывания тени и дадим пользователю возможность определять смещение тени относительно текста.

```
jQuery.fn.shadow = function(options) {
  var defaults = {
    slices: 5,
    opacity: 0.1,
    zIndex: -1,
    sliceOffset: function(i) {
      return {x: i, y: i};
    }
  };
  var opts = jQuery.extend(defaults, options);

  return this.each(function() {
    var $originalElement = jQuery(this);
    for (var i = 0; i < opts.slices; i++) {
      var offset = opts.sliceOffset(i);
      $originalElement
        .clone()
        .css({
          position: 'absolute',
          left: $originalElement.offset().left
                + offset.x,
```

```
            top: $originalElement.offset().top
                 + offset.y,
            margin: 0,
            zIndex: opts.zIndex,
            opacity: opts.opacity
          })
          .appendTo('body');
      }
    });
};
```

Каждая копия текста, образующая тень, имеет собственное смещение относительно оригинального текста. Ранее это смещение просто принималось равным значению индекса копии. А теперь смещение вычисляется с помощью функции `sliceOffset()`, которая может передаваться пользователем в виде параметра. Так, например, мы можем определить отрицательные значения по обеим осям координат:

```
$(document).ready(function() {
  $('h1').shadow({
    sliceOffset: function(i) {
      return {x: -i, y: -2*i};
    }
  });
});
```

В результате текст будет отбрасывать тень влево и вверх, как показано на рис. 11.17.

Рис. 11.17. *Текст будет отбрасывать тень влево и вверх*

Функция позволяет реализовать не только простое изменение направления отбрасывания тени, но и производить более сложное позиционирование, если пользователь расширения предоставит соответствующую функцию обратного вызова. Если функция обратного вызова не будет указана, будет использована реализация поведения по умолчанию.

Настраиваемые значения по умолчанию

Мы можем повысить удобство использования нашего расширения, предусмотрев разумные значения по умолчанию для параметров метода, как только что было показано. Однако, иногда бывает сложно предсказать, какие значения по умолчанию будут достаточно разумными. Если в сценарии придется многократно вызывать метод нашего расширения со значениями параметров, отличными от значений по умолчанию, то возможность изменить эти значения по умолчанию могла бы существенно уменьшить объем программного кода.

Чтобы сделать значения по умолчанию настраиваемыми, необходимо вынести их за рамки метода, в место, где они будут доступны внешнему программному коду:

```
jQuery.fn.shadow = function(options) {
  var opts = jQuery.extend({},
    jQuery.fn.shadow.defaults, options);

  return this.each(function() {
    var $originalElement = jQuery(this);
    for (var i = 0; i < opts.slices; i++) {
      var offset = opts.sliceOffset(i);
      $originalElement
        .clone()
        .css({
          position: 'absolute',
          left: $originalElement.offset().left + offset.x,
          top: $originalElement.offset().top + offset.y,
          margin: 0,
          zIndex: opts.zIndex,
          opacity: opts.opacity
        })
        .appendTo('body');
    }
  });
};

jQuery.fn.shadow.defaults = {
  slices: 5,
  opacity: 0.1,
  zIndex: -1,
  sliceOffset: function(i) {
    return {x: i, y: i};
  }
};
```

Теперь значения по умолчанию определяются в **пространстве имен** расширения `shadow` и доступны как элементы отображения `$.fn.shadow.defaults`. Нам пришлось также исправить вызов метода `$.extend()`, чтобы учесть эти изменения. Поскольку теперь мы предусматриваем возможность многократного использования одного и того же отображения со значениями по умолчанию при каждом обращении к методу `.shadow()`, мы не можем позволить функции `$.extend()` изменять его. Вследствие этого мы передаем методу `$.extend()` пустое отображение `{}` в первом аргументе, и именно этот новый объект изменяется.

Теперь программный код, использующий наше расширение, может изменять значения по умолчанию, которые будут использоваться во всех последующих вызовах метода `.shadow()`. Кроме того, сохраняется возможность переопределять значения параметров при вызове метода.

```
$(document).ready(function() {
  $.fn.shadow.defaults.slices = 10;
  $('h1').shadow({
    sliceOffset: function(i) {
      return {x: -i, y: i};
    }
  });
});
```

Данный сценарий создаст тень из 10 копий, так как это новое значение по умолчанию, а сама тень будет отбрасываться влево и вниз (рис. 11.18) благодаря функции обратного вызова `sliceOffset()`, которая передается методу.

Рис. 11.18. *Текст отбрасывает тень влево и вниз благодаря функции обратного вызова*

Добавление селекторных выражений

Встроенные части библиотеки jQuery также допускают возможность расширения. Вместо того чтобы создавать новые методы, мы можем настраивать под себя существующие. Например, достаточно часто появляется желание расширить **селекторные выражения**, предоставляемые библиотекой jQuery, с целью обеспечения дополнительных, необычных возможностей.

Самое простое в реализации селекторное выражение – **псевдокласс**. Псевдоклассы – это выражения, начинающиеся с двоеточия, такие как `:checked` или `:nth-child()`. Для демонстрации процесса создания селекторного выражения мы реализуем псевдокласс с именем `:css()`. Этот новый селектор позволит нам отыскивать элементы по числовым значениям их атрибутов CSS.

Когда для поиска элементов применяется селекторное выражение, jQuery пытается отыскать соответствующую ему инструкцию во внутреннем отображении с именем `expr`. Это отображение содержит программный код JavaScript, который выполняется для каждого элемента страницы. Если указанный программный код возвращает значение `true`, элемент помещается в набор, возвращаемый в качестве результата. Мы можем добавить новое выражение в это отображение с помощью функции `$.extend()`.

```
jQuery.extend(jQuery.expr[':'], {
  'css': function(element, index, matches, set) {
    var parts = /([\w-]+)\s*([<>=]+)\s*(\d+)/
      .exec(matches[3]);
    var value = parseFloat(jQuery(element).css(parts[1]));
```

```
      switch (parts[2]) {
        case '<':
          return value < parseInt(parts[3]);
        case '<=':
          return value <= parseInt(parts[3]);
        case '=':
        case '==':
          return value == parseInt(parts[3]);
        case '>=':
          return value >= parseInt(parts[3]);
        case '>':
          return value > parseInt(parts[3]);
      }
    }
});
```

Этот программный код сообщает библиотеке jQuery, что строка `css`, следующая за двоеточием, является допустимой и может присутствовать в селекторном выражении, и при ее использовании следует вызывать указанную функцию, чтобы определить, следует ли включать элемент в возвращаемый набор.

Функции передаются четыре параметра:

- `element`: текущий рассматриваемый элемент DOM. Этот параметр требуется большинству селекторов.
- `index`: индекс элемента DOM в возвращаемом наборе. Этот параметр используется такими селекторами, как `:eq()` и `:lt()`.
- `matches`: массив, содержащий результат применения регулярного выражения, использованного для анализа селектора. Обычно используется только элемент `matches[3]`; в случае применения селекторов вида `:a(b)` элемент `matches[3]` содержит `b`, то есть текст в круглых скобках.
- `set`: полное множество всех элементов DOM, для которых было выявлено соответствие данному селектору к настоящему моменту. Этот параметр редко используется на практике.

Селекторы псевдоклассов могут использовать информацию, содержащуюся в этих четырех параметрах, чтобы определить принадлежность элемента к возвращаемому набору. В данном случае нам потребуются только параметры `element` и `matches`.

В функции реализации селектора мы сначала разбиваем селектор на фрагменты для дальнейшего использования с помощью регулярного выражения. Нам требуется, чтобы селектор, такой как `:css(width < 200)`, возвращал все элементы, в которых значение атрибута `width` меньше `200`. Поэтому нам необходимо отыскать в тексте, содержащемся в круглых скобках, требуемое имя свойства (`width`), оператор сравнения (`<`) и значение, с которым производится сравнение (`200`). Регулярное выражение

/([\w-]+)\s*([<>=]+)\s*(\d+)/ отыскивает эти три фрагмента строки и помещает их в массив.

Затем нам требуется извлечь текущее значение свойства. Мы можем использовать метод .css() из библиотеки jQuery, чтобы получить значение свойства, указанного в селекторе. Так как значение свойства возвращается в виде строки, мы используем функцию parseFloat(), чтобы преобразовать его в число.

В заключение выполняется фактическая операция сравнения. Инструкция switch определяет тип сравнения по содержимому селектора и возвращает результат операции (true или false).

Теперь в нашем распоряжении имеется новый селектор, который можно использовать в нашем программном коде jQuery. Действие этого селектора можно продемонстрировать с помощью простого документа HTML, первоначальный внешний вид которого приводится на рис. 11.19.

```
<body>
  <div>Deserunt mollit anim id est laborum</div>
  <div>Ullamco</div>
  <div>Ut enim ad minim veniam laboris</div>
  <div>Quis nostrud exercitation consequat nisi</div>
  <div>Ut aliquip</div>
  <div>Commodo</div>
  <div>Lorem ipsum dolor sit amet ex ea</div>
</body>
```

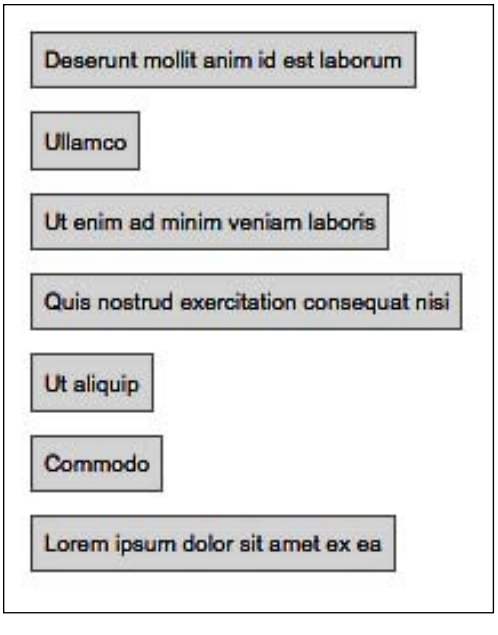


Рис. 11.19. *Первоначальный вид документа*

Теперь благодаря нашему новому селектору задача выделения наименьших элементов (рис. 11.20) решается тривиально просто:

```
$(document).ready(function() {
  $('div:css(width < 100)').addClass('highlight');
});
```

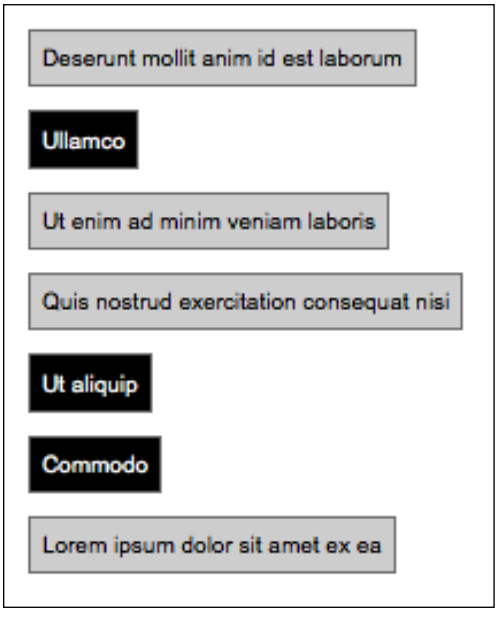


Рис. 11.20. *Задача выделения наименьших элементов решается тривиально просто*

Подготовка расширения к распространению

Если разработка расширения завершена, у нас может появиться желание предложить его другим разработчикам, чтобы они могли применять его с пользой для себя или, возможно, улучшить его. Сделать это можно на сайте официального репозитория **jQuery Plugin Repository**, по адресу: *http://plugins.jquery.com/*. Здесь мы можем выполнить вход, зарегистрировавшись, если необходимо, затем, следуя инструкциям, описать расширение и выгрузить архив в формате `.zip` с программным кодом. Однако прежде мы должны убедиться, что расширение как следует отполировано и готово для общего использования.

Ниже приводятся несколько правил, которым необходимо следовать при создании расширений для обеспечения их совместимости с другим программным кодом. Мы мимоходом уже рассматривали некоторые из них, но ради удобства приведем их еще раз.

Соглашения об именовании

Всем расширениям должны присваиваться имена вида `jQuery.myPlugin.js`, где `myPlugin` – это имя расширения. Внутри файла все глобальные функции должны быть сгруппированы в объекте с именем `jQuery.myPlugin`, если это не единственная функция, в противном случае ей можно присвоить имя `jQuery.myPlugin()`.

К именам методов предъявляются менее жесткие требования, но в любом случае они должны быть как можно более уникальными. Если определен только один метод, ему должно быть присвоено имя `jQuery.fn.myPlugin()`. Если в расширении определено более одного метода, следует постараться давать методам имена, содержащие имя расширения в виде префикса, чтобы предотвратить возможный беспорядок. Не используйте короткие имена для методов, такие как `.load()` или `.get()`, так как это может привести к путанице с методами в других расширениях.

Использование псевдонима $

В расширениях для библиотеки jQuery не следует полагать, что будет доступен псевдоним `$`. Вместо него всякий раз следует использовать полное имя `jQuery`.

Многие разработчики отмечают, что при разработке больших расширений отказ от использования псевдонима `$` делает программный код более трудным для чтения. Чтобы преодолеть этот недостаток, можно определить локальный псевдоним, который будет действовать только в области видимости расширения, определив и выполнив функцию. Одновременное определение и вызов функции выглядит, как показано ниже:

```
(function($) {
  // Здесь находится программный код
})(jQuery);
```

Объемлющая функция принимает единственный аргумент, в котором передается ссылка на глобальный объект `jQuery`. Аргументу присвоено имя `$`, поэтому внутри функции мы можем использовать псевдоним `$` без опасения вступить в конфликт.

Интерфейсы методов

Все методы в библиотеке jQuery вызываются в контексте объекта jQuery, поэтому ключевое слово `this` ссылается на данный объект, который в свою очередь может ссылаться на один или более элементов DOM. Все методы должны вести себя корректно независимо от фактического количества элементов в наборе. Вообще говоря, для обхода набора элементов и их обработки методы должны вызывать `this.each()`.

Методы должны возвращать объект jQuery, чтобы обеспечить возможность составления цепочек. Если набор объектов, соответствующих се-

лектору, изменяется, следует создавать новый объект вызовом метода `.pushStack()` и возвращать его вызывающему программному коду. Если метод возвращает что-то, отличное от объекта jQuery, это должно быть явно отмечено в документации.

Если методы принимают несколько параметров, для их передачи предпочтительнее использовать отображение, чтобы параметры можно было указывать в любом порядке и с их именами. Значения по умолчанию должны определяться в отображении, которое можно переопределить в случае необходимости.

Определения методов должны заканчиваться символом точки с запятой (;), чтобы утилиты сжатия программного кода могли корректно выполнять парсинг файлов. Кроме того, расширения могут начинаться с точки с запятой, чтобы другие, не так тщательно оформленные сценарии не приводили к конфликтам после сжатия.

Оформление документации

В файле перед определением каждой функции или метода должно находиться соответственно ее или его описание в формате ScriptDoc. Описание этого формата можно найти по адресу: *http://www.scriptdoc.org/*.

В заключение

В этой последней главе мы показали, что функциональность, присутствующая в ядре библиотеки jQuery, не ограничивает возможности библиотеки. Легкодоступные расширения существенно раздвигают границы возможностей, но мы сами без труда можем создавать собственные расширения, которые раздвинут эти границы еще шире.

Созданные нами расширения содержат различные особенности, включая глобальные функции, использующие библиотеку jQuery, новые методы объекта jQuery, оперирующие элементами дерева DOM, расширяемые методы, которые легко могут быть настроены, и дополнительные селекторные выражения, позволяющие отбирать элементы DOM новыми способами.

Обладая всеми этими инструментами, мы можем придавать библиотеке jQuery и нашему собственному программному коду JavaScript любую форму, какую только пожелаем.

A

Ресурсы в Интернете

Перечисленные ниже ресурсы в Интернете являются неплохой отправной точкой в поисках дополнительной информации о библиотеке jQuery, JavaScript и о разработке веб-приложений вообще. В Интернете существует слишком много источников качественной информации, чтобы можно было в этом приложении привести хоть что-то, отдаленно напоминающее исчерпывающий список. Более того, здесь не упоминаются многие печатные издания, которые также могут содержать ценную информацию.

Документация к библиотеке jQuery

Эти ресурсы предлагают краткие справочники и подробное описание самой библиотеки jQuery.

jQuery wiki

На сайте *jquery.com* имеется документация в формате wiki, то есть ее содержимое доступно для редактирования. Эта документация содержит полное описание jQuery API, учебные руководства, руководства для начинающих и многое другое:

http://docs.jquery.com/

jQuery API

Помимо официальной документации на сайте *jquery.com* описание прикладного программного интерфейса библиотеки jQuery можно найти по адресу:

http://remysharp.com/jquery-api/

Броузер по функциям и методам jQuery API

Йерн Заферрер (*Jörn Zaeferrer*) объединил удобный броузер дерева прикладного программного интерфейса jQuery с возможностью поиска и сортировкой по алфавиту и категориям:

http://jquery.bassistance.de/api-browser-1.2/

Visual jQuery

Этот красивый и удобный броузер прикладного программного интерфейса был создан *Иегудой Кацем* (*Yehuda Katz*) и дополнен *Реми Шарпом* (*Remy Sharp*). Кроме всего прочего он предоставляет краткий обзор методов некоторых расширений к библиотеке jQuery:

http://www.visualjquery.com/

Обозреватель Adobe AIR jQueryAPI

Реми Шарп (*Remy Sharp*) подготовил описание jQuery API в виде приложения Adobe AIR для просмотра в автономном режиме:

http://remysharp.com/downloads/jquery-api-browser.air.zip

Справочники по JavaScript

Следующие сайты предоставляют справочники и руководства по JavaScript скорее как по языку программирования вообще, чем как по библиотеке jQuery в частности.

Центр разработчиков Mozilla

На этом сайте имеются исчерпывающий справочник по JavaScript, руководство по программированию на языке JavaScript, ссылки на различные инструменты и многое другое:

http://developer.mozilla.org/en/docs/JavaScript/

Dev.opera

Несмотря на то, что сайт компании Opera для веб-разработчиков посвящен в основном собственному броузеру, он содержит значительное число полезных статей о JavaScript:

http://dev.opera.com/articles/

Справочник MSDN JScript

Справочник Microsoft Developer Network JScript Reference содержит описание полного набора функций, объектов и прочего. Он особенно поле-

зен тем, кто стремится разобраться в реализации стандарта ECMAScript в броузере Internet Explorer компании Microsoft:

http://msdn.microsoft.com/en-us/library/x85xxsf4(VS.71).aspx

Quirksmode

Сайт Quirksmode *Петера-Пауля Коха* (*Peter-Paul Koch*) является потрясающим ресурсом, содержащим информацию о различиях реализаций функций JavaScript и многих свойств CSS в разных броузерах:

http://www.quirksmode.org/

JavaScript Toolbox

Сайт JavaScript Toolbox *Мэтта Круза* (*Matt Kruse*) предлагает огромный выбор простых библиотек JavaScript, а также разумные советы по наиболее удачным приемам программирования на JavaScript и коллекцию ссылок на ресурсы в Интернете, посвященные JavaScript:

http://www.javascripttoolbox.com/

Компрессоры программного кода JavaScript

По окончании разработки сайта часто бывает желательно сжать программный код JavaScript. Это обеспечивает уменьшение времени загрузки страницы для всех пользователей сайта.

YUI Compressor

Данный компрессор программного кода JavaScript входит в состав библиотеки Yahoo! UI Library и может использоваться для сжатия исходного программного кода jQuery. Этот инструмент командной строки написан на языке Java и доступен для бесплатной загрузки. Получающийся программный код обладает высокой производительностью и небольшим размером, причем он может быть сжат еще больше с применением алгоритма сжатия Gzip.

http://developer.yahoo.com/yui/compressor/

JSMin

Утилита JSMin, созданная *Дугласом Крокфордом* (*Douglas Crockford*), представляет собой фильтр, который удаляет комментарии и лишние пробелы из файлов JavaScript. Обычно он позволяет уменьшить размер файла в два раза, что приводит к уменьшению времени загрузки, особенно когда используется в комбинации со средствами сжатия файлов на стороне сервера:

http://www.crockford.com/javascript/jsmin.html

Pretty printer

Этот инструмент выполняет форматирование ранее сжатых файлов JavaScript, восстанавливая по мере возможности разрывы строк и отступы. Он принимает ряд параметров, с помощью которых можно влиять на получаемый результат:

http://www.prettyprinter.de/

Справочник по (X)HTML

Библиотека jQuery лучше всего проявляет себя при работе с корректно оформленными документами HTML и XHTML. Ресурс, представленный ниже, окажет помощь в освоении этих языков разметки.

Домашняя страница языка разметки гипертекста консорциума W3C

Консорциум World Wide Web Consortium (W3C) определил стандарт (X) HTML, и домашняя страница HTML является отличной отправной точкой в поисках спецификаций и руководств по этому языку разметки:

http://www.w3.org/MarkUp/

Справочники по CSS

Различные визуальные и анимационные эффекты, с которыми мы не раз сталкивались в этой книге, опираются на использование каскадных таблиц стилей (Cascading Stylesheets). Желающие внедрить визуальные эффекты в свои сайты обязательно должны обратиться к руководствам по CSS.

Домашняя страница каскадных таблиц стилей W3C

На домашней странице CSS консорциума W3C приводится множество ссылок на учебные руководства, спецификации, наборы тестов и другие ресурсы:

http://www.w3.org/Style/CSS/

Mezzoblue CSS cribsheet

Дэйв Шеа (*Dave Shea*) предоставляет полезную шпаргалку по CSS для упрощения процесса проектирования страниц, а также краткое справочное руководство, к которому можно обращаться в случае появления каких-либо проблем:

http://mezzoblue.com/css/cribsheet/

Position is everything

Этот сайт содержит подборку ошибок реализации CSS в броузерах с описанием, как их обойти:

http://www.positioniseverything.net/

Полезные блоги

Для любой технологии, продолжающей свое развитие, постоянно разрабатываются новые приемы и возможности. Оставаться в курсе последних нововведений совсем не сложно, достаточно просто время от времени посещать ресурсы, посвященные новостям веб-разработки.

Блог jQuery

Джон Резиг (*John Resig*) и другие участники официального блога jQuery публикуют объявления о новых версиях и других инициативах участников проекта, а также учебные руководства и редакционные статьи.

http://jquery.com/blog/

Learning jQuery

Карл Шведберг (*Karl Swedberg*) создал этот блог для публикации учебных руководств по jQuery, приемам программирования и для размещения объявлений. В число приглашенных авторов входят члены коллектива разработчиков jQuery *Майк Алсуп* (*Mike Alsup*) и *Брендон Аарон* (*Brandon Aaron*):

http://www.learningjquery.com/

Ajaxian

Этот часто обновляемый блог, поддерживаемый *Дионом Алмаером* (*Dion Almaer*) и *Беном Галбрайтом* (*Ben Galbraith*), публикует огромное число новостей и статей, посвященных JavaScript:

http://ajaxian.com/

Блог Джона Резига (John Resig)

В своем персональном блоге *Джон Резиг* (*John Resig*), создатель библиотеки jQuery, обсуждает передовые вопросы использования JavaScript:

http://ejohn.org/

JavaScript ant

Этот сайт содержит каталог статей, имеющих отношение к языку программирования JavaScript и его использованию в современных веб-

броузерах, а также список ресурсов в Интернете о JavaScript, разбитый на категории:

http://javascriptant.com/

Блог Роберта Наймана (Robert Nyman)

На этом сайте *Роберт Найман* (*Robert Nyman*) обсуждает вопросы разработки интернет-приложений и, в частности, проблемы создания сценариев, выполняющихся на стороне клиента:

http://www.robertnyman.com/

О веб-стандартах с фантазией

Блог *Дастина Диаза* (*Dustin Diaz*) содержит статьи, посвященные веб-дизайну и веб-разработке с упором на JavaScript:

http://www.dustindiaz.com/

Блог Джонатана Снука (Jonathan Snook)

Блог *Джонатан Снука* (*Jonathan Snook*) посвящен общим вопросам программирования и веб-разработки:

http://snook.ca/

Ресурс Мэтта Снайдера (Matt Snider) о JavaScript

Блог *Мэтта Снайдера* (*Matt Snider*) посвящен изучению JavaScript и многих популярных платформ:

http://mattsnider.com/

I can't

На трех сайтах *Кристиана Хейлманна* (*Christian Heilmann*) можно найти выдержки из блогов, примеры программного кода и объемные статьи, где обсуждаются вопросы использования JavaScript и веб-разработки:

http://icant.co.uk/

http://www.wait-till-i.com/

http://www.onlinetools.org/

DOM scripting

Блог *Джереми Кейта* (*Jeremy Keith*), продолжающий обсуждение вопросов, поднятых в популярной книге по работе с деревом DOM, – фантастический ресурс, посвященный ненавязчивому JavaScript:

http://domscripting.com/blog/

Как дни проходят мимо

Стюарт Лангридж (*Stuart Langridge*) экспериментирует с передовыми приемами реализации DOM в броузере:

http://www.kryogenix.org/code/browser/

A list apart

На сайте *A List Apart* исследуются вопросы проектирования, разработки и значение веб-содержимого с особым упором на соблюдение веб-стандартов и использование наиболее удачных приемов:

http://www.alistapart.com/

Платформы разработки веб-приложений с использованием jQuery

По мере того как разработчики открытых проектов узнают о существовании jQuery, многие из них включают эту библиотеку JavaScript в свои собственные системы. Ниже приводится краткий список некоторых из таких проектов:

Digitalus Site Manager: *http://code.google.com/p/digitalus-site-manager/*

Drupal: *http://drupal.org/*

DutchPIPE: *http://dutchpipe.org/*

Hpricot: *http://code.whytheluckystiff.net/hpricot/*

JobberBase: *http://www.jobberbase.com/*

Laconica: *http://laconi.ca/*

Piwik: *http://piwik.org/*

Pommo: *http://pommo.org/*

simfony: *http://www.symfony-project.org/*

SPIP: *http://www.spip.net/*

Textpattern: *http://www.textpattern.com/*

Trac: *http://trac.edgewall.org/*

WordPress: *http://wordpress.org/*

Z-Blog: *http://www.rainbowsoft.org/zblog*

Более полный список можно найти на странице *Sites Using jQuery* (сайты, использующие jQuery) по адресу:

http://docs.jquery.com/Sites_Using_jQuery

B

Инструменты разработчика

Документация может помочь в решении проблем, возникающих в приложениях на JavaScript, но ничто не заменит хорошего комплекта инструментов разработчика. К счастью, в мире существует множество пакетов программного обеспечения, позволяющих просматривать и отлаживать программный код JavaScript, причем большая их часть распространяется бесплатно.

Инструменты для броузера Firefox

Броузер Mozilla Firefox является предпочтительным броузером львиной доли веб-разработчиков, и потому для него имеется самый обширный набор инструментальных средств, зарекомендовавших себя с положительной стороны.

Firebug

Расширение Firebug для броузера Firefox незаменимо при разработке веб-приложений с использованием jQuery:

http://www.getfirebug.com/

Ниже перечислены некоторые из особенностей Firebug:

- Превосходное средство просмотра дерева DOM, позволяющее подбирать названия и имена селекторов, отбирающих фрагменты документа.
- Инструменты управления стилями CSS, позволяющие обнаруживать причины, которые определяют внешний вид страницы, и вносить изменения в него.
- Интерактивная консоль JavaScript.
- Отладчик JavaScript, дающий возможность просматривать значения переменных и следить за выполнением программного кода.

Панель инструментов веб-разработчика

Этот инструмент не только перекрывает возможности расширения Firebug в области исследования дерева DOM, но и содержит средства, облегчающие выполнение типичных задач, таких как управление cookie, исследование форм и изменение размеров страниц. Данная панель инструментов может также использоваться для быстрого отключения поддержки JavaScript в целях проверки, на сколько деградирует функциональность сайта, если клиент отключит поддержку JavaScript в своем броузере:

http://chrispederick.com/work/web-developer/

Venkman

Venkman – это официальный отладчик JavaScript проекта Mozilla. Он предоставляет среду диагностики, напоминающую систему GDB, используемую для отладки программ, написанных на других языках программирования.

http://www.mozilla.org/projects/venkman/

Средство проверки регулярных выражений

Создание в JavaScript регулярных выражений, отыскивающих совпадения в строках, может оказаться далеко не простым делом. Данное расширение для броузера Firefox позволяет экспериментировать с регулярными выражениями, предоставляя интерфейс для ввода испытуемого текста:

http://sebastianzartner.ath.cx/new/downloads/RExT/

Инструменты для броузера Internet Explorer

Нередко сайты ведут себя в IE иначе, чем в других веб-броузерах, поэтому наличие отладочных инструментов для этой платформы имеет большое значение.

Панель инструментов разработчика для Microsoft Internet Explorer

Панель инструментов разработчика в первую очередь используется для просмотра дерева DOM. Кроме того, с ее помощью обеспечивается возможность визуального поиска элементов и их оперативного изменения с применением новых правил CSS. На панели присутствуют и другие инструменты, такие как линейка, позволяющая определять размеры элементов страницы:

http://www.microsoft.com/downloads/details.aspx?FamilyID=e59c3964-672d-4511-bb3e-2d5e1db91038

Microsoft Visual Web Developer

Среда разработки Microsoft Visual Studio может использоваться для отладки программного кода JavaScript:

http://msdn.microsoft.com/vstudio/express/vwd/

Если вам потребуется запустить интерактивный отладчик в свободной версии (Visual Web Developer Express), следуйте инструкциям на странице:

http://www.berniecode.com/blog/2007/03/08/how-to-debug-javascript-with-visual-web-developer-express/

DebugBar

Инструмент DebugBar предоставляет средство просмотра дерева DOM, а также консоль JavaScript для отладки:

http://www.debugbar.com/

Drip

Утечки памяти в программном коде JavaScript могут вызывать снижение производительности и нестабильное поведение в броузере Internet Explorer. Инструмент Drip поможет определить, где происходят утечки памяти и ликвидировать их:

http://Sourceforge.net/projects/ieleak/

Подробнее узнать о типичных причинах, которые приводят к появлению утечек памяти в Internet Explorer, можно в приложении C «Замыкания в JavaScript».

Инструменты для броузера Safari

Броузер Safari – самый юный в блоке платформ веб-разработки, но и для него имеются инструменты, которые помогут определить, почему программный код ведет себя в этом броузере иначе, чем в других.

Меню Develop

Начиная с версии Safari 3.1, в меню Preferences (настройки) присутствует дополнительный пункт, открывающий доступ к специальному подменю, которое называется Develop (разработка). Это подменю обеспечивает доступ к пунктам Web Inspector (веб-инспектор) и JavaScript Console (консоль JavaScript).

Web Inspector

В версии Safari 3 имеется возможность инспектировать отдельные элементы страницы и собирать информацию о правилах CSS, которые применяются к каждому элементу.

http://trac.webkit.org/wiki/Web%20Inspector

Текущие сборки библиотеки WebKit включают существенные дополнения к этому инструменту, предоставляющие многие замечательные особенности, которыми обладает расширение Firebug, такие как интегрированный отладчик JavaScript с именем Drosera.

http://trac.webkit.org/wiki/Drosera

Инструменты для броузера Opera

Несмотря на ограниченную распространенность броузера Opera для настольных систем, он является важным игроком на рынке встраиваемых систем и мобильных устройств, поэтому его особенности также следует учитывать при разработке веб-приложений.

Dragonfly

Хотя Dragonfly все еще находится на ранней стадии разработки, он представляет собой многообещающий инструмент отладки для броузеров Opera, используемых в компьютерах и мобильных устройствах. Своими возможностями, такими как отладчик JavaScript, средства инспектирования и редактирования CSS и DOM, Dragonfly напоминает Firebug.

Прочие инструменты

Инструменты, упомянутые выше, предназначены для использования в конкретных броузерах, тогда как следующие ниже утилиты имеют более широкую область применения.

Firebug Lite

Область применения расширения Firebug ограничена веб-броузером Firefox, однако некоторые его особенности могут быть воспроизведены за счет подключения к странице сценария Firebug Lite. Этот сценарий имитирует консоль Firebug и обеспечивает работоспособность вызовов метода `console.log()` во всех броузерах:

http://www.getfirebug.com/lite.html

NitobiBug

Как и Firebug Lite, NitobiBug – это инструмент, который можно использовать в разных броузерах и который реализует некоторые из возможностей более надежного и совершенного расширения Firebug. Основная его особенность – средства инспектирования дерева DOM и объектов, хотя в нем имеется также вполне удобная консоль. Консоль и инспек-

тор могут вызываться включением в страницу ссылки на файл Nitobi и обращением к методу `nitobi.Debug.log()`.

http://www.nitobibug.com/

Пакет TextMate jQuery

Это расширение для редактора TextMate, популярного в операционной системе Mac OS X, предоставляет возможность подсветки синтаксиса методов и селекторов jQuery, реализует функцию автодополнения для методов и содержит краткий справочник по прикладному программному интерфейсу, доступный непосредственно из программного кода. Кроме того, данный пакет совместим с текстовым редактором E для Windows:

http://github.com/kswedberg/jquery-tmbundle/

Charles

При разработке приложений, интенсивно использующих технологии AJAX, бывает полезно видеть, какие данные в действительности передаются между сервером и броузером. Отладочный сценарий-посредник Charles отображает весь трафик HTTP между двумя точками, включая обычные веб-запросы, трафик HTTPS, удаленные взаимодействия технологии Flash и ответы AJAX:

http://www.xk72.com/charles/

Fiddler

Fiddler – это еще один сценарий-посредник для отладки взаимодействий по протоколу HTTP, по своим возможностям напоминающий Charles. На его сайте говорится, что Fiddler «включает мощную, основанную на событиях подсистему и может быть расширен с применением любого языка программирования, поддерживаемого платформой .NET»:

http://www.fiddlertool.com/fiddler/

Aptana

Aptana – это свободно распространяемая, межплатформенная интегрированная среда разработки веб-приложений, написанная на языке Java. Наряду со стандартными и улучшенными возможностями редактирования программного кода, она включает полный комплект документации по jQuery API, а также имеет собственный отладчик JavaScript, основанный на Firebug.

http://www.aptana.com/

C

Замыкания в JavaScript

На протяжении всей книги мы видели множество методов jQuery, принимающих **функции** в виде аргументов. Наши примеры снова и снова создавали, вызывали и передавали функции методам. Обычно, чтобы пользоваться этим приемом, достаточно поверхностного знакомства с внутренними механизмами JavaScript, вовлеченными в работу, но иногда без знания особенностей языка побочные эффекты наших действий могут казаться нам странными. В этом приложении мы поближе познакомимся с одной из самых необычных (и все же достаточно распространенных) конструкций, связанных с использованием функций, которая называется **замыканием**.

В процессе обсуждения мы рассмотрим множество небольших примеров программного кода, которые будут выводить серию сообщений. Вместо того чтобы задействовать механизм вывода сообщений, характерный для определенного броузера (такой, как метод `console.log()` в броузере Firefox), или создавать серию диалогов `alert()`, мы будем использовать небольшой метод расширения:

```
jQuery.fn.print = function(message) {
  return this.each(function() {
    $('<div class="result" />')
      .text(String(message))
      .appendTo($(this).find('.results'));
  });
};
```

С помощью этого метода мы сможем, например, добавить текст «hello» в элемент `<div id="example">`, записав вызов `$('#example').print('hello')`.

Вложенные функции

Язык JavaScript входит в число языков программирования, которые поддерживают возможность объявления **вложенных функций**. Мно-

гие традиционные языки программирования, такие как C, собирают все функции в одной, самой верхней области видимости. В отличие от них языки с поддержкой вложенных функций позволяют определять небольшие вспомогательные функции там, где они необходимы, и избегать **загрязнения пространства имен**.

Вложенная функция – это просто функция, которая определена внутри другой функции. Например:

```
function outerFn() {
  function innerFn() {
  }
}
```

Здесь `innerFn()` – это вложенная функция, объявленная внутри области видимости функции `outerFn()`. Это означает, что функцию `innerFn()` можно вызвать только внутри `outerFn()`, но не за ее пределами. Следующий пример вызовет ошибку JavaScript:

```
function outerFn() {
  $('#example-2').print('Outer function');
  function innerFn() {
    $('#example-1').print('Inner Function');
  }
}
$('#example-1').print('innerFn():');
innerFn();
```

Однако мы благополучно можем вызывать `innerFn()` внутри функции `outerFn()`:

```
function outerFn() {
  $('#example-2').print('Outer function');
  function innerFn() {
    $('#example-2').print('Inner function');
  }
  innerFn();
}
$('#example-2').print('outerFn():');
outerFn();
```

В результате будет выведено:

```
outerFn():
Outer function
Inner function
```

Этот прием особенно удобно использовать для объявления небольших узкоспециализированных функций. Например, обертывание рекурсивных алгоритмов в нерекурсивный API часто бывает проще выразить при помощи вложенной вспомогательной функции.

Великий побег

Проблема становится интереснее, когда в игру вступают **ссылки на функции**. В некоторых языках программирования, таких как Pascal, допускается использование вложенных функций, но только с целью **сокрытия программного кода**; эти функции навсегда погребены внутри родительских функций. В языке JavaScript, напротив, допускается передавать функции, как данные любого другого типа. Это означает, что вложенные функции имеют возможность «сбежать» от своих захватчиков.

Маршрут побега может иметь разные направления. Например, предположим, что функция присвоена глобальной переменной:

```
var globalVar;
function outerFn() {
  $('#example-3').print('Outer function');
  function innerFn() {
    $('#example-3').print('Inner function');
  }
  globalVar = innerFn;
}
$('#example-3').print('outerFn():');
outerFn();
$('#example-3').print('globalVar():');
globalVar();
```

Вызов функции `outerFn()` после ее определения изменит глобальную переменную `globalVar`. Теперь она ссылается на функцию `innerFn()`. Это означает, что следующий ниже вызов `globalVar()` будет действовать как вызов `innerFn()` и выполнит инструкцию `print`:

```
outerFn():
Outer function
globalVar():
Inner function
```

Обратите внимание: вызов функции `innerFn()` за пределами `outerFn()` по-прежнему будет приводить к появлению ошибки! Функции удалось сбежать путем сохранения ссылки в глобальной переменной, но имя функции все еще заперто внутри области видимости `outerFn()`.

Ссылка на вложенную функцию может также покинуть родительскую функцию через **возвращаемое значение**:

```
function outerFn() {
  $('#example-4').print('Outer function');
  function innerFn() {
    $('#example-4').print('Inner function');
  }
  return innerFn;
}
```

```
$('#example-4').print('var fnRef = outerFn():');
var fnRef = outerFn();
$('#example-4').print(fnRef():');
fnRef();
```

Здесь не производится изменение глобальной переменной внутри `outerFn()`. Вместо этого функция `outerFn()` возвращает ссылку на `innerFn()`. Результатом вызова `outerFn()` является ссылка, которая сохраняется и затем вызывается, что снова приводит к появлению сообщения:

```
var fnRef = outerFn():
Outer function
fnRef():
Inner function
```

Тот факт, что вложенные функции могут вызываться посредством ссылок даже после того, как функция вышла из области видимости, означает, что интерпретатор JavaScript должен сохранять такие функции доступными, пока они могут быть вызваны. В ходе выполнения JavaScript следит за каждой переменной, которая хранит ссылку на функцию, и как только последняя из них выходит из области видимости, приходит **сборщик мусора** и освобождает участок памяти, занятый этой функцией.

Область видимости переменных

Вложенные функции могут иметь свои собственные переменные, область видимости которых ограничена самой функцией:

```
function outerFn() {
  function innerFn() {
    var innerVar = 0;
    innerVar++;
    $('#example-5').print('innerVar = ' + innerVar);
  }
  return innerFn;
}
var fnRef = outerFn();
fnRef();
fnRef();
var fnRef2 = outerFn();
fnRef2();
fnRef2();
```

При каждом вызове функции, через ссылку или как-то иначе, будет создаваться новая переменная `innerVar`, увеличиваться на 1 и отображаться:

```
innerVar = 1
innerVar = 1
innerVar = 1
innerVar = 1
```

Вложенные функции могут ссылаться на глобальные переменные точно так же, как и любые другие функции:

```
var globalVar = 0;
function outerFn() {
  function innerFn() {
    globalVar++;
    $('#example-6').print('globalVar = ' + globalVar);
  }
  return innerFn;
}
var fnRef = outerFn();
fnRef();
fnRef();
var fnRef2 = outerFn();
fnRef2();
fnRef2();
```

Теперь при каждом вызове наша функция будет последовательно увеличивать значение переменной:

```
globalVar = 1
globalVar = 2
globalVar = 3
globalVar = 4
```

Но что если переменная будет локальной по отношению к родительской функции? Так как вложенная функция наследует область видимости родительской функции, она сможет ссылаться и на эту переменную тоже:

```
function outerFn() {
  var outerVar = 0;
  function innerFn() {
    outerVar++;
    $('#example-7').print('outerVar = ' + outerVar);
  }
  return innerFn;
}
var fnRef = outerFn();
fnRef();
fnRef();
var fnRef2 = outerFn();
fnRef2();
fnRef2();
```

Теперь наша функция проявляет гораздо более интересное поведение:

```
outerVar = 1
outerVar = 2
outerVar = 1
outerVar = 2
```

Мы получили смесь двух предыдущих эффектов. Вызов `innerFn()` через каждую ссылку приводит к независимому увеличению `outerVar`. Обратите внимание: второй вызов `outerFn()` не выполняет повторную запись начального значения в переменную `outerVar`, а создает новый экземпляр `outerVar` и связывает его с областью видимости второго вызова функции. В результате этого после описанных выше вызовов новый вызов `fnRef()` выведет значение 3, и последующий вызов `fnRef2()` тоже выведет значение 3. Два счетчика никак не зависят друг от друга.

Когда ссылка на вложенную функцию попадает за пределы области видимости, в которой эта функция была определена, создается **замыкание** этой функции. Переменные, которые не являются ни параметрами, ни локальными переменными вложенной функции, мы называем **свободными переменными**, окружение вызова внешней функции **замыкает** их. По существу обращение функции к переменной, являющейся локальной для внешней функции, дает этой переменной отсрочку в исполнении приговора. По завершении функции память не освобождается, так как она еще необходима замыканию.

Взаимодействия между замыканиями

При наличии более одной вложенной функции замыкания могут порождать эффекты, которые трудно предвидеть. Предположим, что помимо функции, увеличивающей значение переменной на 1, существует еще одна вложенная функция, которая увеличивает эту же переменную на 2:

```
function outerFn() {
  var outerVar = 0;
  function innerFn1() {
    outerVar++;
    $('#example-8').print('(1) outerVar = ' + outerVar);
  }
  function innerFn2() {
    outerVar += 2;
    $('#example-8').print('(2) outerVar = ' + outerVar);
  }
  return {'fn1': innerFn1, 'fn2': innerFn2};
}
var fnRef = outerFn();
fnRef.fn1();
fnRef.fn2();
fnRef.fn1();
var fnRef2 = outerFn();
fnRef2.fn1();
fnRef2.fn2();
fnRef2.fn1();
```

Мы возвращаем ссылки на обе функции, используя **отображение** (это иллюстрирует еще один путь, которым ссылки на вложенные функции

могут покинуть пределы родительской функции). Обе функции вызываются посредством ссылок:

```
(1) outerVar = 1
(2) outerVar = 3
(1) outerVar = 4
(1) outerVar = 1
(2) outerVar = 3
(1) outerVar = 4
```

Обе вложенные функции ссылаются на одну и ту же локальную переменную, поэтому они делят между собой одну и ту же **среду замыкания**. Когда `innerFn1()` увеличивает значение переменной `outerVar` на 1, она тем самым устанавливает новое начальное значение `outerVar` для функции `innerFn2()`, и наоборот. Здесь мы снова видим, что каждый вызов `outerFn()` создает новые экземпляры замыканий с соответствующими им новыми замкнутыми окружениями. Фанаты **объектно-ориентированного программирования** могут заметить, что вызов `outerFn()`, по сути, создает новый **объект** с тремя переменными, которые действуют как **переменные экземпляра**, и замыканиями, которые действуют как **методы экземпляра**. Кроме того, переменные являются **частными**, поскольку к ним невозможно получить доступ из-за пределов области видимости замыкания, что обеспечивает сокрытие данных, характерное для объектно-ориентированного программирования.

Замыкания в библиотеке jQuery

Методы библиотеки jQuery, с которыми мы встречались, часто принимают в виде аргумента по меньшей мере одну функцию. Для удобства мы часто используем **анонимные функции**, что позволяет определять реализацию функций только там, где необходимо. Это означает, что такие функции редко бывают в пространстве имен верхнего уровня; они обычно объявляются как вложенные функции, следовательно, легко могут приводить к созданию замыканий.

Аргументы метода $(document).ready()

Почти весь программный код, который мы создаем при использовании библиотеки jQuery, помещается внутрь функции, передаваемой как аргумент методу `$(document).ready()`. Делается это с целью гарантировать запуск программного кода только после того, как будет загружено дерево DOM, что обычно является обязательным условием для нормальной работы программного кода jQuery. Когда функция создается и передается методу `.ready()`, ссылка на функцию сохраняется в составе глобального объекта jQuery. Позднее, когда дерево DOM будет готово к работе, эта ссылка будет использоваться для вызова функции.

Обычно вызов `$(document).ready()` располагается на самом верхнем уровне программного кода, поэтому данная функция не является в действи-

тельности частью замыкания. Однако так как наш программный код обычно находится внутри этой функции, все *остальное* является вложенной функцией:

```
$(document).ready(function() {
  var readyVar = 0;
  function innerFn() {
    readyVar++;
    $('#example-9').print('readyVar = ' + readyVar);
  }
  innerFn();
  innerFn();
});
```

Этот фрагмент очень похож на предыдущие примеры, за исключением того, что в данном случае роль внешней функции играет функция обратного вызова, которая передается методу `$(document).ready()`. Так как функция `innerFn()` определена внутри функции обратного вызова и ссылается на переменную `readyVar`, находящуюся в области видимости функции обратного вызова, `innerFn()` и ее окружение создают замыкание. Это можно наблюдать по тому, как сохраняется значение переменной `readyVar` между вызовами функции:

```
readyVar = 1
readyVar = 2
```

Тот факт, что большая часть программного кода, использующего jQuery, находится внутри тела функции, играет важную роль, поскольку может защитить от некоторых конфликтов имен. Например, именно данная особенность позволяет использовать `jQuery.noConflict()`, чтобы освободить псевдоним `$` для нужд других библиотек, и дает при этом возможность определить этот псевдоним локально для использования в пределах `$(document).ready()`.

Обработчики событий

Конструкция `$(document).ready()` обычно заключает в себя весь наш программный код, включая и инструкции присваивания **обработчиков событий**. Обработчики событий являются функциями, поэтому они превращаются во вложенные функции. А так как эти вложенные функции сохраняются и вызываются позднее, они тоже могут приводить к образованию замыканий, что можно проиллюстрировать на примере простого обработчика события `click`:

```
$(document).ready(function() {
  var counter = 0;
  $('#example-10 a.add').click(function() {
    counter++;
    $('#example-10').print('counter = ' + counter);
    return false;
```

```
  });
});
```

Переменная `counter` объявлена внутри обработчика `.ready()`, поэтому она доступна только программному коду внутри этого блока и недоступна за его пределами. Однако, она доступна программному коду обработчика события `click`, который наращивает и отображает значение переменной. Поскольку образуется замыкание, при каждом щелчке мыши на ссылке происходит обращение к одному и тому же экземпляру переменной `counter`. Это означает, что при выводе сообщений будет отображаться постоянно увеличивающееся значение:

```
counter = 1
counter = 2
counter = 3
```

Обработчики событий могут делить между собой замкнутое окружение, как и другие функции:

```
$(document).ready(function() {
  var counter = 0;
  $('#example-11 a.add').click(function() {
    counter++;
    $('#example-11').print('counter = ' + counter);
    return false;
  });
  $('#example-11 a.subtract').click(function() {
    counter--;
    $('#example-11').print('counter = ' + counter);
    return false;
  });
});
```

Обе функции ссылаются на одну и ту же переменную `counter`, поэтому операции увеличения и уменьшения значения, реализованные для двух ссылок, оказывают влияние не на независимые, а на одно и то же значение:

```
counter = 1
counter = 2
counter = 1
counter = 0
```

В примерах выше использовались **анонимные функции**, как это обычно бывает в нашем программном коде, использующем библиотеку jQuery. Однако для образования замыканий это не имеет никакого значения – замыкания могут образовываться не только анонимными, но и именованными функциями. Например, мы можем написать анонимную функцию для вывода индекса элемента внутри объекта jQuery:

```
$(document).ready(function() {
  $('#example-12 a').each(function(index) {
```

```
    $(this).click(function() {
      $('#example-12').print('index = ' + index);
      return false;
    });
  });
});
```

Самая внутренняя функция определена внутри функции обратного вызова для метода `.each()`, поэтому данный пример фактически создает столько функций, сколько имеется ссылок на странице. Каждая из этих функций подключается как обработчик события `click` к одной из ссылок. Каждая функция получает в аргументе `index` порядковый номер своего замкнутого окружения, который является параметром функции обратного вызова метода `.each()`. Эта реализация действует точно так же, как если бы мы оформили обработчик события `click` в виде именованной функции:

```
$(document).ready(function() {
  $('#example-13 a').each(function(index) {
    function clickHandler() {
      $('#example-13').print('index = ' + index);
      return false;
    }
    $(this).click(clickHandler);
  });
});
```

Просто версия с анонимной функцией немного короче. Однако местоположение именованной функции имеет большое значение:

```
$(document).ready(function() {
  function clickHandler() {
    $('#example-14').print('index = ' + index);
    return false;
  }
  $('#example-14 a').each(function(index) {
    $(this).click(clickHandler);
  });
});
```

При щелчке мышью на ссылке эта версия будет вызывать ошибку JavaScript, потому что переменная `index` отсутствует в замкнутом окружении функции `clickHandler()`. Она остается свободной переменной и потому не определена в данном контексте.

Угроза утечки памяти

Управление памятью в JavaScript реализовано с использованием приема, известного как **сборка мусора**. Это отличает JavaScript от низкоуровневых языков, таких как C, где от программиста требуется явно **ре-**

зервировать блоки памяти и освобождать их, когда они становятся ненужными. В других языках программирования, таких как Objective-C, в помощь программисту реализована система **подсчета ссылок**, которая дает пользователю возможность делать заметки о количестве ссылок на определенный участок памяти, чтобы его можно было освободить, когда он перестанет использоваться. Напротив, JavaScript – это язык программирования высокого уровня, и сам неявно заботится об освобождении памяти.

Всякий раз, когда в программном коде появляется новый элемент, размещаемый в памяти, такой как объект или функция, за ним резервируется участок памяти. Так как объект может передаваться функциями и присваиваться переменным, ссылки на него могут одновременно присутствовать в нескольких частях программы. JavaScript следит за этими **указателями**, и когда уничтожается последний из них, память, занимаемая объектом, освобождается. Рассмотрим цепочку указателей, показанную на рис. С.1.

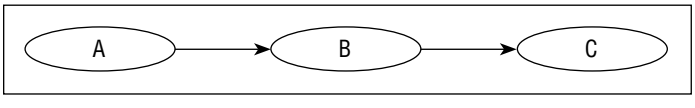


Рис. С.1. *Цепочка указателей*

Здесь объект *A* имеет свойство, указывающее на объект *B*, а объект *B* имеет свойство, указывающее на объект *C*. Даже если объект *A* окажется единственной переменной в текущей области видимости, в памяти все равно будут храниться все три объекта, потому что существуют указатели, ссылающиеся на них. Однако когда объект *A* покинет область видимости (например, после выхода из функции, внутри которой он был объявлен), он будет уничтожен сборщиком мусора. После этого не останется ни одной ссылки на объект *B*, поэтому и он будет уничтожен; в конце концов будет также уничтожен объект *C*.

Со ссылками могут возникать более сложные для обработки ситуации, как показано на рис. С.2.

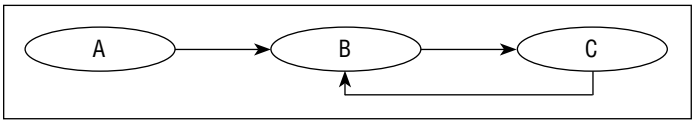


Рис. С.2. *Цепочка указателей с циклическими ссылками*

Теперь мы добавили в объект *C* свойство, ссылающееся на объект *B*. В данном случае, когда будет уничтожен объект *A*, в объекте *C* продолжит существовать ссылка на объект *B*. Такая **циклическая ссылка** требует от интерпретатора JavaScript особой обработки – он должен заме-

тить, что вся петля изолирована от переменных в данной области видимости.

Случайные циклические ссылки

Замыкания могут приводить к непреднамеренному созданию циклических ссылок. Функции являются объектами, которые должны сохраняться в памяти, поэтому любые переменные, имеющиеся в их замкнутом окружении, также должны сохраняться в памяти:

```
function outerFn() {
  var outerVar = {};
  function innerFn() {
    alert(outerVar);
  }
  outerVar.fn = innerFn;
  return innerFn;
};
```

Здесь создается объект `outerVar`, который используется внутри вложенной функции `innerFn()`. Затем в объекте `outerVar` создается свойство, указывающее на `innerFn()`, и возвращается ссылка на `innerFn()`. В результате функция `innerFn()` образует замыкание, ссылаясь на объект `outerVar`, который в свою очередь ссылается на функцию `innerFn()`. Но циклическая ссылка может быть еще более коварной:

```
function outerFn() {
  var outerVar = {};
  function innerFn() {
    alert('hello');
  }
  outerVar.fn = innerFn;
  return innerFn;
};
```

Здесь мы изменили функцию `innerFn()` так, что она больше не ссылается на объект `outerVar`. Однако это не устранило циклическую ссылку! Хотя функция `innerFn()` не ссылается на объект `outerVar`, он по-прежнему существует в **замкнутом окружении** `innerFn()`. Вследствие образования замыкания функция `innerFn()` *неявно* ссылается на все переменные в области видимости `outerFn()`. Таким образом, замыкания легко могут образовывать непреднамеренные циклические ссылки.

Проблема утечки памяти в Internet Explorer

Обычно все это не представляет большой проблемы, так как JavaScript в состоянии обнаружить такие циклические ссылки и освободить память, когда необходимо. Однако в Internet Explorer существует проблема, связанная с обработкой одного определенного класса циклических ссылок. Когда цикл образуется между элементами DOM и обычными объектами JavaScript, IE не может освободить ни один из них, потому

что они обслуживаются разными механизмами управления памятью. Эти циклические ссылки продолжают существовать, пока не будет закрыто окно броузера, что с течением времени может привести к потреблению больших объемов памяти. Часто причиной образования таких циклических ссылок является простой обработчик события:

```
$(document).ready(function() {
  var div = document.getElementById('foo');
  div.onclick = function() {
    alert('hello');
  };
});
```

Когда выполняется присваивание обработчика события `click`, образуется замыкание с элементом `div` в замкнутом окружении. Но теперь элемент `div` содержит обратную ссылку на замыкание – собственное свойство `onclick`. Такая циклическая ссылка не может быть ликвидирована броузером Internet Explorer даже при переходе на другую страницу.

Добрая весть

Теперь запишем тот же программный код, но уже с использованием обычных конструкций jQuery:

```
$(document).ready(function() {
  var $div = $('#foo');
  $div.click(function() {
    alert('hello');
  });
});
```

Хотя здесь по-прежнему образуется замыкание, создающее ту же циклическую ссылку, что и прежде, этот программный код не порождает утечку памяти в IE. К счастью, библиотека jQuery знает о потенциальной угрозе утечки памяти и вручную освобождает все обработчики событий, которые были присвоены. Пока мы неукоснительно придерживаемся использования методов jQuery подключения обработчиков событий, мы можем не бояться утечек памяти в подобных случаях.

Это не означает, что мы полностью избавились от проблем; мы по-прежнему должны внимательно относиться к выполнению других операций с привлечением элементов DOM. Подключение объектов JavaScript к элементам DOM все еще может вызывать утечки памяти в Internet Explorer; библиотека jQuery просто помогает сузить круг влияния этой ситуации.

Чтобы помочь избежать подобных утечек памяти, библиотека jQuery дает нам в руки еще один инструмент. В главе 7 мы видели, что метод `.data()` позволяет добавлять информацию к элементам DOM практически так же, как при использовании расширенных свойств. Поскольку эти данные не сохраняются непосредственно, как при использовании

расширенных свойств (jQuery сохраняет данные во внутреннем отображении, используя значение атрибута `id` элемента), образования циклических ссылок не происходит, и мы обходим стороной проблему утечки памяти. Всякий раз, когда механизм расширенных свойств кажется удобным механизмом хранения данных, мы должны подумать о возможности использования метода `.data()` как более безопасной альтернативы.

В заключение

Замыкания JavaScript представляют собой мощную особенность языка. Они часто весьма полезны для сокрытия переменных от иного программного кода, в результате которого можно избежать конфликтов с именами переменных, используемых в другом месте. Из-за того, что при использовании jQuery функции обычно передаются методам в виде аргументов, они очень часто могут непреднамеренно образовывать замыкания. Понимание замыканий позволяет писать более эффективный и более краткий программный код, а соблюдение мер предосторожности и использование защитных мер, встроенных в библиотеку jQuery, помогает избегать ловушек, связанных с памятью.

D

Краткий справочник

Данное приложение играет роль краткого справочника по прикладному программному интерфейсу jQuery, включая селекторы и методы. Более подробное обсуждение этой темы можно найти в сопутствующей книге «jQuery Reference Guide», а также на сайте документации для jQuery *http://docs.jquery.com*.

Селекторные выражения

Фабричная функция `$()` в библиотеке jQuery используется для поиска элементов страницы с целью последующего выполнения операций над ними. Эта функция принимает строку с CSS-подобным синтаксисом, называемую **селекторным выражением**. Селекторные выражения подробно обсуждаются в главе 2.

Селектор	Соответствуют
`*`	Все элементы.
`#id`	Элементы с указанным значением атрибута `ID`.
`element`	Все элементы заданного типа.
`.class`	Все элементы с заданным классом.
`a, b`	Элементы, соответствующие селектору `a` или `b`.
`a b`	Элементы `b`, являющиеся потомками по отношению к `a`.
`a > b`	Элементы `b`, являющиеся дочерними по отношению к `a`.
`a + b`	Элементы `b`, которые следуют непосредственно за `a`.

Селектор	Соответствуют
`a ~ b`	Элементы `b`, являющиеся братскими по отношению к `a`.
`:first`	Первый элемент в результирующем наборе.
`:last`	Последний элемент в результирующем наборе.
`:not(a)`	Все элементы в результирующем наборе, которые не соответствуют `a`.
`:even`	Элементы в результирующем наборе с четными порядковыми номерами (отсчет начинается с 0).
`:odd`	Элементы в результирующем наборе с нечетными порядковыми номерами (отсчет начинается с 0).
`:eq(index)`	Элемент в результирующем наборе с указанным порядковым номером (отсчет начинается с 0).
`:gt(index)`	Все элементы в результирующем наборе с порядковыми номерами больше указанного (отсчет начинается с 0).
`:lt(index)`	Все элементы в результирующем наборе с порядковыми номерами меньше указанного (отсчет начинается с 0).
`:header`	Элементы-заголовки (такие как `<h1>`, `<h2>`).
`:animated`	Элементы, находящиеся в состоянии воспроизведения анимационного эффекта.
`:contains(text)`	Элементы, содержащие указанный текст.
`:empty`	Элементы, не имеющие дочерних узлов.
`:has(a)`	Элементы, содержащие элементы-потомки, соответствующие `a`.
`:parent`	Элементы, имеющие дочерние узлы.
`:hidden`	Элементы, которые скрыты либо с помощью CSS, либо потому что они являются элементами `<input type="hidden" />`.
`:visible`	Все элементы, не соответствующие селектору `:hidden`.
`[attr]`	Элементы, имеющие атрибут `attr`.

Селектор	Соответствуют
`[attr=value]`	Элементы, имеющие атрибут `attr` со значением `value`.
`[attr!=value]`	Элементы, в которых значение атрибута `attr` не равно `value`.
`[attr^=value]`	Элементы, в которых значение атрибута `attr` начинается с `value`.
`[attr$=value]`	Элементы, в которых значение атрибута `attr` заканчивается на `value`.
`[attr*=value]`	Элементы, в которых значение атрибута `attr` содержит `value`.
`:nth-child(index)`	Элемент, являющийся `index`-м дочерним элементом в родительском элементе (отсчет начинается с 1).
`:nth-child(even)`	Элемент, являющийся дочерним элементом с четным порядковым номером в родительском элементе (отсчет начинается с 1).
`:nth-child(odd)`	Элемент, являющийся дочерним элементом с нечетным порядковым номером в родительском элементе (отсчет начинается с 1).
`:nth-child(formula)`	Элемент, являющийся `n`-м дочерним элементом в родительском элементе (отсчет начинается с 1). Формулы имеют вид `an+b` для целых чисел `a` и `b`.
`:first-child`	Первые дочерние элементы для своих родителей.
`:last-child`	Последние дочерние элементы для своих родителей.
`:only-child`	Элементы, являющиеся единственными дочерними для своих родителей.
`:input`	Все элементы `<input>`, `<select>`, `<textarea>` и `<button>`.
`:text`	Элементы `<input>` с атрибутом `type="text"`.
`:password`	Элементы `<input>` с атрибутом `type="password"`.
`:radio`	Элементы `<input>` с атрибутом `type="radio"`.
`:checkbox`	Элементы `<input>` с атрибутом `type="checkbox"`.
`:submit`	Элементы `<input>` с атрибутом `type="submit"`.
`:image`	Элементы `<input>` с атрибутом `type="image"`.

Селектор	Соответствуют
:reset	Элементы <input> с атрибутом type="reset".
:button	Элементы <input> с атрибутом type="button", а также элементы <button>.
:file	Элементы <input> с атрибутом type="file".
:enabled	Активные элементы форм.
:disabled	Неактивные элементы форм.
:checked	Отмеченные флажки и радиокнопки.
:selected	Выбранные элементы <option>.

Методы навигации по дереву DOM

После создания объекта jQuery с помощью $() мы можем изменять набор элементов вызовом одного из следующих **методов навигации по дереву DOM**. Методы навигации по дереву DOM подробно рассматриваются в главе 2.

Метод навигации	Возвращает объект jQuery, содержащий...
.filter(selector)	Элементы из набора, соответствующие заданному селектору.
.filter(callback)	Элементы из набора, для которых функция callback возвращает значение true.
.eq(index)	Элемент из набора с указанным порядковым номером (отсчет начинается с 0).
.slice(start, [end])	Элементы из набора с порядковыми номерами из указанного диапазона (отсчет начинается с 0).
.not(selector)	Элементы из набора, не соответствующие заданному селектору.
.add(selector)	Элементы из набора плюс все элементы, соответствующие заданному селектору.
.find(selector)	Потомки элементов из набора, соответствующие заданному селектору.
.contents()	Дочерние узлы (включая текстовые узлы).
.children([selector])	Дочерние узлы, которые дополнительно могут быть отфильтрованы указанным селектором.

Метод навигации	Возвращает объект jQuery, содержащий...
`.next([selector])`	Братские узлы, следующие непосредственно за каждым элементом из набора, которые дополнительно могут быть отфильтрованы указанным селектором.
`.nextAll([selector])`	Все братские узлы, следующие за каждым элементом из набора, которые дополнительно могут быть отфильтрованы указанным селектором.
`.prev([selector])`	Братские узлы, непосредственно предшествующие каждому элементу из набора, которые дополнительно могут быть отфильтрованы указанным селектором.
`.prevAll([selector])`	Все братские узлы, предшествующие каждому элементу из набора, которые дополнительно могут быть отфильтрованы указанным селектором.
`.siblings([selector])`	Все братские узлы, которые дополнительно могут быть отфильтрованы указанным селектором.
`.parent([selector])`	Родители каждого элемента из набора, которые дополнительно могут быть отфильтрованы указанным селектором.
`.parents([selector])`	Все предки, которые дополнительно могут быть отфильтрованы указанным селектором.
`.closest selector`	Первый элемент, соответствующий указанному селектору, начиная от элемента в наборе и двигаясь вверх по дереву DOM.
`.offsetParent()`	Позиционированный родитель (то есть имеющий свойство `position` со значением `relative` или `absolute`) **первого** элемента в наборе.
`.andSelf()`	Элементы в наборе плюс предыдущий набор выбранных элементов во внутреннем стеке jQuery.
`.end()`	Предыдущий набор выбранных элементов во внутреннем стеке jQuery.
`.map(callback)`	Результат функции `callback`, вызываемой для каждого элемента в наборе.

Методы событий

Чтобы иметь возможность реагировать на действия пользователя, нам приходится регистрировать свои обработчики событий с помощью **методов событий**. Обратите внимание, что многие события DOM относятся только к элементам определенных типов; эти тонкости здесь не рассматриваются. Методы событий подробно обсуждаются в главе 3.

Метод события	Описание
`.ready(handler)`	Подключает обработчик события `handler`, который будет вызван, когда будут полностью загружены дерево DOM и таблицы стилей CSS.
`.bind(type, [data], handler)`	Подключает обработчик `handler`, который будет вызван, когда элементу будет послано событие указанного типа.
`.one(type, [data], handler)`	Подключает обработчик `handler`, который будет вызван, когда элементу будет послано событие указанного типа. При первом вызове обработчик автоматически отключается.
`.unbind([type], [handler])`	Отключает обработчики событий элемента (для указанного типа события, конкретный обработчик или все обработчики).
`.live(type, handler)`	Подключает обработчик `handler`, который будет вызван, когда элементу будет послано событие указанного типа с использованием приема делегирования.
`.die(type, [handler])`	Отключает обработчики событий элемента, ранее подключенные методом `.live()`.
`.blur(handler)`	Подключает обработчик `handler`, который будет вызван, когда элемент потеряет фокус ввода.
`.change(handler)`	Подключает обработчик `handler`, который будет вызван, когда изменится значение элемента.
`.click(handler)`	Подключает обработчик `handler`, который будет вызван в ответ на щелчок мышью на элементе.

Метод события	Описание
`.dblclick(handler)`	Подключает обработчик `handler`, который будет вызван в ответ на двойной щелчок мышью на элементе.
`.error(handler)`	Подключает обработчик `handler`, который будет вызван, когда элемент примет событие `error` (зависит от типа броузера).
`.focus(handler)`	Подключает обработчик `handler`, который будет вызван, когда элемент получит фокус ввода.
`.keydown(handler)`	Подключает обработчик `handler`, который будет вызван при нажимании клавиши, когда элемент имеет фокус ввода.
`.keypress(handler)`	Подключает обработчик `handler`, который будет вызван при нажатии клавиши, когда элемент имеет фокус ввода.
`.keyup(handler)`	Подключает обработчик `handler`, который будет вызван при отпускании клавиши, когда элемент имеет фокус ввода.
`.load(handler)`	Подключает обработчик `handler`, который будет вызван по окончании загрузки элемента.
`.mousedown(handler)`	Подключает обработчик `handler`, который будет вызван при нажимании кнопки мыши в пределах элемента.
`.mouseenter(handler)`	Подключает обработчик `handler`, который будет вызван при наведении указателя мыши на элемент. Событие не всплывает.
`.mouseleave(handler)`	Подключает обработчик `handler`, который будет вызван при выводе указателя мыши за пределы элемента. Событие не всплывает.
`.mousemove(handler)`	Подключает обработчик `handler`, который будет вызван при перемещении указателя мыши над элементом.
`.mouseout(handler)`	Подключает обработчик `handler`, который будет вызван при выводе указателя мыши за пределы элемента.

Метод события	Описание
`.mouseover(handler)`	Подключает обработчик `handler`, который будет вызван при наведении указателя мыши на элемент.
`.mouseup(handler)`	Подключает обработчик `handler`, который будет вызван при отпускании кнопки мыши в пределах элемента.
`.resize(handler)`	Подключает обработчик `handler`, который будет вызван при изменении размеров элемента.
`.scroll(handler)`	Подключает обработчик `handler`, который будет вызван при изменении позиции полосы прокрутки элемента.
`.select(handler)`	Подключает обработчик `handler`, который будет вызван при выделении текста в элементе.
`.submit(handler)`	Подключает обработчик `handler`, который будет вызван при отправке элемента формы.
`.unload(handler)`	Подключает обработчик `handler`, который будет вызван при выгрузке элемента из памяти.
`.hover(enter, leave)`	Подключает обработчик `enter`, который будет вызван при наведении указателя мыши на элемент, и обработчик `leave`, который будет вызван при выводе указателя мыши за пределы элемента.
`.toggle(handler1, handler2, ...)`	Подключает обработчик `handler1`, который будет вызван в ответ на первый щелчок мышью, обработчик `handler2`, который будет вызван в ответ на второй щелчок, и так далее.
`.trigger(type, [data])`	Запускает обработчики события указанного типа для данного элемента и выполняет действие по умолчанию, предусмотренное для события.
`.triggerHandler(type, [data])`	Запускает обработчики события указанного типа для данного элемента без выполнения действия по умолчанию, предусмотренного для события.
`.blur()`	Возбуждает событие `blur`.

Метод события	Описание
`.change()`	Возбуждает событие `change`.
`.click()`	Возбуждает событие `click`.
`.dblclick()`	Возбуждает событие `dblclick`.
`.error()`	Возбуждает событие `error`.
`.focus()`	Возбуждает событие `focus`.
`.keydown()`	Возбуждает событие `keydown`.
`.keypress()`	Возбуждает событие `keypress`.
`.keyup()`	Возбуждает событие `keyup`.
`.select()`	Возбуждает событие `select`.
`.submit()`	Возбуждает событие `submit`.

Методы эффектов

Методы эффектов могут использоваться для применения анимационных эффектов к элементам DOM. Подробнее методы эффектов рассматриваются в главе 4.

Метод эффекта	Описание
`.show()`	Отображает выбранные элементы.
`.hide()`	Скрывает выбранные элементы.
`.show(speed, [callback])`	Отображает выбранные элементы, постепенно изменяя их высоту, ширину и непрозрачность.
`.hide(speed, [callback])`	Скрывает выбранные элементы, постепенно изменяя их высоту, ширину и непрозрачность.
`.toggle([speed], [callback])`	Отображает или скрывает выбранные элементы.
`.slideDown([speed], [callback])`	Отображает выбранные элементы с эффектом развертывания вниз.
`.slideUp([speed], [callback])`	Скрывает выбранные элементы с эффектом свертывания вверх.

Метод эффекта	Описание
`.slideToggle([speed], [callback])`	Отображает или скрывает выбранные элементы с эффектом развертывания вниз или свертывания вверх соответственно.
`.fadeIn([speed], [callback])`	Отображает выбранные элементы с эффектом постепенного проявления.
`.fadeOut([speed], [callback])`	Скрывает выбранные элементы с эффектом постепенного растворения.
`.fadeTo(speed, opacity, [callback])`	Устанавливает непрозрачность элемента равной указанному значению.
`.animate(attributes, [speed], [easing], [callback])`	Применяет анимационные эффекты к указанным атрибутам CSS.
`.animate(attributes, options)`	Низкоуровневый интерфейс к методу `.animate()`, позволяющий управлять очередью эффектов.
`.stop([clearQueue], [jumpToEnd])`	Останавливает воспроизведение текущего анимационного эффекта и запускает следующий эффект в очереди, если таковой существует.
`.queue()`	Извлекает очередь эффектов для первого элемента в наборе.
`.queue(callback)`	Добавляет функцию обратного вызова `callback` в конец очереди.
`.queue(newQueue)`	Замещает старую очередь новой.
`.dequeue()`	Запускает следующий эффект в очереди.

Методы манипулирования деревом DOM

Подробнее методы манипулирования деревом DOM обсуждаются в главе 5.

Метод	Описание
`.attr(key)`	Возвращает значение атрибута с именем `key`.
`.attr(key, value)`	Значение атрибута `key` устанавливается равным `value`.

Метод	Описание
`.attr(key, fn)`	Значение атрибута `key` устанавливается равным возвращаемому значению функции `fn` (вызывается отдельно для каждого элемента).
`.attr(map)`	Значения атрибутов устанавливаются в соответствии с указанным набором пар ключ-значение.
`.removeAttr(key)`	Удаляет атрибут с именем `key`.
`.addClass(class)`	Добавляет указанный класс к каждому элементу в наборе.
`.removeClass(class)`	Удаляет указанный класс из каждого элемента в наборе.
`.toggleClass(class)`	Удаляет указанный класс из элементов в наборе, в которых он присутствует, и добавляет в элементы из набора, в которых он отсутствует.
`.hasClass(class)`	Возвращает `true`, если хотя бы один элемент в наборе имеет указанный класс.
`.html()`	Возвращает разметку HTML, содержащуюся в первом элементе из набора.
`.html(value)`	Записывает в каждый элемент из набора разметку HTML, представленную аргументом `value`.
`.text()`	Возвращает текстовое содержимое всех элементов в наборе в виде единой строки.
`.text(value)`	Записывает в каждый элемент из набора текст, представленный аргументом `value`.
`.val()`	Возвращает значение первого элемента в наборе.
`.val(value)`	Значение элемента устанавливается равным `value`.
`.css(key)`	Возвращает значение атрибута CSS с именем `key`.
`.css(key, value)`	Значение атрибута CSS `key` устанавливается равным `value`.

Метод	Описание
`.css(map)`	Значения атрибутов CSS устанавливаются в соответствии с указанным набором пар ключ-значение.
`.offset()`	Возвращает координаты в пикселях верхнего левого угла первого элемента в наборе относительно видимой области.
`.position()`	Возвращает координаты в пикселях верхнего левого угла первого элемента в наборе относительно элемента, возвращаемого методом `.offsetParent()`.
`.scrollTop()`	Возвращает позицию вертикальной прокрутки первого элемента в наборе.
`.scrollTop(value)`	Устанавливает позицию вертикальной прокрутки всех элементов в наборе в значение `value`.
`.scrollLeft()`	Возвращает позицию горизонтальной прокрутки первого элемента в наборе.
`.scrollLeft(value)`	Устанавливает позицию горизонтальной прокрутки всех элементов в наборе в значение `value`.
`.height()`	Возвращает высоту первого элемента в наборе.
`.height(value)`	Устанавливает высоту всех элементов в наборе в значение `value`.
`.width()`	Возвращает ширину первого элемента в наборе.
`.width(value)`	Устанавливает ширину всех элементов в наборе в значение `value`.
`.innerHeight()`	Возвращает высоту первого элемента в наборе, включая ширину отступов, но не включая ширину границы.
`.innerWidth()`	Возвращает ширину первого элемента в наборе, включая ширину отступов, но не включая ширину границы.
`.outerHeight(includeMargin)`	Возвращает высоту первого элемента в наборе, включая ширину отступов, границ и при необходимости ширину полей.

Метод	Описание
.outerWidth(includeMargin)	Возвращает ширину первого элемента в наборе, включая ширину отступов, границ и при необходимости ширину полей.
.append(content)	Вставляет содержимое content в конец внутренней области каждого элемента из набора.
.appendTo(selector)	Вставляет элементы из набора в конец внутренней области каждого элемента, соответствующего селектору selector.
.prepend(content)	Вставляет содержимое content в начало внутренней области каждого элемента из набора.
.prependTo(selector)	Вставляет элементы из набора в начало внутренней области каждого элемента, соответствующего селектору selector.
.after(content)	Вставляет содержимое content после каждого элемента из набора.
.insertAfter(selector)	Вставляет элементы из набора после каждого элемента, соответствующего селектору selector.
.before(content)	Вставляет содержимое content перед каждым элементом из набора.
.insertBefore(selector)	Вставляет элементы из набора перед каждым элементом, соответствующим селектору selector.
.wrap(content)	Обертывает каждый элемент в наборе содержимым content.
.wrapAll(content)	Обертывает все элементы в наборе как одно целое содержимым content.
.wrapInner(content)	Обертывает внутреннее содержимое каждого элемента в наборе содержимым content.
.replaceWith(content)	Замещает элементы из набора содержимым content.
.replaceAll(selector)	Замещает элементы, соответствующие селектору selector, элементами из набора.

Метод	Описание
`.empty()`	Удаляет дочерние узлы из каждого элемента в наборе.
`.remove([selector])`	Удаляет узлы в наборе (при необходимости отфильтрованные селектором `selector`) из дерева DOM.
`.clone([withHandlers])`	Создает копии всех элементов в наборе, при необходимости также копирует обработчики событий.
`.data(key)`	Возвращает элемент данных с именем `key`, ассоциированный с первым элементом в наборе.
`.data(key, value)`	Устанавливает элемент данных с именем `key` для всех элементов в наборе в значение `value`.
`.removeData(key)`	Удаляет элементы данных с именем `key`, ассоциированные со всеми элементами в наборе.

Методы поддержки AJAX

Используя **методы поддержки AJAX**, можно получать данные с сервера без обновления всей страницы. Подробнее методы поддержки AJAX обсуждаются в главе 6.

Метод AJAX	Описание
`$.ajax(options)`	Выполняет запрос AJAX, используя представленный набор параметров. Это низкоуровневый метод, который обычно вызывается другими, более удобными методами.
`.load(url, [data], [callback])`	Выполняет запрос AJAX по указанному адресу `url` и помещает ответ в элементы из набора.
`$.get(url, [data], [callback], [returnType])`	Выполняет запрос AJAX по указанному адресу `url` с использованием метода GET.
`$.getJSON(url, [data], [callback])`	Выполняет запрос AJAX по указанному адресу `url`; ответ интерпретируется как блок данных в формате JSON.

Метод AJAX	Описание
`$.getScript(url, [callback])`	Выполняет запрос AJAX по указанному адресу `url`; ответ запускается на выполнение как сценарий JavaScript.
`$.post(url, [data], [callback], [returnType])`	Выполняет запрос AJAX по указанному адресу `url` с использованием метода POST.
`.ajaxComplete(handler)`	Подключает обработчик `handler`, который будет вызываться в случае завершения выполнения любого запроса AJAX.
`.ajaxError(handler)`	Подключает обработчик `handler`, который будет вызываться в случае завершения выполнения любого запроса AJAX с ошибкой.
`.ajaxSend(handler)`	Подключает обработчик `handler`, который будет вызываться в случае запуска любого запроса AJAX.
`.ajaxStart(handler)`	Подключает обработчик `handler`, который будет вызываться в случае запуска любого запроса AJAX и отсутствии при этом других активных запросов.
`.ajaxStop(handler)`	Подключает обработчик `handler`, который будет вызываться в случае завершения любого запроса AJAX и отсутствии при этом других активных запросов.
`.ajaxSuccess(handler)`	Подключает обработчик `handler`, который будет вызываться в случае успешного завершения выполнения любого запроса AJAX.
`$.ajaxSetup(options)`	Устанавливает значения по умолчанию параметров для всех последующих запросов AJAX.
`.serialize()`	Преобразует значения элементов формы в строку запроса.
`.serializeArray()`	Преобразует значения элементов формы в блок данных формата JSON.
`$.param(map)`	Преобразует произвольное отображение значений в строку запроса.

Прочие методы

Следующие вспомогательные методы не могут быть отнесены ни к одной из категорий выше, но они часто бывают полезны в сценариях, использующих библиотеку jQuery.

Метод или свойство	Описание
`$.support`	Возвращает отображение свойств, позволяющих определить, поддерживает ли броузер различные особенности и стандарты.
`$.each(collection, callback)`	Выполняет итерации по коллекции `collection` и для каждого элемента вызывает функцию `callback`.
`$.extend(target, addition, ...)`	Изменяет объект `target`, добавляя в него свойства из других объектов, представленных аргументами.
`$.grep(array, callback, [invert])`	Фильтрует массив `array` с помощью функции `callback`.
`$.makeArray(object)`	Преобразует объект `object` в массив.
`$.map(array, callback)`	Создает новый массив, куда помещаются результаты функции `callback`, вызываемой для каждого элемента.
`$.inArray(value, array)`	Определяет, присутствует ли значение `value` в массиве `array`.
`$.merge(array1, array2)`	Объединяет содержимое массивов `array1` и `array2`.
`$.unique(array)`	Удаляет из массива `array` дубликаты элементов DOM.
`$.isFunction(object)`	Определяет, является ли объект `object` функцией.
`$.trim(string)`	Удаляет пробельные символы в конце строки `string`.
`$.noConflict([extreme])`	Возвращает псевдоним `$` в состояние, предшествовавшее загрузке библиотеки jQuery.
`.hasClass(className)`	Определяет, имеет ли хотя бы один элемент в наборе указанный класс.

Метод или свойство	Описание
`.is(selector)`	Определяет, соответствует ли хотя бы один элемент из набора указанному селектору.
`.each(callback)`	Выполняет итерации по набору элементов и для каждого из них вызывает функцию `callback`.
`.length`	Возвращает количество элементов в наборе.
`.get()`	Возвращает массив узлов DOM, соответствующих элементам в наборе.
`.get(index)`	Возвращает узел DOM, соответствующий элементу из набора с указанным индексом.
`.index(element)`	Возвращает индекс данного узла DOM в наборе элементов.

Алфавитный указатель

K

L

M

А

Б

В

Г

Д

З

И

P

C

Т

У

Ф

Ц

Ч

Э