ПРОФЕССИОНАЛЬНО О РАЗРАБОТКЕ ВЕБ-ПРИЛОЖЕНИЙ!

2-е издание

JavaScript ДЛЯ ПРОФЕССИОНАЛОВ

Практические приемы программирования на JavaScript для современных разработчиков

Джон Резиг, Расс Фергюсон, Джон Пакстон

www.williamspublishing.com

Apress®
www.apress.com

JavaScript

ДЛЯ ПРОФЕССИОНАЛОВ

ВТОРОЕ ИЗДАНИЕ

Pro JavaScript Techniques

SECOND EDITION

John Resig
Russ Ferguson
John Paxton

Apress®

JavaScript

ДЛЯ ПРОФЕССИОНАЛОВ

ВТОРОЕ ИЗДАНИЕ

Джон Резиг
Расс Фергюсон
Джон Пакстон

ВИЛЬЯМС

Москва • Санкт-Петербург • Киев
2016

ББК 32.973.26-018.2.75
Р34
УДК 681.3.07

Издательский дом "Вильямс"
Зав. редакцией *С.Н. Тригуб*

Перевод с английского и редакция *И.В. Берштейна*

По общим вопросам обращайтесь в Издательский дом "Вильямс" по адресу:
info@williamspublishing.com, http://www.williamspublishing.com

Резиг, Джон, **Фергюсон**, Расс, **Пакстон**, Джон.

Р34 JavaScript для профессионалов, 2-е изд. : Пер. с англ. — М. : ООО "И.Д. Вильямс", 2016. — 240 с. : ил. — Парал. тит. англ.

ISBN 978-5-8459-2054-6 (рус.)

ББК 32.973.26-018.2.75

Научно-популярное издание

Джон Резиг, Расс Фергюсон, Джон Пакстон

JavaScript для профессионалов,

2-е издание

Литературный редактор *И.А. Попова*
Верстка *М.А. Удалов*
Художественный редактор *Е.П. Дынник*
Корректор *Л.А. Гордиенко*

Подписано в печать 03.12.2015. Формат 70x100/16.
Гарнитура Times.
Усл. печ. л. 19,35. Уч.-изд. л. 13,6.
Тираж 500 экз. Заказ № 7422

Отпечатано способом ролевой струйной печати
в АО «Первая Образцовая типография»
Филиал «Чеховский Печатный Двор»
142300, Московская область, г. Чехов, ул. Полиграфистов, д.1

ООО "И. Д. Вильямс", 127055, г. Москва, ул. Лесная, д. 43, стр. 1

ISBN 978-5-8459-2054-6 (рус.)
ISBN 978-1-43-026391-3 (англ.)

Оглавление

Содержание

Посвящения

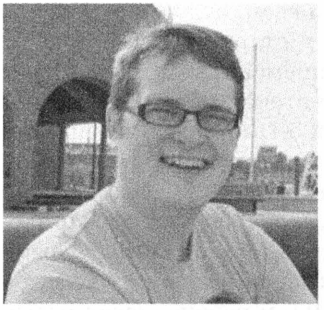
Я буду всегда благодарен моему отцу. Даже теперь он служит для меня образцом, по которому я учусь. Нам не хватает вас, отец и Родд. Смотреть сериалы “Доктор Кто” и “Сумеречная зона” никогда не означало одно и то же.

Расс Фергюсон

Адреине, которая всегда верила.

Джон Пакстон

Об авторах



Джон Резиг работает разработчиком в Академии Хана и является создателем библиотеки jQuery для JavaScript. Помимо данной книги, он является автором книги *Secrets of the JavaScript Ninja* (издательство Manning, 2012 г.; в русском переводе эта книга вышла под названием *Секреты JavaScript ниндзя* в ИД "Вильямс", 2013 г.). Джон работает также внештатным научным сотрудником в университете города Киото, где он изучает Укиё-э — искусство японской гравюры на дереве, иначе называемой ксилографией. Он разработал обширную базу данных и механизм поиска ксилографических изображений, доступных по адресу `http://ukiyo-e.org`.

Расс Фергюсон работает разработчиком и инструктором в районе Нью-Йорка. В настоящее время он руководит компанией SunGard Consulting Services, занимающейся разработкой приложений для таких клиентов, как Morgan Stanley и Comcast. Многие годы Расс преподает в институте имени Пратта и школе дизайна имени Парсонса. Он разработал приложения как для начинающих, так и для упрочившихся организаций вроде Chase Bank, Publicis Groupe, DC Comics и MTV/Viacom. В число его интересов входит поощрение молодого поколения к программированию и освоению технологических способов, позволяющих изменить употребление средств массовой информации и участие в них, а также упражнения в японском языке, писательство, кино, концерты, коллекционирование вин и саке. В Tweeter его можно найти по адресу `@asciibn`.

Джон Пакстон является программистом, инструктором, автором книг и презентатором, проживающим в своем родном штате Нью-Джерси. Изучая историю в университете имени Джона Хопкинса, он обнаружил, что проводил больше времени в компьютерном классе, чем в архивах документов. С тех пор его интересы разделялись между программированием и преподаванием, и за последние пятнадцать лет ему пришлось программировать на самых разных языках, применяемых в веб-разработке. В настоящее время Джон остановил свой выбор на языках JavaScript и Java, хотя иногда он испытывает ностальгические порывы к Perl и XML. С ним можно связаться в Twitter по адресу `@paxtonjohn`, а также на его веб-сайте по адресу `speedingplanet.com`.

О технических рецензентах

Иан Девлин интересуется всем, что касается веб, и в настоящее время работает старшим веб-разработчиком в агентстве, базирующемся в немецком городе Дюссельдорф. Он является знатоком HTML5 и одним из основателей веб-сайта HTML5 Hub, организованного компанией Intel для сообщества веб-разработчиков, где (а также в журнале *net*) он опубликовал целый ряд статей, посвященных вопросам разработки веб-приложений для сфер деятельности компаний Mozilla, Opera, Intel и Adobe Systems. Он также написал книгу по мультимедийным средствам HTML5 и рецензировал целый ряд книг, вышедших в издательстве Apress.

Марк Ненадов имеет более чем 15-летний опыт разработки программного обеспечения, главным образом технологий с открытым исходным кодом. Он проживает в г. Эссекс, что в канадской провинции Онтарио, со своей дорогой женой и двумя обожаемыми дочерьми, ожидая еще и сына. В свободное от разработки программного обеспечения или семейных дел время Марк любит путешествовать автостопом, наблюдать дикую природу, читать, изучать историю, рецензировать и редактировать чужие рукописи или писать что-нибудь свое. Он является заядлым поэтом, и его стихотворения публиковались в Соединенных Штатах, Канаде, Пакистане, Индии, Австралии, Англии и Ирландии.

Благодарности

Как всегда, мне хотелось бы поблагодарить многих людей, в том числе сотрудников издательства Apress, а среди них Луизу Корриган (Louise Corrigan), Риту Фернандо (Rita Fernando) и Кристину Рикеттс (Christine Ricketts). Без них у меня не было бы возможности работать над этой книгой. Благодарю также других авторов данной книги, Джона Резига и Джона Пакстона, знания и опыт которых отражены на ее страницах.

Выражаю благодарность техническим рецензентам Марку Ненадову и Иану Девлину за то, что они помогли нам, авторам книги, сделать ее более совершенной и прояснить свои намерения. Благодарю свою семью за моральную поддержку во время работы над книгой, когда мне приходилось дольше обычного сидеть за компьютером.

Расс Фергюсон

Мне не удалось бы внести свой посильный вклад в написание этой книги без веры, терпения, стойкости и наставничества со стороны Луизы Корриган и Кристины Рикеттс. Обе они относились к задержкам в работе над книгой и запаздыванию в написании отдельных глав намного более терпеливо, чем это обычно принято у людей.

Джон Пакстон

От издательства

Вы, читатель этой книги, и есть главный ее критик и комментатор. Мы ценим ваше мнение и хотим знать, что было сделано нами правильно, что можно было сделать лучше и что еще вы хотели бы увидеть изданным нами. Нам интересно услышать и любые другие замечания, которые вам хотелось бы высказать в наш адрес.

Мы ждем ваших комментариев и надеемся на них. Вы можете прислать нам бумажное или электронное письмо либо просто посетить наш веб-сайт и оставить свои замечания там. Одним словом, любым удобным для вас способом дайте нам знать, нравится или нет вам эта книга, а также выскажите свое мнение о том, как сделать наши книги более интересными для вас.

Посылая письмо или сообщение, не забудьте указать название книги и ее авторов, а также ваш обратный адрес. Мы внимательно ознакомимся с вашим мнением и обязательно учтем его при отборе и подготовке к изданию последующих книг.

Наши электронные адреса:

E-mail: `info@williamspublishing.com`

WWW: `http://www.williamspublishing.com`

Наши почтовые адреса:

в России: 127055, г. Москва, ул. Лесная, д.43, стр. 1

в Украине: 03150, Киев, а/я 152

ГЛАВА 1

Профессиональные методики программирования на JavaScript

Предлагаем читателям ознакомиться с профессиональными методиками программирования на JavaScript. В этой книге дается краткий обзор текущего состояния языка JavaScript, особенно его применения профессиональными программистами. А кто такой профессиональный программист? Это человек, имеющий твердые навыки основ программирования на JavaScript, а возможно, и на ряде других языков, но заинтересованный в расширении и углублении своих навыков программирования на JavaScript. Для этого ему придется рассмотреть не только типичные функциональные средства вроде объектной модели документов (DOM), но и разобраться с шаблоном проектирования "модель–представление–онтроллер" (MVC) на стороне клиента. Поэтому в данной книге рассматриваются обновленные прикладные программные интерфейсы (API), новые функциональные средства и возможности, а также творческие способы применения написанного кода.

Это второе издание данной книги. С момента выхода в свет первого ее издания в 2006 году произошло немало перемен. За это время язык JavaScript претерпел болезненный переход от игрушечного состояния к полноценному языку написания сценариев, который стал полезным и эффективным для решения самых разных задач. Если хотите, язык JavaScript пребывал в зачаточном состоянии, а теперь он находится в конце другого перехода, который столь же метафорично можно охарактеризовать как взросление. Ныне язык JavaScript применяется повсеместно (по некоторым оценкам — на 85–95% всех действующих веб-сайтов), в том числе и на начальных страницах. В связи с этим многие считают JavaScript самым распространенным в мире языком программирования, а число тех, кто его применяет на практике, постоянно растет. Но важнее другое: эффективность и возможности этого языка.

Язык JavaScript вырос из игрушечного языка (динамических подстановок и манипуляций со строкой состояния) в эффективное средство программирования, хотя и с ограниченными возможностями (достаточно вспомнить о проверке достоверности на стороне клиента), достигнув своего текущего состояния с расширенными

возможностями программирования, не ограничивающимися только браузерами. Программисты пишут на JavaScript инструментальные средства, обеспечивающие функциональные возможности шаблона MVC, которые долгое время относились к области действия сервера, а также средства для воспроизведения сложных данных, библиотеки шаблонов и многое другое. И этот перечень можно продолжить. Если в прошлом разработчики опирались на программную платформу .NET и клиент Java Swing для обеспечения полноценного, функционально насыщенного интерфейса к данным на стороне сервера, то теперь они могут реализовать то же самое в виде приложения JavaScript для браузера. А применяя программную платформу Node.js, нетрудно создать на JavaScript собственную версию виртуальной машины, на которой можно выполнять любое количество разных приложений, все из которых написаны на JavaScript и не требуют для этого браузера.

В этой главе описывается, как было достигнуто текущее состояние языка Java Script, а также перспективы его дальнейшего развития. В ней рассматриваются различные усовершенствования в технологии браузеров, распространенность которых способствовала революционному развитию JavaScript. Текущее состояние JavaScript требует обследования, чтобы осознать свое положение, прежде чем двигаться дальше. А в последующих главах будет показано, что нужно знать для профессионального программирования на JavaScript.

Как было достигнуто текущее состояние JavaScript

На момент выхода в свет первого издания данной книги браузеры Google Chrome и Mozilla Firefox только появились на рынке, где бал правили версии 6 и 7 браузера Internet Explorer, а его версия 8 только начинала завоевывать популярность. Стимулами для дальнейшей разработки JavaScript послужило несколько факторов.

Большую часть своей истории развития язык JavaScript зависел от браузера. Ведь браузер служил средой выполнения для сценариев JavaScript, а доступ программиста к функциональным возможностям JavaScript сильно зависел от марки, модели и версии браузера, использовавшегося для посещения веб-сайта, разрабатываемого данным программистом. К средине 2000-х годов войны, бушевавшие между браузерами в 1990-е годы, завершились полной победой браузера Internet Explorer, и на этом дальнейшее их развитие временно остановилось. Но это положение дел вскоре изменилось благодаря появлению браузеров Mozilla Firefox и Google Chrome. Первый из них стал наследником Netscape — одного из самых первых браузеров, а второй — сразу же важным игроком на сцене ввиду достаточно сильной поддержки со стороны компании Google.

В обоих этих браузерах был принят ряд проектных решений, способствовавших революционному развитию JavaScript. Первое решение состояло в поддержке различных стандартов, внедренных Консорциумом Всемирной паутины (W3C — World Wide Web Consortium). Браузеры Chrome и Firefox обычно следовали спецификации этих стандартов, реализуя ее как можно точнее, будь то модель DOM, обработка событий или технология Ajax. Для программистов это означало, что им не нужно было больше писать отдельный код специально для браузеров Chrome и Firefox. Они уже привыкли писать отдельный код для Internet Explorer и какого-нибудь другого браузера, и поэтому ветвление кода по типам браузеров для них не было внове. Но возможность упростить такое ветвление принесла им долгожданное облегчение.

Что касается стандартов, то в браузерах Firefox и Chrome было воплощено плодотворное сотрудничество их создателей с Европейской ассоциацией производителей компьютеров (European Computer Manufacturer's Association — ECMA, а теперь — Ecma). Эта ассоциация является стандартизирующим органом, надзирающим над JavaScript, а формально — над соблюдением стандарта ECMAScript, поскольку обозначение JavaScript стало торговой маркой компании Oracle, хотя эти подробности не так и важны. В дальнейшем мы будем пользоваться обозначением JavaScript для ссылки на язык программирования, а обозначением ECMAScript — на спецификацию, к которой привязана реализация этого языка. Стандарты ECMAScript были постепенно преданы забвению, как, впрочем, и разработка браузера Internet Explorer. Но как только возросла конкуренция между браузерами, стандарты ECMAScript снова стали актуальными. Так, в версии 5 стандарта ECMAScript (2009 года) были законодательно закреплены многие изменения, которые произошли за десять лет после выпуска предыдущей версии этого стандарта. Но на этом дело не остановилось, и в 2011 году вышла версия 5.1. А в будущем предстоит еще немало работы для версиями 6 и 7 данного стандарта.

Следует воздать должное браузеру Chrome, поскольку его появление также способствовало дальнейшему развитию JavaScript. Механизм JavaScript в браузере Chrome, называемый V8, стал очень важной частью успешного дебюта Chrome в 2008 году. Разработчики браузера Chrome создали механизм, который действовал намного быстрее, чем большинство других механизмов JavaScript, и следовал этой главной цели в последующих версиях. В действительности возможности механизма V8 оказались настолько впечатляющими, что он был положен в основу программной платформы Node.js в качестве интерпретатора JavaScript, независящего от конкретного браузера. Первоначально платформа Node.js предназначалась в качестве сервера, который должен был пользоваться JavaScript как своим главным прикладным языком, но в дальнейшем она стала гибкой платформой для выполнения любого количества JavaScript-ориентированных приложений.

Если вернуться к браузеру Chrome, то другим нововведением, внедренным в области браузеров, стало понятие *возобновляемого приложения*. Вместо того чтобы загружать отдельные обновления для установки вручную, браузер Chrome обновляется автоматически. И хотя такой подход иногда оказывается не совсем удобным в корпоративном мире, он является большим благом для некорпоративных (т.е. индивидуальных) пользователей Интернета. Если вы пользуетесь браузером Chrome, а в последние годы — браузером Firefox, то ваш браузер обновляется автоматически, не требуя от вас никаких дополнительных усилий. И несмотря на то что корпорация Microsoft уже давно продвигает обновления системы безопасности через службу Windows Update, она так и не удосужилась внедрять новые функциональные средства в браузер Internet Explorer, если только они не сопутствуют новой версии Windows. Иными словами, обновления браузера Internet Explorer происходят медленно, тогда как браузеры Chrome и Firefox всегда имеют самые последние, совершенные и довольно безопасные функциональные средства.

Как только компания Google стала делать акцент на функциональных средствах браузера Chrome, ее примеру последовали остальные производители браузеров. Иногда дело даже доходило до смешного, как, например, в случае с привязкой браузера Firefox к порядку нумерации версий браузера Chrome. Но в то же время это вынудило разработчиков из компании Mozilla и корпорации Microsoft трезво и хладнокровно анализировать действие механизмов JavaScript. Оба эти производителя браузеров значительно усовершенствовали свои механизмы JavaScript за последние

годы, и некогда бесспорное лидерство браузера Chrome в данной области теперь кажется не таким уж неоспоримым.

И наконец, корпорация Microsoft практически отказалась от своего классического принципа "охвата и расширения" — по крайней мере, в отношении языка JavaScript. В версии браузера Internet Explorer 9 был внедрен механизм обработки событий, предложенный консорциумом W3C, а также стандартизированы его интерфейсы DOM и прикладные программные интерфейсы Ajax API. Что же касается большинства стандартных функциональных средств JavaScript, то теперь уже нет никакой необходимости реализовывать две версии одного и того же кода, хотя унаследованный код для устаревших браузеров по-прежнему вызывает определенные трудности.

И это стало почти панацеей. Ведь быстродействие JavaScript теперь высоко, как никогда прежде. А писать прикладной код для разных браузеров стало много проще. Стандарты описывают сложившееся положение дел и дают полезные ориентиры для разработки функциональных средств в перспективе. И большинство современных браузеров обновляются автоматически. О чем же беспокоиться теперь и куда двигаться дальше?

Современное состояние JavaScript

Разрабатывать серьезные приложения на JavaScript теперь стало как никогда проще. Мы окончательно и бесповоротно порвали с прошлым, когда требовалось писать отдельный код для разных браузеров, а также реализовывать неудачные стандарты и медленные механизмы JavaScript, что нередко было запоздалой мыслью. Рассмотрим современное состояние среды JavaScript, в частности, следующие две области: современные браузеры и инструментальные средства.

Современное состояние языка JavaScript зависит от понятия современного браузера. А что такое современный браузер? В разных организациях это понятие описывается по-разному. Так, в компании Google заявляют, что их приложения поддерживают текущие и основные предыдущие версии браузеров. (Любопытно, что служба Gmail до сих пор нормально работает в браузере Internet Explorer 9, насколько мы можем судить!) В интересной статье люди, имеющие отношение к веб-сайту Би-би-си, открыто заявили, что современным они считают такой браузер, который поддерживает следующие возможности.

1. Функции `document.querySelector()` / `document.querySelectorAll()`.
2. Функцию `window.addEventListener()`.
3. Прикладной программный интерфейс Storage API (локальное хранилище типа `localStorage` и сеансовое хранилище типа `sessionStorage`).

Вероятно, самой распространенной в веб является библиотека jQuery для JavaScript. В ее версии 1.x поддерживается браузер Internet Explorer, начиная с версии 6, а в версии 2.x — современные браузеры вроде Internet Explorer 9 и более поздних версий. И можно безошибочно сказать, что браузер Internet Explorer служит своего рода водоразделом между прошлым и современным в данной области. Две другие основные разновидности браузеров являются возобновляемыми. И хотя браузеры Safari и Opera не являются возобновляемыми, они обновляются более быстрыми темпами, чем браузер Internet Explorer, хотя их доля на рынке несравнимо мала.

Так где же проходит граница для современных браузеров? Увы, она проходит где-то между версиями 9 и 11 браузера Internet Explorer, тогда как версия 8 уже находится на дальнем краю истории развития браузеров, поскольку в ней не поддерживается ни большинство функциональных средств по стандарту ECMAScript 5, ни прикладной программный интерфейс API, предложенный консорциумом W3C для обработки событий. И этот перечень можно продолжить. Следовательно, когда мы будем обсуждать современные браузеры, то упомянем, по крайней мере, версию Internet Explorer 9, но не прежние версии этого браузера. А там, где уместно и просто, мы укажем на полизаполнения для прежних версий браузера Internet Explorer, хотя нижней границей, отделяющей старые браузеры от современных, будет все-таки служить версия Internet Explorer 9.

Рост популярности библиотек

Помимо современного браузера, имеется еще одно важное свойство, определяющее современное состояние среды JavaScript: библиотеки. За последние восемь лет наметился бурный рост различных библиотек JavaScript. Так, в информационном хранилище GitHub уже насчитывается более 800 тысяч библиотечных записей для JavaScript, 900 из которых пользуются наибольшей популярностью. Начав свое существование со скромных коллекций служебных функций, экосистема библиотек JavaScript развилась (в какой-то степени хаотично) в обширный ландшафт возможностей.

Как это оказывает влияние на тех, кто разрабатывает приложения на JavaScript? Разумеется, существует модель "библиотеки в качестве расширения", где библиотека предоставляет дополнительные функциональные возможности. В этом отношении следует упомянуть библиотеки по шаблону MVC вроде Backbone и Angular, которые будут рассмотрены далее в этой главе, или библиотеки визуализации данных вроде d3 или Highcharts. Но язык JavaScript находится в интересном положении, поскольку библиотеки могут также предоставить на уровне интерфейса функциональные средства, которые являются стандартными в одних браузерах и нестандартными в других.

Долгое время стандартным примером разнообразно реализуемого в JavaScript функционального средства была обработка событий. Так, у браузера Internet Explorer имеется свой прикладной программный интерфейс API для обработки событий, а в других браузерах для этой цели, как правило, применяется прикладной программный интерфейс API, предложенный консорциумом W3C. Различные библиотеки предоставляют единообразные реализации для обработки событий, включая и обоюдовыгодное решение. Некоторые из этих библиотек стоят обособленно, но наиболее успешные из них предоставляют также нормализованные функциональные возможности для технологии Ajax, модели DOM, а ряд других функциональных средств по-разному реализован в браузерах.

Наиболее распространенной из них является библиотека jQuery. С самого начала библиотека jQuery была самой удобной для применения новых функциональных средств JavaScript безотносительно к их поддержке в браузерах. Поэтому вместо того чтобы обращаться к прикладным программным интерфейсам браузера Internet Explorer или консорциума W3C для обработки событий, можно просто воспользоваться функцией `.on()` из библиотеки jQuery, обеспечивающей единообразный интерфейс с учетом отличий в реализации. Аналогичные функциональные возможности предоставляют и другие библиотеки, в том числе Dojo, Ext JS, Prototype, YUI,

MooTools и т.д. Эти библиотеки инструментальных средств нацелены на стандартизацию прикладных программных интерфейсов API для разработчиков.

Стандартизация простирается дальше, чем обеспечение простого кода ветвления. Такие библиотеки нередко улучшают дефектные реализации. Официально прикладной программный интерфейс API для функции может не претерпевать особых изменений при смене версий, но в нем имеются программные ошибки. В одних случаях они исправляются, в других — не исправляются, а в третьих — исправления вносят новые программные ошибки. Например, более полдесятка исправлений, внесенных в версии 1.11 библиотеки jQuery, относятся к прикладному программному интерфейсу API для обработки событий.

Одни библиотеки (в частности, jQuery) предоставляют также новые или разные интерпретации определенных возможностей. Так, функция селектора из библиотеки jQuery, положенная в ее основу, предшествовала теперь уже ставшим стандартными функциям `querySelector()` и `querySelectorAll()` и послужила стимулом для их включения в состав JavaScript. А в других библиотеках доступ к функциональным возможностям предоставляется, несмотря на самые разные реализации. Далее в этой книге будет рассмотрен новый для Ajax протокол CORS (Cross Origin Resource Sharing — Совместное использование ресурсов из разных источников), позволяющий делать Ajax-запросы других серверов, а не только того сервера, который первоначально обслужил страницу. И в третьих библиотеках уже реализована версия такой функциональной возможности, в которой используется протокол CORS, но там, где требуется, в качестве запасного варианта употребляется вариант формата JSON с заполнением (JSON-P).

В силу своего служебного характера некоторые библиотеки стали частью набора инструментальных средств для тех, кто профессионально программирует на JavaScript. Функциональные средства таких библиотек могут быть (пока еще) не стандартизированы в JavaScript, но в них накоплены знания и функциональные возможности, которые просто ускоряют реализацию проектных решений. Тем не менее за последние годы возникла полемика, действительно ли jQuery и другие библиотеки так нужны для разработки веб-приложений в современных браузерах. Рассмотрим для примера упомянутые выше требования Би-би-си. Безусловно, большую часть функциональных возможностей можно реализовать самостоятельно, если иметь в своем распоряжении три перечисленные выше функции. Но ведь библиотека jQuery содержит также упрощенный, хотя и расширенный интерфейс DOM; она справляется с программными ошибками в самых разных крайних случаях; а если потребуется поддержка браузера Internet Explorer 8 или более ранней версии, то без библиотеки jQuery просто не обойтись. Таким образом, те, кто профессионально программирует на JavaScript, должны рассматривать требования к проекту с учетом оправданности риска изобрести подобно колесу то, что уже предоставляет jQuery или аналогичная библиотека.

О поддержке мобильных устройств

В прежней литературе по JavaScript и веб-разработке можно было обнаружить лишь часть, а возможно, и целую главу, посвященную просмотру веб-содержимого на мобильных устройствах. Такой просмотр составлял лишь малую долю общего просмотра веб-содержимого, а рынок был настолько раздроблен, что разработкой приложений для мобильных устройств интересовались только специалисты. Но теперь дело обстоит совсем иначе. С момента выхода в свет первого издания данной

книги просмотр веб-содержимого на мобильных устройствах получил широкое развитие и стал совершенно другой отраслью, чем разработка настольных приложений. Согласно статистике, взятой из самых разных источников, просмотр веб-содержимого на мобильных устройствах теперь составляет 20–30% от всего просмотра, а к тому моменту, когда вы будете читать данную книгу, его доля может стать еще больше, поскольку она постоянно растет с того времени, как появились мобильные устройства вроде iPhone. В настоящее время более 40% всего просмотра веб-содержимого на мобильных устройствах приходится на браузер Safari под iOS, хотя и браузер Chrome под Android становится все более популярным, а по некоторым статистическим данным он даже превосходит браузер Safari. Версия браузера Safari для iOS отличается от его настольной версии, как, впрочем, и версии браузеров Chrome и Firefox для Android. Таким образом, поддержка мобильных устройств стала общей тенденцией.

Браузеры в мобильных устройствах представляют не только новые возможности, но и трудности. С одной стороны, мобильные устройства нередко обладают более ограниченными ресурсами, чем настольные системы, хотя этот пробел быстро восполняется. А с другой стороны, мобильные устройства предоставляют новые возможности (события проводки пальцем по экрану, более точную геолокацию и т.д.) и принципы взаимодействия (использование руки вместо мыши, проводки пальцем для прокрутки). В зависимости от конкретных требований теперь приходится разрабатывать приложение, которое должно быть вполне пригодным как для мобильных устройств, так и для настольных систем, или специально реализовывать уже имеющиеся функциональные возможности на мобильной платформе.

Ландшафт JavaScript сильно изменился за последние годы. Несмотря на некоторую стандартизацию прикладных программных интерфейсов API, возникло немало новых трудностей. Как это все повлияет на тех, кто профессионально программирует на JavaScript?

Дальнейшие перспективы

Мы сами должны установить некоторые нормы. И мы уже установили одну норму: браузер Internet Explorer 9 в качестве нижней границы для взаимодействия с современными браузерами, а другие браузеры возобновляются и не требуют вмешательства. А что же тогда поддержка мобильных устройств? Несмотря на сложность этого вопроса, версии iOS 6 и Android 4.1 (Jelly Bean), как правило, служат нижней границей. Вычисления на мобильных устройствах обновляются быстрее и чаще, чем в настольных системах, и поэтому этими более поздними версиями мобильных операционных систем можно пользоваться с большей уверенностью.

Отвлечемся на время от норм, чтобы обсудить не версии браузеров, операционных систем или платформ, а круг потенциальных пользователей. Самой ценной является такая статистика, которая сообщает о *конкретном*, а не общем круге потенциальных пользователей. Допустим, что веб-приложение разрабатывается для заказчика, принявшего за норму браузер Internet Explorer 10. А возможно, идея разработать приложение сильно зависит от функциональных средств, которые предоставляет только браузер Chrome. Или, может быть, версия приложения для настольной системы не предполагается, но оно предназначено для развертывания на планшетных компьютерах iPad и Android. Следовательно, нужно непременно учитывать целевой круг пользователей. В этой книге рассматривается более широкий круг вопросов, а не только разработка веб-приложений. Но было бы недальновидно

тратить зря время на рассмотрение программных ошибок в браузере Internet Explorer 9, которые могут повлиять на разработку упомянутого выше приложения для планшетного компьютера, не так ли? А теперь вернемся к вопросу установления собственных норм.

В качестве иллюстраций и примеров тестирования в данной книге отдается предпочтение браузеру Google Chrome. Но там, где это уместно, на ее страницах иногда демонстрируется выполнение кода в браузере Firefox или Internet Explorer. Для разработчиков браузер Chrome является золотым стандартом, совсем не обязательно удобным для пользователей, но, безусловно, информационно открытым для программистов. В одной из глав этой книги будут рассмотрены различные инструментальные средства, доступные для разработчиков, и не только для браузера Chrome, но и для Firefox (как с дополнением Firebug, так и без него), и Internet Explorer.

В качестве стандартной библиотеки в данной книге выбрана библиотека jQuery. Разумеется, возможны и другие варианты выбора, но предпочтение отдано библиотеке jQuery по следующим причинам. Во-первых, это наиболее распространенная библиотека общего назначения для разработки веб-приложений на JavaScript. И во-вторых, один из авторов данной книги (Джон Резиг) имеет непосредственное отношение к созданию библиотеки jQuery, что навело другого автора (Джона Пакстона) на мысль поработать с этой библиотекой. Обновляя издание настоящей книги, мы решили заменить методики, рассматривавшиеся в предыдущем издании, функциональными возможностями библиотеки jQuery. В подобных случаях мы постарались не изобретать снова колесо, а по мере надобности ссылаться на соответствующие функциональные возможности библиотеки jQuery. И, конечно, на страницах данной книги мы обсудим также новые и перспективные методики!

Интегрированные среды разработки (ИСР) веб-приложений на JavaScript значительно обновились за последние годы под действием собственного развития языка JavaScript. Возможности ИСР стали настолько обширными, что перечислить их здесь просто невозможно, хотя следует упомянуть о применении некоторых из них. Так, Джон Резиг пользуется специализированной версией текстового редактора Vim в качестве своей среды разработки. А Джон Пакстон как более ленивый выбрал в качестве ИСР отличный продукт WebStorm компании JetBrains (`http://www.jetbrains.com/webstorm/`). Компания Adobe Systems предлагает свободно доступную ИСР Brackets с открытым исходным кодом (`http://brackets.io/`) в ее текущей версии 1.3. Имеется также ИСР Eclipse, а многие разработчики положительно отзываются о специальном применении текстовых редакторов SublimeText и Emacs в своей практике. Но, как всегда, вы вольны пользоваться той ИСР, которая вам больше всего подходит.

Имеются и другие инструментальные средства, помогающие разрабатывать веб-приложения на JavaScript. Но вместо того чтобы перечислять их здесь, мы посвятили им отдельную главу данной книги. А это означает, что настало время сделать краткий обзор ее содержания.

Краткое содержание остальной части книги

Начиная с главы 2 мы рассмотрим самые последние и наиболее примечательные возможности языка JavaScript. Это не только новые языковые средства, доступные через тип `Object`, но и прежние понятия вроде ссылок, функций, области действия

и замыканий. Весь этот материал собран под единым заглавием "Языковые средства, функции и объекты" данной главы, хотя в ней рассматриваются не только они.

Глава 3 посвящена созданию повторно используемого кода. В связи с тем что в главе 2 не рассматривается метод `Object.create()`, этому едва ли не самому крупному нововведению в JavaScript, а также его последствиям для объектно-ориентированного кода JavaScript уделяется внимание в главе 3, где также рассматриваются функциональные конструкторы, прототипы и принципы объектно-ориентированного программирования (ООП), реализованные в JavaScript.

Посвятив две отдельные главы разработке прикладного кода, мы должны подумать о том, как управлять им. Поэтому в главе 4 "Отладка кода JavaScript" рассматриваются соответствующие инструментальные средства. Сначала в ней исследуются браузеры и их инструментальные средства разработки.

Глава 5 "Объектная модель документов" начинается с обсуждения некоторых наиболее употребительных из функциональных возможностей JavaScript, включая модель DOM. И хотя прикладной программный интерфейс DOM API стал сложнее и не упростился с момента выхода в свет первого издания данной книги, с тех пор появились новые средства, с которыми нужно ознакомить читателей.

В главе 6 "События" мы предпринимаем попытку овладеть событиями. Нам приятно сообщить, что стандартизация прикладного программного интерфейса API для событий осуществлена в соответствии с рекомендациями консорциума W3C. Это дает возможность отказаться от служебных библиотек и наконец-то обратиться непосредственно к прикладному программному интерфейсу API для событий, не особенно беспокоясь о том, насколько браузеры отличаются друг от друга.

Одним из неигрушечных применений JavaScript стала проверка достоверности форм на стороне клиента. Поразительно, но производителям браузеров потребовалось около десятилетия на раздумывание о необходимости внедрить функциональные возможности для проверки достоверности форм, помимо фиксации события передачи формы. Поэтому в главе 7 "JavaScript и проверка достоверности форм" мы обсуждаем совершенно новые функциональные возможности, предоставляемые для проверки достоверности форм как в HTML, так и в JavaScript.

Всем, кто разрабатывает веб-приложения на JavaScript, полезно будет ознакомиться с материалом главы 8 "Введение в Ajax". В ней представлены функциональные возможности протокола CORS и технологии Ajax, а также способы преодоления их наиболее нелепых ограничений.

В главе 9 "Инструментальные средства для веб-производства" рассматриваются такие инструментальные средства командной строки, как Yeoman, Bower, Git и Grunt. Эти средства демонстрируют, насколько быстро вводятся все нужные файлы и папки. Благодаря этому можно сосредоточиться непосредственно на разработке.

В главе 10 "AngularJS и тестирование", опираясь на знания, приобретенные в предыдущих главах, мы поясняем принцип действия библиотеки AngularJS и порядок реализации как модульного, так сквозного тестирования.

И наконец, в главе 11 "Перспективы развития JavaScript" рассматриваются перспективы дальнейшего развития языка JavaScript. К моменту выхода этого издания данной книги в свет версия стандарта ECMAScript 6 более или менее упрочится, тогда как работа над версией стандарта ECMAScript 7 будет только вестись, хотя и активно. Помимо основных перспектив дальнейшего развития языка JavaScript, в этой главе рассматриваются те языковые средства, которыми можно пользоваться уже теперь.

Резюме

Большая часть этой главы была посвящена всему, что окружает язык JavaScript: платформам, истории развития, ИСР и т.д. Мы считаем, что история объясняет настоящее, и поэтому хотели пояснить текущее состояние языка JavaScript и как оно было достигнуто, чтобы стало понятнее его назначение и что он представляет собой в настоящее время. Разумеется, мы собираемся посвятить немалую часть данной книги обсуждению внутреннего механизма работы JavaScript, особенно для профессиональных программистов. Мы считаем своим долгом изложить методики и прикладные программные интерфейсы API, которые должны знать все, кто профессионально программирует на JavaScript. Итак, приступим к делу без лишнего шума.

ГЛАВА 2

Языковые средства, функции и объекты

Объекты являются основополагающими единицами языка JavaScript. Буквально все в JavaScript является объектом и взаимодействует на объектно-ориентированном уровне. Чтобы построить язык JavaScript на строгом объектно-ориентированном основании, в него следует включить арсенал языковых средств, которые делают особенными как его основание, так и его возможности.

В этой главе обсуждаются некоторые из самых важных особенностей языка JavaScript, в том числе ссылки, область действия, замыкания и контекст. Они совсем не обязательно являются краеугольными камнями данного языка, но в то же время служат изящными арками, поддерживающими и совершенствующими свод JavaScript. Далее в главе мы углубленно рассмотрим объектно-ориентированный характер языка JavaScript, включая обсуждение классов в сравнении с прототипами. И наконец, будет показано, как пользоваться объектно-ориентированными возможностями JavaScript, включая создание новых объектов и их поведение. Вполне возможно, что большая часть материала этой главы совершенно изменит ваше представление о языке JavaScript, если вы отнесетесь к нему серьезно.

Языковые средства

В языке JavaScript имеется немало средств, которые составляют основу данного языка. Такими средствами обладают немногие языки программирования. Зачастую в языке обнаруживается определенное сочетание подходящих средств, которые создают обманчивое представление о его эффективности.

Ссылки и значения

Данные в переменных JavaScript хранятся одним из двух способов: в виде копий и ссылок. Все, что является значением примитивного типа, *копируется* в переменную. К примитивным типам относятся символьные строки, числа, логические, пустые и неопределенные значения. К наиболее примечательным характеристикам

примитивных типов относится их присваивание, копирование, передача и возврат из функций по *значению*.

А в остальном JavaScript опирается на ссылки. Любая переменная, в которой не хранится значение одного из упомянутых выше примитивных типов, по существу, содержит *ссылку* на объект. *Ссылка* является указателем на ячейку оперативной памяти, выделяемой для объекта (массива, даты или чего-нибудь другого). Конкретный объект (массив, дата или что-нибудь другое) называется *объектом ссылки*. Это невероятно эффективное языковое средство, присутствующее во многих языках программирования. Оно позволяет достичь определенной эффективности в том, что две (или больше) переменные не содержат свои копии на один и тот же объект, а просто ссылаются на него. Обновление объекта ссылки по одной ссылке отражается на другой ссылке. Благодаря сохранению ряда ссылок на объекты программирование на JavaScript становится намного более гибким. Примером тому служит код, демонстрируемый в листинге 2.1, где две переменные указывают на один и тот же объект, а модификация содержимого этого объекта по одной ссылке отражается на другой ссылке.

Листинг 2.1. Пример ссылки нескольких переменных на один объект

```
// Присвоить переменной obj пустой объект.
// Фигурные скобки {} делают код короче,
//  чем оператор 'new Object()'
var obj = {};

// Присвоить другой переменной еще одну ссылку на объект
var refToObj = obj;

// Модифицировать свойство исходного объекта
obj.oneProperty = true;

// Теперь ясно, что изменение отражается на обеих переменных,
// поскольку обе они ссылаются на один и тот же объект
console.log( obj.oneProperty === refToObj.oneProperty );

// Это изменение носит двухсторонний характер, так как обе
// переменные, obj и refToObj, содержат ссылки
refToObj.anotherProperty = 1;
console.log( obj.anotherProperty === refToObj.anotherProperty );
```

У объектов имеются следующие языковые средства: свойства и методы. Нередко эти средства совместно называют *членами* объекта. Свойства содержат данные объекта и могут быть примитивными или самими объектами. А методы являются функциями, воздействующими на данные объекта. В некоторых дискуссиях по JavaScript методы включаются в состав свойств. Но зачастую их удобнее отделять от свойств.

Самомодифицирующиеся объекты в JavaScript весьма редки. Рассмотрим один характерный тому пример. Так, в сам объект типа `Array` можно ввести дополнительные элементы, используя метод `push()`. А поскольку значения в объекте типа `Array`, по существу, хранятся в виде свойств этого объекта, то в результате возникает ситуация, аналогичная приведенной в листинге 2.1, где объект становится глобально

модифицируемым, а следовательно, содержимое нескольких переменных изменится одновременно. Пример такой ситуации приведен в листинге 2.2.

Листинг 2.2. Пример самомодифицирующегося объекта

```
// Создать массив элементов.
// Как и в примере из листинга 2.1, квадратные скобки []
//  делают код короче, чем оператор 'new Array()'
var items = [ 'one', 'two', 'three' ];

// Создать ссылку на элементы в массиве
var itemsRef = items;

// Ввести элемент в исходный массив
items.push( 'four' );

// Длина каждого массива должна остаться прежней, так как
// обе переменные указывают на один и тот же объект массива
console.log( items.length == itemsRef.length );
```

Очень важно не забывать, что ссылки указывают только на объект ссылки, но не на другую ссылку. Например, в языке Perl допускается иметь ссылку, указывающую на другую переменную, которая также содержит ссылку. Но в языке JavaScript она распространяется дальше по цепочке, указывая только на базовый объект. Пример подобной ситуации демонстрируется в листинге 2.3, где физический объект изменяется, но ссылка продолжает указывать на прежний объект.

Листинг 2.3. Изменение ссылки на объект при сохранении его целостности

```
// Присвоить переменной items массив символьных строк,
// являющийся объектом
 var items = [ 'one', 'two', 'three' ];
// Присвоить переменной itemsRef ссылку на элементы
var itemsRef = items;

// Присвоить переменной items эквивалент нового объекта
items = [ 'new', 'array' ];

// Теперь переменные items и itemsRef указывают на разные объекты.
// В частности, переменная items указывает на массив [ 'new', 'array' ],
// а переменная itemsRef — на массив [ 'one', 'two', 'three' ]
console.log( items !== itemsRef );
```

И наконец, рассмотрим необычный пример, который, казалось бы, включает в себя ссылки, но на самом деле они в нем отсутствуют. В результате сцепления символьных строк всегда получается новый строковый объект, а не модифицированный вариант первоначальной символьной строки. Символьные строки, содержащие числовые и логические значения, относятся к примитивным типам, и поэтому они фактически не являются объектами ссылки, а содержащие их переменные — ссылками, как демонстрируется в листинге 2.4.

Листинг 2.4. Пример модификации объекта, приводящий к появлению нового объекта, но не к самомодифицикации существующего объекта

```
// Присвоить переменной item эквивалент нового строкового объекта
var item = 'test';

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

// Сцепить новый строковый объект с уже имеющимся.
// ПРИМЕЧАНИЕ: при этом создается новый объект, а не
// модифицируется исходный объект
item += 'ing';

// Значения переменных item и itemRef НЕ равны, поскольку
// создан совершенно новый объект
console.log( item != itemRef );
```

Особые недоразумения нередко вызывают символьные строки, поскольку они действуют как объекты. Чтобы создать экземпляр символьной строки, достаточно сделать вызов `new String()`. У символьных строк имеются свойства вроде `length`, а также методы наподобие `indexOf()` и `toUpperCase()`. Но при взаимодействии с переменными или функциями символьные строки ведут себя во многом как примитивные типы данных.

Понимание механизма действия ссылок может вызвать немалые трудности, когда вы только начинаете пользоваться ими. Тем не менее такое понимание имеет решающее значение для профессионального программирования на JavaScript. В последующих разделах мы продолжим рассмотрение языковых средств, которые не являются новыми или привлекательными, но все-таки очень важными для написания качественного, чистого кода на JavaScript.

Область действия

Область действия является непростым языковым средством JavaScript. В той или иной форме область действия присутствует в большинстве языков программирования, где она отличается лишь своей протяженностью. В языке JavaScript имеются только две области действия: функциональная и глобальная. Но такая простота обманчива. Так, у функций имеется своя область действия, тогда как у блоков (вроде операторов `while`, `if` и `for`) она отсутствует. Это может показаться необычным тем, кто раньше программировал на языке, где у блоков кода имеется своя область действия. Пример последствий, к которым приводит ограничение кода областью действия функций, приведен в листинге 2.5.

Листинг 2.5. Пример, демонстрирующий область действия переменных в JavaScript

```
// Присвоить глобальной переменной foo символьную строку 'test'
var foo = 'test';

// В блоке условного оператора if
if ( true ) {
  // присвоить переменной foo символьную строку 'new test'
```

```
  // ПРИМЕЧАНИЕ: эта переменная по-прежнему относится к
  // глобальной области действия!
  var foo = 'new test';
}

// Как видите, переменная foo теперь содержит
// символьную строку 'new test'
console.log( foo === 'new test' );

// Создать функцию, модифицирующую переменную foo
function test() {
  var foo = 'old test';
}

// Но когда эта функция вызывается, переменная foo остается
// в области ее действия
test();

// Об этом свидетельствует тот факт, что переменная foo
// по-прежнему содержит символьную строку 'new test'
console.log( foo === 'new test' );
```

Как следует из листинга 2.5, переменные находятся в глобальной области действия. Все переменные с глобальной областью действия фактически доступны в качестве свойств оконного объекта браузерного приложения JavaScript. А в других средах присутствует глобальный контекст, к которому относятся переменные с глобальной областью действия.

В примере кода из листинга 2.6 значение, присваиваемое переменной `foo`, находится в области действия функции `test()`. Но нигде в коде из листинга 2.6 область действия этой переменной не объявляется с помощью оператора `var foo`. Если область действия переменной `foo` не указана явно, то такая переменная становится определяемой глобально, даже если она предназначена для применения только в контексте функции.

Листинг 2.6. Пример объявления переменной с неявно определяемой глобальной областью действия

```
// Функция, в которой переменной foo присваивается значение
function test() {
  foo = 'test';
}

// Вызвать функцию, чтобы присвоить значение переменной foo
test();

// Как видите, переменная foo находится теперь
// в глобальной области действия
console.log( window.foo === 'test' );
```

Соблюдение области действия в JavaScript нередко вызывает недоразумение. Такое недоразумение у тех, кто раньше программировал на языке, где у блоков кода имеется своя область действия, может привести к случайному появлению глобаль-

ных переменных, как показано в листинге 2.6. Подобное недоразумение нередко усложняется неточностью употребления ключевого слова `var`. Поэтому ради простоты профессионально программирующие на JavaScript должны всегда инициализировать переменные с помощью ключевого слова `var` независимо от их области действия. Благодаря этому переменные будут иметь предполагаемую область действия, а случайного появления глобальных переменных удастся избежать.

Объявляя переменные в теле функции, мы отдаем себе отчет в существовании вопроса поднятия переменных. У любой переменной, объявленной в теле функции, имеется свое объявление, а не значение, которым она инициализируется, и это объявление поднимается вверх области действия. Этим в языке JavaScript обеспечивается доступность имени переменной во всей области действия. Язык JavaScript особенно проявляет свою эффективность для написания сценариев, когда область действия сочетается с понятиями контекста и замыканий, обсуждаемых в двух последующих разделах.

Контекст

Прикладной код всегда имеет некоторую форму контекста (область действия, в которой выполняется прикладной код). Контекст может быть эффективным инструментальным средством, необходимым для написания объектно-ориентированного кода. Это типичное средство в других языках, но в JavaScript, как это нередко бывает, оно трактуется несколько иначе.

Доступ к контексту осуществляется через переменную `this`, которая всегда ссылается на контекст, в котором выполняется прикладной код. Напомним, что глобальные объекты фактически являются свойствами оконного объекта. Это означает, что даже в глобальном контексте ссылка по-прежнему делается на объект. В листинге 2.7 демонстрируются простые примеры обращения с контекстом.

Листинг 2.7. Примеры применения функций в контексте и последующего переключения контекста на другую переменную

```
function setFoo(fooInput) {
  this.foo = fooInput;
}

var foo = 5;
console.log( 'foo at the window level is set to: ' + foo );

var obj = {
  foo : 10
};

console.log( 'foo inside of obj is set to: ' + obj.foo );

// Это приведет к изменению переменной foo на уровне окна
setFoo( 15 );
console.log( 'foo at the window level is now set to: ' + foo );

// Это приведет к изменению переменной foo на уровне объекта
obj.setFoo = setFoo;
```

```
obj.setFoo( 20 );
console.log( 'foo inside of obj is now set to: ' + obj.foo );
```

В листинге 2.7 функция `setFoo()` выглядит несколько странно. Как правило, ссылка `this` не применяется в теле обобщенной служебной функции. Зная, что в конечном итоге придется присоединить ссылку `setFoo` к ссылке `obj`, мы воспользовались ссылкой `this` для доступа к контексту переменной `obj`. Но такой способ не является строго обязательным. В языке JavaScript имеются два метода, которые позволяют выполнять функцию в произвольно указанном контексте. В листинге 2.8 демонстрируется, как этого добиться с помощью методов `call()` и `apply()`.

Листинг 2.8. Примеры смены контекста функций

```
// Простая функция, устанавливающая стиль цвета для своего контекста
function changeColor( color ) {
  this.style.color = color;
}
// Вызвать эту функцию для оконного объекта не удается,
// поскольку у него отсутствует объект стиля
changeColor('white' );

// Создать новый элемент разметки div, у которого будет объект стиля
var main = document.createElement('div');

// Задать для него черный цвет, используя метод call().
// Метод call() задает контекст с помощью первого аргумента,
// а все остальные аргументы передает вызываемой функции
changeColor.call( main, 'black' );

// Проверить результаты, используя функцию console.log().
// В итоге должна быть выведена символьная строка 'black'
console.log(main.style.color);

// Функция, задающая цвет в элементе разметки body
function setBodyColor() {
  // Метод apply() задает контекст для элемента разметки body
  // с помощью первого аргумента, а в качестве второго аргумента
  // он получает массив аргументов, который затем передается
  // вызываемой функции
  changeColor.apply( document.body, arguments );
}

// Задать черный цвет фона для тела веб-страницы
setBodyColor('black' );
```

Несмотря на всю полезность контекста, он может быть очевидным не сразу. Это станет яснее, когда мы рассмотрим особенности объектной ориентации.

Замыкания

Замыкания — это средства, с помощью которых внутренняя функция может ссылаться на переменные, присутствующие во внешней объемлющей функции после того, как ее родительские функции уже прекратили свое существование. Но в любом случае это лишь формальное определение. Вероятно, замыкания удобнее рассматривать в связи с контекстами. До сих пор, когда мы определяли литерал объекта, этот объект был открыт для модификации. И, как было показано, мы можем в любой момент вводить свойства и функции в объект. Но что, если нам потребуется контекст, который заблокирован, т.е. контекст, в котором значения сохранены как по умолчанию? Как получить контекст, который недоступен без предоставления прикладного программного интерфейса API? Доступ к контексту только выбранным способом как раз и обеспечивает замыкание.

Тема замыканий весьма обширна и сложна. Поэтому настоятельно рекомендуется обратиться за дополнительными сведениями о замыканиях к ресурсам, перечисленным в конце этого раздела. Итак, начнем рассмотрение замыканий с двух простых примеров, приведенных в листинге 2.9.

Листинг 2.9. Два примера, показывающих, каким образом замыкания могут сделать более ясным прикладной код

```
// Найти элемент разметки с идентификатором 'main'
var obj = document.getElementById('main');

// Изменить стилевое оформление его границ
obj.style.border = '1px solid red';

// Инициализировать обратный вызов через одну секунду
setTimeout(function(){
  // В итоге объект будет скрыт
  obj.style.display = 'none';
}, 1000);

// Обобщенная функция для отображения задержанного
// предупреждающего сообщения
function delayedAlert( msg, time ) {
  // Инициализировать охватываемый обратный вызов
  setTimeout(function(){
    // Использовать сообщение msg, переданное из объемлющей функции
    console.log( msg );
  }, time );
}

// Вызвать функцию delayedAlert() с двумя аргументами
delayedAlert('Welcome!', 2000 );
```

При первом вызове функции `setTimeout()` демонстрируется пример тех трудностей, с которыми приходится сталкиваться начинающим программировать на JavaScript, когда они имеют дело с замыканиями. Нередко в программах начинающих разработчиков можно встретить следующую строку кода:

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

или такую:

```
setTimeout('otherFunction(' + num + ',' + num2 + ')', 1000);
```

В обоих приведенных выше примерах вызываемые функции выражаются в виде символьных строк. Это может вызвать трудности в процессе минимизации, когда прикладной код подготавливается к передаче в эксплуатацию. Используя замыкания, можно вызывать функции, обращаться к переменным и передавать параметры, как и предполагалось первоначально.

Используя понятие замыканий, вполне возможно обойти все эти затруднения в прикладном коде. Первый пример из листинга 2.9 довольно прост. В нем обратный вызов функции `setTimeout()` делается через 1000 миллисекунд после ее первого вызова, но в ней по-прежнему делается ссылка на переменную `obj`, которая определяется глобально как элемент разметки с идентификатором `'main'`. А во втором примере определяется функция `delayedAlert()`, демонстрирующая разрешение затруднения, связанного с функцией `setTimeout()`, наряду с возможностью иметь замыкания в области действия функции. Применяя столь простые замыкания в своем коде, вы обнаружите, насколько повышается его ясность по сравнению с синтаксическим беспорядком в нем.

А теперь рассмотрим любопытный побочный эффект, который могут дать замыкания. В некоторых языках функционального программирования употребляется понятие *карринга* — предварительного заполнения ряда аргументов функции с помощью вновь создаваемой более простой функции. В листинге 2.10 демонстрируется простой пример карринга, когда создается новая функция, предварительно заполняющая аргумент другой функции.

Листинг 2.10. Пример карринга функций с помощью замыканий

```
// Функция, формирующая новую функцию для сложения чисел
function addGenerator( num ) {
  // Возвратить простую функцию для сложения чисел,
  // где первое число заимствуется из формирующей функции
  return function( toAdd ) {
    return num + toAdd
  };
}

// Теперь переменная addFive содержит функцию, принимающую
// один аргумент, складывающую с ним число 5 и возвращающую
// суммарное число
var addFive = addGenerator( 5 );

// Как видите, если передать функции из переменной addFive
// аргумент 4, то в итоге получится число 9
console.log( addFive( 4 ) == 9 );
```

В программировании на JavaScript имеется еще одно типичное затруднение, которое могут разрешить замыкания. Начинающие программировать на JavaScript зачастую неумышленно оставляют немало лишних переменных в глобальной области действия. И обычно это считается неудачной практикой программирования,

поскольку лишние переменные могут негласно вступать в конфликт с другими библиотеками, приводя к недоразумениям. Используя самоисполняющуюся анонимную функцию, можно, по существу, скрыть от другого кода все переменные, которые обычно оказываются глобальными (листинг 2.11).

Листинг 2.11. Пример применения анонимной функции для сокрытия переменных от глобальной области действия

```
// Создать новую анонимную функцию, чтобы использовать
// ее в качестве оболочки
(function(){
  // Переменная, которая обычно оказывается глобальной
  var msg = 'Thanks for visiting! ';

  // Привязать новую функцию к глобальному объекту
  window.onload = function(){
    // В этой функции используется "скрытая" переменная
   console.log( msg );
};

// Замкнуть анонимную функцию и выполнить ее
})();
```

И наконец, рассмотрим еще одно затруднение, связанное с замыканиями. Напомним, что замыкание позволяет ссылаться на переменные, существующие в родительской функции. Но оно предоставляет не исходное значение этой переменной в момент ее создания, а последнее ее значение в родительской функции. Чаще всего это явление можно наблюдать при выполнении цикла `for`, где в качестве итератора служит одна переменная `i`. Новые функции, создаваемые в цикле `for`, используют замыкание для повторной ссылки на итератор. Но дело в том, что к тому моменту, когда вызываются новые замыкаемые функции, они ссылаются на последнее значение итератора (т.е. на последнюю позицию в массиве), а не на предполагаемое значение. В листинге 2.12 демонстрируется применение анонимных функций для выведения области действия в качестве примера, где возможно предполагаемое замыкание.

Листинг 2.12. Пример применения анонимных функций для выведения области действия, необходимой для создания нескольких функций, использующих замыкание

```
// Элемент разметки с идентификатором 'main'
var obj = document.getElementById('main');

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

// Перебрать все элементы массива
for ( var i = 0; i < items.length; i++ ) {
  // Использовать самоисполняющуюся анонимную функцию
  // для выведения области действия
  (function(){
    // Запомнить значение в этой области действия.
```

```
    // Каждая переменная item особенная.
    // Не полагаться на переменные, создаваемые
    // в родительском контексте.
    var item = items[i];
    // Привязать функцию к элементу
    obj['on' + item ] = function() {
      // Переменная item ссылается на родительскую переменную,
      // успешно ограниченную пределами контекста данного цикла for
      console.log('Thanks for your ' + item );
    };
  })();
}
```

Мы еще вернемся к замыканиям в разделе, посвященном объектно-ориентированному коду, где будет показано, каким образом с их помощью реализуются закрытые свойства. Усвоить понятие замыканий непросто. Нам потребовалось немало времени и труда, чтобы по-настоящему понять и раскрыть истинный потенциал замыканий. Правда, имеется немало превосходных ресурсов, поясняющих принцип действия замыканий в JavaScript. К их числу относится статья "Замыкания в JavaScript" (JavaScript Closures) Ричарда Корнфорда (Richard Cornford), доступная по адресу `http://jibbering.com/faq/faq_notes/closures.html`, а также пояснительная документация на веб-сайте Mozilla Developer Network по адресу https://developer.mozilla.org/ru/docs/Web/JavaScript/Closures.

Перегрузка функций и проверка соответствия типов

Типичным средством в других объектно-ориентированных языках является возможность *перегрузки* функций для реализации разных видов поведения в зависимости от типа или количества передаваемых им аргументов. И хотя эта возможность не является языковым средством JavaScript, мы можем все же воспользоваться уже имеющимися возможностями для реализации перегрузки функций.

Для перегрузки функций в JavaScript нужно знать следующее: сколько аргументов им передано и каковы их типы. Начнем с рассмотрения количества передаваемых аргументов.

В теле каждой функции, определяемой в JavaScript, имеется контекстуальная переменная `arguments`, действующая как объект типа массива, содержащий все аргументы, передаваемые функции. Объект `arguments` не является подлинным массивом, поскольку он не разделяет прототип с объектом типа `Array` и не обладает функциями обработки массивов вроде `push()` или `indexOf()`. Тем не менее он обладает позиционным доступом к массиву (например, по ссылке `arguments[2]` возвращается третий аргумент) и свойством `length`. Два характерных тому примера приведены в листинге 2.13.

Листинг 2.13. Два примера перегрузки функций в JavaScript

```
// Простая функция для отправки сообщения
function sendMessage( msg, obj ) {
  // Если предоставляется как сообщение, так и объект,
  if ( arguments.length === 2 ) {
```

```
    // Отправить сообщение объекту
    // (Предполагается, что объект obj имеет свойство log!)
    obj.log( msg );
  } else {
    // В противном случае допустить, что было передано
    // только сообщение. Следовательно, вывести только
    // сообщение об ошибке по умолчанию
    console.log( msg );
  }
}

// Оба эти вызова функций вполне работоспособны
sendMessage( 'Hello, World!' );
sendMessage( 'How are you?', console );
```

В связи с изложенным выше возникает вопрос: может ли объект `arguments` действовать как полноценный массив? Для объекта `arguments` это невозможно, но в то же время имеется возможность создать его копию в виде массива. Вызвав метод `slice()` из прототипа объекта `Array`, можно быстро скопировать объект `arguments` в массив, как демонстрируется в листинге 2.14.

Листинг 2.14. Преобразование аргументов в массив

```
function aFunction(x, y, z) {
  var argsArray = Array.prototype.slice.call( arguments, 0 );
  console.log( 'The last argument is: ' + argsArray.pop() );
}

// Вывести сообщение 'The last argument is 3'
aFunction( 1, 2, 3 );
```

Более подробно свойство `prototype` будет рассмотрено несколько ниже. А до тех пор достаточно сказать, что прототип обеспечивает доступ к методам объектов в статическом режиме.

Но что, если сообщение вообще не определено? Мы должны иметь возможность проверять не только наличие, но и отсутствие аргумента в вызове функции. Мы можем воспользоваться тем фактом, что любой не предоставленный аргумент имеет неопределенное значение. В листинге 2.15 демонстрируется простая функция, выводящая сообщение об ошибке или сообщение по умолчанию, если конкретный аргумент не предоставляется. (Обратите внимание на то, что в данном случае мы вынуждены воспользоваться оператором `typeof`, поскольку иначе аргумент с литеральной строкой `"undefined"` (не определено) будет указывать на ошибку.)

Листинг 2.15. Вывод сообщения об ошибке или сообщения по умолчанию

```
function displayError( msg ) {
  // Проверить и убедиться, что аргумент msg неопределен
  if ( typeof msg === 'undefined' ) {
    // Если аргумент msg неопределен,
    // задать сообщение по умолчанию
    msg = 'An error occurred.';
```

```
  }
  // Вывести сообщение
  console.log( msg );
}

displayError();
```

Применение оператора `typeof` помогает нам плавно перейти к теме проверки соответствия типов. И эту тему весьма полезно и важно рассмотреть, поскольку язык JavaScript является динамически типизированным. Имеется немало способов для проверки соответствия типа переменной, но здесь будут рассмотрены лишь два наиболее полезных способа.

Первый способ проверки соответствия типа объекта состоит в применении оператора `typeof`, название которого говорит само за себя. Этот оператор предоставляет строковое имя, обозначающее тип содержимого переменной. Характерный пример применения этого способа демонстрируется в листинге 2.16.

Листинг 2.16. Пример применения оператора `typeof` для определения типа объекта

```
var num = '50';
var arr = 'apples,oranges,pears';

// Проверить, представлено ли число символьной строкой
if ( typeof num === 'string' ) {
  // Если это так, выполнить синтаксический анализ строки,
  // чтобы извлечь из нее число
  num = parseInt( num );
}

// Проверить, представлен ли массив символьной строкой
if ( typeof arr == 'string' ) {
  // Если это так, создать массив, разделив его элементы запятой
  arr = arr.split( ',' );
}
```

Преимущество оператора `typeof` заключается в том, что для проверки содержимого переменной совсем не обязательно знать его конкретный тип. Такое решение было бы идеальным для всех переменных, кроме типа `Object`, `Array` или специального объекта типа `User`, поскольку в этом случае оператор `typeof` возвращает символьную строку `"object"`, что затрудняет различение конкретных типов объектов. Другой способ требует проверять содержимое переменной относительно конкретного существующего типа.

Второй способ проверки соответствия типа объекта состоит в применении оператора `instanceof`. Этот оператор проверяет свой левый операнд относительно конструктора в правом операнде, что может показаться более сложным, чем есть на самом деле! Характерный пример применения этого способа демонстрируется в листинге 2.17. А в следующей главе, посвященной объектно-ориентированным свойствам JavaScript, мы рассмотрим функцию `Object.isPrototypeOf()`, которая также помогает определить тип объекта.

Листинг 2.17. Пример применения оператора `instanceof` для определения типа объекта

```
var today = new Date();
var re = /[a-z]+/i;

// Приведенные ниже строки кода не дают достаточных сведений
// о типе проверяемых переменных
console.log('typeof today: ' + typeof today);
console.log('typeof re: ' + typeof re);

// Выяснить, относятся ли переменные к более конкретному типу
if (today instanceof Date) {
  console.log('today is an instance of a Date.');
}

if (re instanceof RegExp) {
  console.log( 're is an instance of a RegExp object.' );
}
```

В основу проверки соответствия типа переменных и длины массивов аргументов положены простые принципы, но они позволяют изобрести сложные способы, которые можно приспособить и сделать более удобными как для разработчиков, так и для пользователей прикладного кода. Если требуется проверить объект на соответствие конкретному типу (является ли он массивом, датой или объектом специального типа), рекомендуется создать специальную функцию для определения типа проверяемого объекта. Во многих библиотеках имеются служебные функции для определения массивов, дат и прочих типов объектов. Инкапсулируя этот код в теле функции, можно обеспечить единственное место для проверки данного конкретного типа вместо того, чтобы проверять код, разбросанный по всей кодовой базе.

Новые инструментальные средства для управления объектами

Одним из привлекательных достижений в языке JavaScript стало расширение набора инструментальных средств для управления объектами. Как будет показано в этом разделе, подобные инструментальные средства можно применять к литералам объектов, похожим на структуры данных, а также к экземплярам объектов.

Объекты

Объекты служат основой для языка JavaScript. Буквально все в этом языке является объектом. И этому факту обязана большая часть истинного потенциала данного языка. На самом элементарном уровне объекты существуют в виде совокупности свойств аналогично хеш-конструкциям в других языках программирования. Два характерных примера создания объекта с рядом свойств демонстрируются в листинге 2.18.

Листинг 2.18. Два примера создания простого объекта и установки его свойств

```
// Создать новый объект типа Object и сохранить его в переменной obj
var obj = new Object();

// Установить разные значения в некоторых свойствах объекта
obj.val = 5;
obj.click = function(){
  console.log('hello');
};

// Ниже приведен эквивалент кода, в котором для определения
// свойств применяются фигурные скобки {...} наряду с парами
// "ключ-значение"
var obj = {

  // Установить имена и значения свойств,
  // используя пары "ключ-значение"
  val: 5,
  click: function(){
    console.log('hello');
  }
};
```

В действительности возможности объектов не намного шире представленных выше. Но дело усложняется, когда создаются новые объекты, наследующие свойства других объектов.

Модификация объектов

В языке JavaScript теперь имеются три метода, способные оказать помощь в управлении модификацией объектов. Мы рассмотрим их по шкале от наименьших до наибольших ограничений.

По умолчанию объект в JavaScript может быть модифицирован в любой момент. Используя метод `Object.preventExtensions()`, можно предотвратить ввод новых свойств в объект. Когда это происходит, могут быть использованы все текущие свойства, но ни одно из новых свойств ввести не удастся. Попытка ввести новое свойство приведет к ошибке `TypeError`, а иначе неудачный исход операции окажется негласным. Обнаружить подобную ошибку, вероятнее всего, удастся в строгом режиме. Характерный тому пример приведен в листинге 2.19.

Листинг 2.19. Пример применения метода `Object.preventExtensions()`

```
// Создать новый объект типа Object и сохранить его в переменной obj
var obj = {};

// Создать новый объект типа Object, используя метод preventExtensions()
var obj2 = Object.preventExtensions(obj);

// Выдать ошибку TypeError при попытке определить новое свойство
function makeTypeError(){
```

```
  'use strict';

  // Выдать ошибку TypeError при попытке определить новое свойство
  Object.defineProperty(obj2, 'greeting',{value: 'Hello World'});
}

makeTypeError();
```

Используя метод `Object.seal()`, можно ограничить возможности объекта аналогично тому, как это делается с помощью метода `Object.preventExtensions()`. Но в отличие от предыдущего примера, в данном случае свойства нельзя удалять или преобразовывать в методы доступа (получения). Попытка удалить или ввести свойства также приведет к ошибке `TypeError`. Характерный тому пример приведен в листинге 2.20.

Листинг 2.20. Пример применения метода `Object.seal()`

```
// Создать новый объект и воспользоваться методом Object.seal(),
// чтобы ограничить его возможности
var obj = {};
obj.greeting = 'Welcome';
Object.seal(obj);

// Обновить существующее записываемое свойство.
// Преобразовать существующее свойство в метод доступа нельзя,
// выдается ошибка TypeError
obj.greeting = 'Hello World';
Object.defineProperty(
    obj, 'greeting', {get:function(){return 'Hello World'; } });

// Удалить существующее свойство нельзя, неудачный исход
// операции оказывается негласным
delete obj.greeting;

function makeTypeError(){
  'use strict';

  // Выдается ошибка TypeError при попытке удалить свойство
  delete obj.greeting;

  // Свойство можно все же обновить
  obj.greeting = 'Welcome';
  console.log(obj.greeting);
}

makeTypeError();
```

Как показывает пример, демонстрируемый в листинге 2.21, метод `Object.freeze()` является самым ограничивающим из трех рассматриваемых здесь методов. После применения этого метода объект считается неизменяемым. Это означает, что в данном объекте нельзя вводить, удалять или обновлять свойства. Если же

свойство само является объектом, то оно может быть обновлено. Это так называемое *неполное замораживание*. Чтобы сделать объект полностью неизменяемым, следует заморозить все его свойства, значения которых являются объектами.

Листинг 2.21. Пример применения метода `Object.freeze()`

```
// Создать новый объект с двумя свойствами.
// Второе свойство является объектом
var obj = {
  greeting: "Welcome",
  innerObj: {}
};

// Заморозить содержимое переменной obj
Object.freeze(obj);

// Неудачный исход операции оказывается негласным
obj.greeting = 'Hello World';

// Свойство innerObj может быть по-прежнему обновлено
obj.innerObj.greeting = 'Hello World';
console.log('obj.innerObj.greeting = ' + obj.innerObj.greeting);

// Преобразовать существующее свойство в метод доступа нельзя,
// выдается ошибка TypeError
Object.defineProperty(
    obj, 'greeting', {get:function(){return 'Hello World'; } });

// Удалить существующее свойство нельзя, неудачный исход
// операции оказывается негласным
delete obj.greeting;

function makeTypeError(){
  'use strict';
}

// Выдается ошибка TypeError при попытке удалить свойство
delete obj.greeting;

// Заморозить внутренний объект
Object.freeze(obj.innerObj);

// Свойство innerObj теперь заморожено. Неудачный исход
// операции оказывается негласным
obj.innerObj.greeting = 'Worked so far...';

function makeTypeError(){
  'use strict';
```

```
  // Выдается ошибка TypeError при любой попытке
  // каким-то образом изменить свойство
  delete obj.greeting;
  obj.innerObj.greeting = 'Worked so far...';
  obj.greeting = "Welcome";
};

makeTypeError();
```

Ясно понимая, каким образом можно управлять изменчивостью объекта, можно согласовать обращение с ним на определенном уровне. Так, если имеется объект `User`, то на его основе можно создать любой объект с полной уверенностью, что новый объект будет иметь те же самые свойства, что и у исходного объекта. А ввод любых свойств во время выполнения может привести к неудачному исходу.

Резюме

Важность ясного понимания языковых понятий, рассмотренных в этой главе, трудно переоценить. В первой половине этой главы подробно разъяснялось, как ведет себя язык JavaScript и как им лучше всего пользоваться. Это должно послужить отправной точкой для уяснения профессионального подхода к программированию на JavaScript. Ясное понимание того, как обращаться с объектами, ссылками и областью действия, безусловно, может коренным образом изменить подход к написанию кода на JavaScript.

На подобных навыках основываются передовые методики, предоставляющие дополнительные способы разрешения затруднений при программировании на Java Script. Уяснение особенностей области действия и контекста привело к применению замыканий, а тщательный анализ определения типов позволил внедрить перегрузку функций в язык, для которого данное средство не было характерным. Далее в этой главе был рассмотрен тип `Object` — один из самых основных функциональных типов в JavaScript. Различные новые средства данного типа позволяют намного лучше управлять создаваемыми литералами объектов. И это позволяет естественным образом перейти к следующей главе, с которой начинается рассмотрение объектно-ориентированных свойств языка JavaScript.

ГЛАВА 3

Создание повторно используемого кода

В начале предыдущей главы мы обсуждали объекты как основополагающие единицы языка JavaScript. Рассмотрев литералы объектов в JavaScript, мы посвятим большую часть этой главы тому, как эти объекты согласуются с принципами объектно-ориентированного программирования. В этом отношении язык JavaScript находится в состоянии напряжения между классическим программированием и собственными уникальными возможностями JavaScript.

Перейдя от организации прикладного кода к объектам, мы рассмотрим другие образцы управления прикладным кодом. Мы должны гарантировать от засорения глобального пространства имен или чрезмерной опоры на глобальные переменные. Таким образом, обсуждение в этой главе начнется с пространств имен, хотя они — лишь вершина айсберга, а также с новых шаблонов вызова функций, помогающих надлежащим образом оградить прикладной код: сначала модулей, а затем немедленно вызываемых функциональных выражений (IIFE).

Как только прикладной код будет организован в отдельном файле, целесообразно рассмотреть инструментальные средства для управления многими файлами JavaScript. Безусловно, мы можем опираться на сети доставки содержимого для некоторых библиотек, которыми мы могли бы воспользоваться. Но мы должны также подумать о том, как лучше всего загрузить свои файлы JavaScript, чтобы в конечном итоге у нас не получились HTML-файлы, которые содержат следующие друг за другом дескрипторы сценариев.

Объектно-ориентированные свойства JavaScript

Язык JavaScript является прототипным, а не классическим языком программирования. Разберемся, прежде всего, в этой его особенности. Язык Java является классическим языком программирования, поскольку все в нем требует наличия класса. А в языке JavaScript все имеет свой прототип, и поэтому он является прототипным языком программирования. Но в этом прототипном характере JavaScript кроется противоречие, как считает Дуглас Крокфорд (Douglas Crockford) и другие.

Подобно герою поневоле, JavaScript иногда не желает выделяться из общего ряда языков программирования, чтобы выставить свои достоинства в выгодном свете. Что ж, дадим ему плащ и посмотрим, что из этого выйдет!

Итак, напомним еще раз, что JavaScript не является классическим языком программирования. Имеется немало литературы, интернет-ресурсов, руководств, презентаций и библиотек, где предпринимается попытка наложить на JavaScript языковые структуры, основанные на классах. Можете углубленно изучить эти первоисточники сами, но имейте в виду, что их авторы пытаются сделать невозможное: вбить квадратный колышек в круглое отверстие. Мы не будем здесь этого делать. В этой главе не обсуждается, как заставить JavaScript действовать подобно Java. Вместо этого уделим основное внимание тому, насколько возможности JavaScript соответствуют принципам объектно-ориентированного программирования (ООП), как в одних случаях этих возможностей явно не хватает, а в других они имеются с избытком.

В конечном счете зачем нам вообще ООП? Оно предоставляет образцы применения, позволяющие упростить повторное использование кода, исключив его дублирование. Кроме того, ООП помогает тщательнее продумывать код, с которым мы работаем, и предоставляет общий план для успешной реализации. Но это не только общий план. Ведь прототипы в JavaScript дают возможность достичь той же самой цели, но другим путем.

Начнем с самого прототипа. У всякого типа данных (`Object`, `Function`, `Date` и т.д.) в JavaScript имеется свой прототип. В стандарте ECMAScript он называется скрытым свойством объекта и обозначается как `[[Prototype]]`. До сих пор прототип был доступен двумя путями: как нестандартное свойство `__proto__` и как свойство `prototype`. Сначала раскрытие свойства `__proto__` считалось ненадежным способом доступа в браузерах, но даже в этом случае подобный доступ не всегда осуществлялся одинаково, поскольку критические части кода JavaScript реализовывались в браузерах по-разному! А с утверждением стандарта ECMAScript 6 свойство `__proto__` станет официальным для типов данных и будет доступно для любой реализации, соответствующей данному стандарту. Но это время пока еще не настало.

Имеется также возможность получить доступ к свойству `prototype` некоторых типов данных. У всех основных типов данных в JavaScript (`Date`, `String`, `Array` и т.д.) имеется открытое свойство `prototype`. И у всякого типа данных в JavaScript, созданного из конструктора функции, также имеется открытое свойство `prototype`. Но у экземпляров этих типов данных, будь то символьные строки, даты или что-нибудь другое, свойство `prototype` отсутствует. Дело в том, что свойство `prototype` недоступно для экземпляров. Мы не будем вообще пользоваться свойством `prototype`, поскольку в качестве конструкторов собираемся применять объекты, а не функции.

Таким образом, мы будем пользоваться литералом объекта в качестве основания для других объектов. Если это кажется сильно похожим на классы и экземпляры, то, несмотря на некоторые сходства, у такого подхода имеются, как и следовало ожидать, свои отличия. В качестве примера рассмотрим объект `Person`, приведенный в листинге 3.1.

Листинг 3.1. Объект `Person`

```
var Person = {
  firstName : 'John',
  lastName : 'Connolly',
```

```
  birthDate : new Date('1964-09-05'),
  gender: 'male',
  getAge : function() {
    var today = new Date();
    var diff = today.getTime() - this.birthDate.getTime();
    var year = 1000 * 60 * 60 * 24 * 365.25;
    return Math.floor(diff / year);
  }
};
```

В этом объекте нет ничего примечательного: у отдельного лица имеется имя, фамилия, пол, дата рождения и способ расчета его возраста. `Person`, по существу, является литералом объекта, а не тем, что можно было бы признать как класс. Но нам требуется использовать объект `Person` так, как если бы это был класс. Нам требуется создать дополнительные объекты, соответствующие представленной выше структуре данного объекта `Person`. Чтобы сохранить отличие бесклассового языка JavaScript от объектно-ориентированных языков программирования, имеющих классы, мы будем обращаться к объекту `Person` как к типу данных аналогично `Date`, `Array`, `RegExp` и всем другим подобным типам. Нам требуется создать экземпляры типа `Person`, и для этого мы можем воспользоваться методом `Object.create()`, как показано в листинге 3.2.

Листинг 3.2. Создание экземпляров типа `Person`

```
var Person = {
  firstName : 'John',
  lastName : 'Connolly',
  birthDate : new Date( '1964-09-05' ),
  gender : 'male',
  getAge : function () {
    var today = new Date();
    var diff = today.getTime() - this.birthDate.getTime();
    var year = 1000 * 60 * 60 * 24 * 365.25;
    return Math.floor( diff / year );
  },

  toString : function () {
    return this.firstName + ' ' + this.lastName +
           ' is a ' + this.getAge() + ' year-old ' + this.gender;
  }
};

var bob = Object.create( Person );
bob.firstName = 'Bob';
bob.lastName = 'Sabatelli';
bob.birthDate = new Date( '1969-06-07' );
console.log( bob.toString() );
```

Экземпляр был получен из объекта `Person`. Этот экземпляр сохраняется в переменной `bob` без всяких классов. Но в то же время имеется связь между созданными экземплярами объекта `Person` и типом `Person`, и эта связь поддерживается через

свойство `[[Prototype]]`. Если вы пользуетесь относительно новым браузером (на момент написания данной книги это были версии браузеров Internet Explorer 11, Firefox 27 и Chrome 33), то можете открыть окно консоли из меню инструментов разработки и посмотреть на свойство `__proto__` в переменной `bob`. При этом вы обнаружите, что оно указывает на объект `Person`. В действительности этот факт можно проверить с помощью выражения `bob.__proto__ === Person`.

Метод `Object.create()` был внедрен в JavaScript по стандарту ECMAScript 5 под предлогом упрощения и прояснения взаимосвязи между объектами, особенно теми объектами, которые были связаны с их прототипами. Но благодаря этому удалось за один раз создать взаимосвязь между объектами. Эта взаимосвязь очень похожа на принцип ООП, определяющий взаимосвязь между классом и его экземпляром. Но поскольку в языке JavaScript отсутствуют классы, то мы просто получаем объекты с взаимосвязью между ними.

Такую взаимосвязь нередко называют *цепочкой прототипов*. В языке Java Script цепочка прототипов служит одним из двух мест, которые проверяются, чтобы разрешить значение члена объекта. Это означает, что когда делается ссылка `foo.bar` или `foo[bar]`, механизм JavaScript осуществляет поиск значения члена `bar` в следующих двух потенциальных местах: в самом объекте `foo` или в цепочке его прототипа.

В своей статьей, посвященной объектно-ориентированным свойствам JavaScript (`http://davidwalsh.name/javascript-objects`), Кайл Симпсон (Kyle Simpson) изящно поясняет, как следует рассматривать этот процесс. Вместо того чтобы рассматривать взаимосвязь между переменной `bob` и объектом `Person` как экземпляром класса или между потомком и родителем, мы должны трактовать ее как случай делегирования поведения. У объекта, хранящегося в переменной `bob`, имеются свои свойства `firstName` и `lastName`, но отсутствуют любые возможности функции `getAge()`. Это означает, что они делегируются объекту `Person`. Делегирующая взаимосвязь устанавливается с помощью метода `Object.create()`, а цепочка прототипа является механизмом такого делегирования, позволяя делегировать поведение далее по цепочке. С точки зрения переменной `bob` функциональные возможности накапливаются по мере успешности вызова метода `Object.create()`, накладываясь на дополнительные возможности.

Между прочим, у вас может возникнуть озабоченность по поводу того, что стандарт ECMAScript 5 не поддерживается в интересующем вас браузере или, по крайней мере, в нем отсутствует своя версия метода `Object.create()`. Это затруднение легко разрешить, прибегнув к полизаполнению метода `Object.create()` в любом браузере с помощью механизма JavaScript, как показано в листинге 3.3.

Листинг 3.3. Полизаполнение метода `Object.create()`

```
if ( typeof Object.create !== 'function' ) {
  Object.create = function ( o ) {
    function F() {
    }
    F.prototype = o;
    return new F();
  };
}
```

И наконец, некоторым не нравится идея постоянно пользоваться методом `Object.create()` для создания объектов, поскольку им удобнее пользоваться типичным выражением `someInstance = new Type()`. В таком случае им следует рассмотреть способ быстрой модификации объекта `Person`, демонстрируемый в листинге 3.4, где для формирования дополнительных объектов `Person` предоставляется фабричный метод.

Листинг 3.4. Объект `Person` с фабричным методом

```
var Person = {
  firstName : 'John',
  lastName : 'Connolly',
  birthDate : new Date( '1964-09-05' ),
  gender : 'male',
  getAge : function () {
    var today = new Date();
    var diff = today.getTime() - this.birthDate.getTime();
    var year = 1000 * 60 * 60 * 24 * 365.25;
    return Math.floor( diff / year );
  },

  toString : function () {
    return this.firstName + ' ' + this.lastName + ' is a ' +
           this.getAge() + ' year-old ' + this.gender;
  },

  extend : function ( config ) {
    var tmp = Object.create( this );
    for ( var key in config ) {
      if ( config.hasOwnProperty( key ) ) {
        tmp[key] = config[key];
      }
    }
    return tmp;
  }
};

var bob = Person.extend( {
  firstName : 'Bob',
  lastName : 'Sabatelli',
  birthDate : new Date( '1969-06-07' )
} );

console.log( bob.toString() );
```

В данном примере функция `extend()` инкапсулирует вызов метода `Object.create()`. Когда вызывается функция `extend()`, в ее теле вызывается метод `Object.create()`. По-видимому, функция `extend()` вызывается с переданным ей объектом конфигурации, что весьма характерно для JavaScript. Циклически сохраняя свойства в объекте `tmp`, функция `extend()` гарантирует также, что вновь создаваемый

объект `tmp` расширяется только теми свойствами объекта `config`, которые уже присутствуют в объекте `tmp`. Как только свойства будут скопированы из объекта `config` в объект `tmp`, последний можно возвратить в качестве экземпляра типа `Person`. Итак, рассмотрев новый способ установления взаимосвязи между объектами в JavaScript, обсудим, как это согласуется с типичными принципами ООП.

Наследование

Безусловно, самый большой вопрос вызывает наследование. Главным достоинством объектно-ориентированного кода является повторное использование функциональных возможностей, наследуемых из общих родительских классов в более конкретных порожденных классах. Как было показано ранее, взаимосвязь между двумя объектами нетрудно установить с помощью метода `Object.create()`. Такой способ можно применить и для создания любой требующейся иерархии наследования. (Точнее говоря, иерархии единичного наследования. Ведь метод `Object.create()` не допускает множественное наследование.) Напомним, что основная идея состоит в делегировании поведения. Когда подклассы создаются с помощью метода `Object.create()`, они делегируют часть своего поведения типам данных далее по цепочке прототипов. Наследование методом `Object.create()` в большей степени происходит снизу вверх, а не сверху вниз, как это характерно для ООП.

По существу, для наследования достаточно воспользоваться методом `Object.create()`. Точнее говоря, с помощью метода `Object.create()` устанавливается взаимосвязь между родительским и порожденным типом данных. В порожденном типе данных можно вводить новые функциональные возможности, а также удалять или переопределять уже имеющиеся. Если вызвать метод `Object.create()` с аргументом, обозначающим любой объект, выбранный в качестве родительского типа, то в конечном итоге будет возвращено значение, выбранное в качестве порожденного типа. Следуя далее примеру, приведенному в листинге 3.4, нужно вызвать метод `extend()` (или повторно воспользоваться методом `Object.create()`), чтобы создать экземпляры данного порожденного типа (листинг 3.5).

Листинг 3.5. Тип `Person` является родительским для типа `Teacher`

```
var Person = {
  firstName : 'John',
  lastName : 'Connolly',
  birthDate : new Date( '1964-09-05' ),
  gender : 'male',
  getAge : function () {
    var today = new Date();
    var diff = today.getTime() - this.birthDate.getTime();
    var year = 1000 * 60 * 60 * 24 * 365.25;
    return Math.floor( diff / year );
  },

  toString : function () {
    return this.firstName + ' ' + this.lastName + ' is a ' +
           this.getAge() + ' year-old ' + this.gender;
  },
```

```
  extend : function ( config ) {
    var tmp = Object.create( this );
    for ( var key in config ) {
      if ( config.hasOwnProperty( key ) ) {
        tmp[key] = config[key];
      }
    }
    return tmp;
  }
};

var Teacher = Person.extend( {
  job : 'teacher',
  subject : 'English Literature',
  yearsExp : 5,
  toString : function () {
    return this.firstName + ' ' + this.lastName + ' is a ' +
           this.getAge() + ' year-old ' + this.gender + ' ' +
           this.subject + ' teacher.';
  }
} );

var patty = Teacher.extend( {
  firstName : 'Patricia',
  lastName : 'Hannon',
  subject: 'chemistry',
  yearsExp : 20,
  gender : 'female'
} );

console.log( patty.toString() );
```

Метод `Object.create()` установил связь между прототипами `[[Prototype]]` обоих типов, `Teacher` и `Person`. Если вы пользуетесь одним из упоминавшихся ранее современных браузеров, то, просматривая свойство `__proto__` типа `Teacher`, можете обнаружить, что оно указывает на тип `Person`.

Как пояснялось в главе 2, оператор `instanceof` позволяет определить, является ли объект экземпляром типа данных. Но в данном случае этот оператор не годится, так как он опирается на явное свойство `prototype` для прослеживания взаимосвязи между объектом и типом данных. Проще говоря, правым операндом оператора `instanceof` должна быть функция, а скорее всего, — конструктор функции, тогда как левым его операндом — нечто, созданное из конструктора функции, хотя и необязательно из конструктора функции в правой части данного оператора. Так как же выяснить, является ли объект экземпляром типа данных? Для этого следует вызвать функцию `isPrototypeOf()`.

Функцию `isPrototypeOf()` можно вызвать для любого объекта. Она присутствует во всех объектах в JavaScript аналогично функции `toString()`. Достаточно вызвать ее для объекта, выполняющего роль типа данных (`Person` или `Teacher` в приведенных выше примерах), передав ей в качестве аргумента объект, выполняющий роль

экземпляра (bob или patty там же). Следовательно, в результате вызова `Teacher.isPrototypeOf(patty)` будет возвращено логическое значение `true`, как и следовало ожидать. В листинге 3.6 приведен пример кода, в котором объекты `Person` и `Teacher`, экземпляры `bob` и `patty` сочетаются с вызовами функции `isPrototypeOf()`.

Листинг 3.6. Применение функции `isPrototypeOf()`.

```
var Person = {
  firstName : 'John',
  lastName : 'Connolly',
  birthDate : new Date( '1964-09-05' ),
  gender : 'male',
  getAge : function () {
    var today = new Date();
    var diff = today.getTime() - this.birthDate.getTime();
    var year = 1000 * 60 * 60 * 24 * 365.25;
    return Math.floor( diff / year );
  },

  toString : function () {
    return this.firstName + ' ' + this.lastName + ' is a ' +
           this.getAge() + ' year-old ' + this.gender;
  },

  extend : function ( config ) {
    var tmp = Object.create( this );
    for ( var key in config ) {
      if ( config.hasOwnProperty( key ) ) {
        tmp[key] = config[key];
      }
    }
    return tmp;
  }
};

var Teacher = Person.extend( {
  job : 'teacher',
  subject : 'English Literature',
  yearsExp : 5,
  toString : function () {
    return this.firstName + ' ' + this.lastName + ' is a ' +
           this.getAge() + ' year-old ' + this.gender + ' ' +
           this.subject + ' teacher.';
  }
} );

var bob = Person.extend( {
  firstName : 'Bob',
  lastName : 'Sabatelli',
```

```
  birthDate : new Date( '1969-06-07' )
} );

var patty = Teacher.extend( {
  firstName : 'Patricia',
  lastName : 'Hannon',
  subject: 'chemistry',
  yearsExp : 20,
  gender : 'female'
} );

console.log( 'Is bob an instance of Person? ' +
             Person.isPrototypeOf(bob) ); // истинно
console.log( 'Is bob an instance of Teacher? ' +
             Teacher.isPrototypeOf( bob ) ); // ложно
console.log( 'Is patty an instance of Teacher? ' +
             Teacher.isPrototypeOf( patty ) ); // истинно
console.log( 'Is patty an instance of Person? ' +
             Person.isPrototypeOf( patty ) ); // истинно
```

Функции `isPrototypeOf()` сопутствует другая функция, называемая `getPrototypeOf()`. Так, в результате вызова `Object.getPrototypeOf(obj)` возвращается ссылка на тип данных, который послужил основанием для текущего объекта. Как упоминалось ранее, аналогичную информацию можно обнаружить, просмотрев свойство `__proto__` (листинг 3.7), хотя оно пока еще не стало стандартным.

Листинг 3.7. Применение функции `getPrototypeOf()`.

```
console.log( 'The prototype of bob is Person' +
             Object.getPrototypeOf( bob ) );
```

А как же получить доступ к переопределяемым методам? В порожденном объекте, конечно, можно переопределить метод из родительского объекта. И в такой возможности нет ничего особенного, поскольку она вполне ожидаема в любой объектно-ориентированной системе. Но в большинстве объектно-ориентированных систем переопределенный метод получает доступ к родительскому методу через свойство или метод доступа, вызываемый по ссылке вроде `super`. Это означает, что когда метод переопределяется, его можно, как правило, вызвать с помощью специального ключевого слова.

Но объектно-ориентированному коду JavaScript, основанному на прототипах, просто недоступна функция `super()`. Обычно из этого затруднительного положения можно выйти тремя способами. Во-первых, можно написать код для повторной реализации ссылки `super`. Чтобы найти в цепочке наследования объект, имевшийся в предыдущем варианте переопределяемого метода, придется вернуться обратно по цепочке прототипов — возможно, с помощью метода `getPrototypeOf()`. (Напомним, что переопределение методов всегда происходит в родительском типе данных; это может быть "прародительский" класс или нечто, находящее еще выше по цепочке прототипов.) В таком случае придется каким-то образом сделать доступным этот метод, вызвав его с тем же самым рядом аргументов, которые передаются методу, который переопределяется. И хотя это вполне возможно, но скверно и совершенно

не эффективно. Во-вторых, можно явным образом вызвать родительский метод, как демонстрируется в листинге 3.8.

Листинг 3.8. Воспроизведение действия функции `super()`.

```
var Person = {
  firstName : 'John',
  lastName : 'Connolly',
  birthDate : new Date( '1964-09-05' ),
  gender : 'male',
  getAge : function () {
    var today = new Date();
    var diff = today.getTime() - this.birthDate.getTime();
    var year = 1000 * 60 * 60 * 24 * 365.25;
    return Math.floor( diff / year );
  },

  toString : function () {
    return this.firstName + ' ' + this.lastName + ' is a ' +
           this.getAge() + ' year-old ' + this.gender;
  },

  extend : function ( config ) {
    var tmp = Object.create( this );
    for ( var key in config ) {
      if ( config.hasOwnProperty( key ) ) {
        tmp[key] = config[key];
      }
    }
    return tmp;
  }
};

var Teacher = Person.extend( {
  job : 'teacher',
  subject : 'English Literature',
  yearsExp : 5,
  toString : function () {
    var originalStr = Person.toString.call(this);
    return originalStr + ' ' + this.subject + ' teacher.';
  }
} );

var patty = Teacher.extend( {
  firstName : 'Patricia',
  lastName : 'Hannon',
  subject: 'chemistry',
  yearsExp : 20,
  gender : 'female'
```

```
} );

console.log( patty.toString() );
```

Обратите особое внимание на метод `toString()` в типе `Teacher`. Этот метод делает явным вызов метода `toString()` из типа `Person`. Многие разработчики объектно-ориентированного кода могут оспорить необходимость жесткого кодирования взаимосвязи между типами `Teacher` и `Parent`. Тем не менее это позволяет достичь конечной цели, быстро, изящно и эффективно разрешив данное затруднение, хотя такой способ делает код непереносимым и пригоден только для объектов, которые каким-то образом связаны с объектом `Parent`.

И в-третьих, можно вообще пренебречь языковым средством `super`, поскольку в JavaScript оно отсутствует, в отличие от многих других языков ООП. Но это языковое средство не самое важное в объектно-ориентированном коде. Возможно, в будущем ключевое слово `super` и появится в JavaScript с соответствующими функциональными возможностями. (В настоящее время известно, что в стандарте ECMAScript 6 предусмотрено свойство `super` для объектов.) А до тех пор можно вполне обойтись и без него.

Доступность членов

В объектно-ориентированном коде нередко требуется управлять доступностью данных объекта. Большинство членов, будь то функции или свойства, являются открытыми в соответствии с реализацией JavaScript. Но что, если требуются закрытые функции или свойства? В языке JavaScript отсутствуют простые и понятные модификаторы доступа вроде `private`, `protected` или `public`, управляющие правами доступа к члену свойства. Но в то же время можно добиться эффекта открытых членов. Более того, можно предоставить особый доступ к этим закрытым членам через функцию, которую Дуглас Крокфорд называет *привилегированной*.

Напомним, что в JavaScript предусмотрены только две области действия: глобальная и область действия функции, выполняемой в настоящий момент. В предыдущей главе мы уже воспользовались такой возможностью в отношении замыканий — критической части реализации привилегированного доступа к закрытым членам. Это делается следующим образом: создайте сначала закрытые члены с помощью ключевого слова `var` в теле функции, строящей объект. (Вам решать, будут ли эти закрытые члены функциями или свойствами.) Затем создайте в той же самой области действия функцию с подразумеваемым доступом к закрытым данным, поскольку функция и закрытые данные относятся к одной и той же области действия. Далее введите вновь созданную функцию в сам объект, сделав открытой именно функцию, а не закрытые данные. Эта функция относится к той же самой области действия, и поэтому она по-прежнему может иметь косвенный доступ к данным. Соответствующий пример демонстрируется в листинге 3.9.

Листинг 3.9. Закрытые члены

```
var Person = {
  firstName : 'John',
  lastName : 'Connolly',
  birthDate : new Date( '1964-09-05' ),
  gender : 'male',
```

```
  getAge : function () {
    var today = new Date();
    var diff = today.getTime() - this.birthDate.getTime();
    var year = 1000 * 60 * 60 * 24 * 365.25;
    return Math.floor( diff / year );
  },

  toString : function () {
    return this.firstName + ' ' + this.lastName + ' is a ' +
           this.getAge() + ' year-old ' + this.gender;
  },

  extend : function ( config ) {
    var tmp = Object.create( this );

    for ( var key in config ) {
      if ( config.hasOwnProperty( key ) ) {
        tmp[key] = config[key];
      }
    }

    // Когда этот объект был создан?
    var creationTime = new Date();

    // Метод доступа в настоящий момент открыт
    var getCreationTime = function() {
      return creationTime;
    };

    tmp.getCreationTime = getCreationTime;
    return tmp;
  }
};

var Teacher = Person.extend( {
  job : 'teacher',
  subject : 'English Literature',
  yearsExp : 5,
  toString : function () {
    var originalStr = Person.toString.call(this);
    return originalStr + ' ' + this.subject + ' teacher.';
  }
} );

var patty = Teacher.extend( {
  firstName : 'Patricia',
  lastName : 'Hannon',
  subject: 'chemistry',
  yearsExp : 20,
```

```
  gender : 'female'
} );

console.log( patty.toString() );
console.log( 'The Teacher object was created at %s',
             patty.getCreationTime() );
```

Как видите, переменная `creationTime` является локальной по отношению к функции `extend()`. Но она недоступна за пределами функции. Если вы попытаетесь исследовать объект `Person` на консоли, например, с помощью функции `console.dir()`, то вряд ли обнаружите переменную `creationTime` в списке открытых свойств объекта `Person`. Первоначально то же самое справедливо и для функции `getCreationTime()`. Эта функция была создана в той же самой области действия, что и переменная `creationTime`, и поэтому данная функция имеет доступ к переменной `creationTime`. С помощью простого присваивания функция `getCreationTime()` присоединяется к возвращаемому экземпляру объекта. Теперь функция `getCreationTime()` является открытым методом с привилегированным доступом к закрытым данным в переменной `creationTime`.

В качестве минимальной оговорки следует заметить, что такой способ является отнюдь не самым эффективным. Всякий раз, когда получается экземпляр объекта `Person` или любой из производных от него типов, по существу, создается совершенно новая функция с доступом к контексту исполнения вызова функции `extend()`, в которой получен экземпляр объекта `Person`. С другой стороны, когда используется метод `Object.create()`, открытые функции являются ссылками на функции для того типа, который передается методу `Object.create()`. Привилегированные функции не особенно эффективны в мелких масштабах, как в данном случае. Но если ввести дополнительные привилегированные методы, то каждый из них сохранит ссылку на контекст исполнения и будет собственным экземпляром этого привилегированного метода. В итоге затраты оперативной памяти могут резко возрасти. Поэтому привилегированными методами следует пользоваться экономно, резервируя их для тех данных, которым требуется строгий контроль доступа. В противном случае следует просто иметь в виду, что большинство данных в JavaScript все равно являются открытыми.

Перспективы объектно-ориентированных возможностей JavaScript

Было бы опрометчиво не упомянуть о том, что в стандарте ECMAScript 6 предполагается внести некоторые изменения в объектно-ориентированные возможности JavaScript. Наиболее примечательным из них является внедрение ключевого слова `class`. Это ключевое слово будет служить для определения типов данных, а не классов, поскольку классы в JavaScript по-прежнему отсутствуют! Эти изменения будут также включать в себя условия на применение ключевого слова `extends` для установления взаимосвязи наследования. И наконец, для переопределения функций в порожденном типе в стандарте ECMAScript 6 отдельно предусмотрено ключевое слово `super`, предназначенное для ссылки на вариант функции в цепочке прототипов.

Но все это лишь синтаксические удобства. На самом деле под ними скрывается применение в механизме JavaScript функциональных конструкторов. Эти новые

языковые средства на самом деле не устанавливают никаких новых функциональных возможностей: они просто внедряют идиому, более удобную для тех, у кого имеется опыт программирования на других языках ООП. Хуже того, они продолжают затмевать некоторые из самых лучших языковых средств JavaScript, пытаясь привести его в соответствие с тем, как должен выглядеть "подлинный" язык ООП. Ведь язык JavaScript порой смущается надеть на себя плащ и трико, чтобы окончательно раскрыть свой подлинный потенциал.

Упаковка кода JavaScript

Перейдя от объектно-ориентированных свойств языка JavaScript, мы должны рассмотреть порядок организации кода для его широкого применения. С этой целью нам нужно установить инструментальные средства для надлежащей инкапсуляции прикладного кода, предотвратить случайное применение глобального контекста, а также найти способы сделать прикладной код повторно используемым и распространяемым. Рассмотрим различные требования к такому коду по порядку.

Пространства имен

До сих пор мы объявляли типы данных (а еще раньше — функции и переменные) как часть глобального контекста. Мы делали это не явно, а благодаря тому, что не объявляли объекты и переменные как часть любого другого контекста. Но нам бы хотелось инкапсулировать функции, переменные, объекты, типы данных и прочее в отдельном контексте, чтобы не полагаться на глобальный контекст. С этой целью мы сделаем (хотя бы поначалу) ставку на пространства имен.

Пространства имен не являются чем-то особенным для JavaScript, но, как и многие другие языковые средства в JavaScript, они имеют свои особенности. Пространство имен предоставляет контекст для переменных и функций. Само пространство имен, конечно, не является глобальным. Это нечто вроде выбора меньшего из двух зол. Вместо принадлежности различных переменных и функций окну можно сделать так, чтобы одна переменная принадлежала окну, а различные данные и функциональные возможности — этой переменной. Реализовать это нетрудно: достаточно воспользоваться литералом объекта, чтобы инкапсулировать код, который требуется скрыть от глобального контекста (листинг 3.10).

Листинг 3.10. Пространства имен

```
// Пример пространства имен
var FOO = {};

// Ввести переменную
FOO.x = 10;

// Ввести функцию
FOO.addEmUp = function(x, y) {
  return x + y;
};
```

Пространства имен лучше всего подходят в качестве специальных решений задач инкапсуляции кода, который в противном случае оказывается независимым.

Если же попытаться воспользоваться пространствами имен для всего прикладного кода, они могут быстро стать неупорядоченными по мере накапливания все больших функциональных возможностей и данных. Можно также попытаться установить одни пространства имен в других, эмулируя действие пакетов в Java. В частности, такой прием применяется в библиотеке Ext JS. Но ее разработчики потратили немало времени на обдумывание порядка организации функциональных возможностей, принадлежности кода конкретному пространству имен или подпространству имен и т.д. Широко применяя пространства имен, приходится идти на определенные компромиссы.

Кроме того, пространства имен жестко кодируются. В приведенном выше примере кода это пространство имен `FOO`, в упомянутой выше библиотеке Ext JS — пространство имен `Ext`, а в не менее распространенной библиотеке YUI — пространство имен `YAHOO`. Эти пространства имен, по существу, являются зарезервированными словами для обозначения библиотек. Но что произойдет, если разместить две или больше библиотеки в одном и том же пространстве имен аналогично применению пространства имен **$** в библиотеке jQuery? Возможны конфликты. В библиотеку JQuery был внедрен код, учитывающий возможность конфликтов, как только они возникнут. Несмотря на то что возникновение подобных конфликтов потенциально менее вероятно в вашем коде, такую возможность не следует упускать из виду. Это особенно справедливо в среде коллективной разработки, где несколько программистов имеют доступ к пространству имен, повышая вероятность случайной перезаписи или удаления пространства имен другого программиста.

Модульный шаблон

Имеются некоторые инструментальные средства, помогающие улучшить порядок применения пространств имен. С этой целью можно применить модульный шаблон, инкапсулирующий формирование пространства имен в теле функции. Это допускает разнообразные усовершенствования, включая наложение ограничения на те функции и данные, которые должно содержать пространство имен; применение открытых переменных в формирующей функции, что позволяет упростить реализацию некоторых функциональных возможностей; а также наличие функции для формирования пространства имен. Последнее означает, что в коде JavaScript можно динамически формировать часть или все пространство имен во время выполнения, а не компиляции.

Модули можно выбрать по желанию простыми или сложными. В листинге 3.11 демонстрируется простой пример создания модуля.

Листинг 3.11. Создание модуля

```
function getModule() {
  // Пример пространства имен
  var FOO = {};

  // Ввести переменную
  FOO.x = 10;

  // Ввести функцию
  FOO.addEmUp = function ( x, y ) {
    return x + y;
```

```
  };

  return FOO;
}

var myNamespace = getModule();
```

В данном примере код из пространства имен инкапсулирован в теле функции. Следовательно, когда объект `FOO` устанавливается первоначально, он оказывается закрытым в теле функции `getModule()`. А затем объект `FOO` можно возвратить любому коду, вызывающему функцию `getModule()`. И в этом коде можно воспользоваться инкапсулированной структурой данных по мере надобности, включая именование всего, что только потребуется.

Еще одно преимущество данного шаблона заключается в возможности еще раз воспользоваться уже знакомым нам замыканием, чтобы установить данные, которые являются закрытыми только в пространстве имен. Так, если в пространстве имен, т.е. в инкапсулированном в нем коде, потребуются внутренние данные или функции, их можно ввести, не делая их открытыми (листинг 3.12).

Листинг 3.12. Модули с закрытыми данными

```
function getModule() {
  // Пример пространства имен
  var FOO = {};

  // Ввести переменную
  FOO.x = 10;

  // Ввести функцию
  FOO.addEmUp = function ( x, y ) {
    return x + y;
  };

  // Закрытая переменная
  var events = [];

  FOO.addEvent = function(eventName, target, fn) {
    events.push({eventName: eventName, target: target, fn: fn});
  };

  FOO.listEvents = function(eventName) {
    return events.filter(function(evtObj) {
      return evtObj.eventName === eventName
    });
  };

  return FOO;
}

var myNamespace = getModule();
```

В данном примере реализован открытый интерфейс для ввода некоторого механизма отслеживания событий с помощью переменной `addEvents`. В дальнейшем, возможно, придется вернуться к ссылкам на события по их именам посредством переменной `listEvents`. Но конкретная совокупность событий является открытой и находится под управлением открытого прикладного программного интерфейса API, который им предоставляется, но скрыт от непосредственного доступа.

Как и пространствам имен, модулям присущ тот же самый недостаток выбора наименьшего из двух зол. Мы фактически обменяли глобальную переменную для пространства имен на глобальную функцию `getModule()`. Но не было бы лучше получить полный контроль над тем, что в конечном итоге оказывается в глобальном пространстве имен, не прибегая для этой цели к применению объектов или функций из глобальной области действия? Для этой цели, к счастью, имеется рассматриваемое далее инструментальное средство, помогающее нам поступить именно таким образом.

Немедленно вызываемые функциональные выражения

Если требуется избежать засорения глобального пространства имен, то для этой цели логично выбрать функции. Ведь функции создают свой контекст исполнения, когда они выполняются, причем этот контекст подчиняется, но не обосабливается от глобального пространства имен. Когда выполнение функции завершается, контекст исполнения становится доступным для сборки "мусора" с целью освободить выделенные для него ресурсы. Но все рассмотренные ранее функции были глобальными или оставались частью пространства имен, которое само по себе является глобальным. А нам бы хотелось иметь функцию, которая выполняется немедленно, не именуя ее и не вводя ее в глобальное или иное пространство имен или контекст. И тогда в теле такой функции мы могли бы построить нужный нам модуль. Такой объект можно было бы возвратить, экспортировать или сделать доступным иным образом, но в нашем распоряжении не было бы открытой функции для получения ресурсов, требующихся для формирования данного объекта. Именно такой замысел положен в основу немедленно вызываемого функционального выражения (IIFE).

Все рассмотренные до сих пор функции служили примерами *объявления функции*. Каким бы образом мы ни определяли функцию, будь то `funcName { ... }` или `var funcName = function() { ... }`, мы объявляли функцию, резервируя ее применение на будущее. Можем ли мы вместо этого создать *функциональное выражение*, которое будет служить функцией, которая создается и сразу же исполняется? Да, можем, но для этого нам потребуется в какой-то степени синтаксическая смелость.

Как же нам выполнить функцию? Как правило, если у функции имеется имя, мы набираем ее имя, присоединяя к нему круглые скобки и тем самым указывая, что нам требуется выполнить код, связанный с этим именем. Но мы не можем сделать то же самое с определением функции, как с таковым, поскольку в конечном итоге получим ошибку `SyntaxError`, которая вряд ли нас устроит.

Но в то же время мы можем разместить объявление функции в круглых скобках, указав тем самым синтаксическому анализатору, что это не оператор, а выражение. В круглых скобках нельзя указывать операторы, но только код, вычисляемый как выражение, а именно это нам и требуется от функции. Для реализации такой возможности нам потребуется дополнительный синтаксис в виде еще одного ряда кру-

глых скобок, которые обычно указываются в конце объявления самой функции. Этот синтаксис в полной мере демонстрируется в примере кода из листинга 3.13.

Листинг 3.13. Немедленно вызываемое функциональное выражение

```
// Обычная функция
function foo() {
  console.log( 'Called foo!' );
}

// Присваивание функции переменной
var bar = function () {
  console.log( 'Called bar!' );
};

// Функциональное выражение
(function () {
  console.log( 'This function was invoked immediately!' )
})();

// Альтернативный синтаксис
(function () {
  console.log( 'This function was ALSO invoked immediately!' )
}());
```

Сравните в данном примере два первых объявления функций с двумя другими объявлениями функциональных выражений. Функциональные выражения заключены в круглые скобки с целью сделать их именно выражениями, тогда как второй ряд круглых скобок служит для вызова выражения. Ловко!

Между прочим, имеются самые разные составляющие синтаксиса JavaScript, приводящие к немедленно вызываемым функциональным выражениям: функции как составляющие логической оценки, унарные операторы, предваряемые объявлением функции, и т.д. В статье Бена Альмана (Ben Alman), посвященной немедленно вызываемым функциональным выражениям (`http://benalman.com/news/2010/11/immediately-invoked-function-expression/`), подробно описываются допустимые синтаксические конструкции, механизм действия и причины появления немедленно вызываемых функциональных выражений.

А теперь, когда стало понятно, как создается немедленно вызываемое функциональное выражение, как же им воспользоваться? У немедленно вызываемых функциональных выражений имеется немало применений, но в данном случае нас больше всего интересует формирование модуля. Можно ли зафиксировать результат немедленного вызова функционального выражения в переменной? Разумеется, можно! Следовательно, мы можем заключить формирователь модуля в немедленно вызываемом функциональном выражении и возвратить из него модуль (листинг 3.14).

Листинг 3.14. Формирователь модуля в немедленно вызываемом функциональном выражении

```
var myModule = (function () {
  // Закрытая переменная
  var events = [];
```

```
  return {
    x : 10,
    addEmUp : function ( x, y ) {
      return x + y;
    },
    addEvent : function ( eventName, target, fn ) {
      events.push( {eventName : eventName, target : target, fn : fn} );
    },
    listEvents : function ( eventName ) {
      return events.filter( function ( evtObj ) {
        return evtObj.eventName === eventName
      } );
    }
  };
})();
```

В данном примере мы внесли некоторые изменения в код. Во-первых, мы фиксируем теперь результат выполнения фабричного немедленно вызываемого функционального выражения в переменной `myModule`, а не в `myNamespace`. И во-вторых, возвращаем объект непосредственно, вместо того, чтобы сначала создавать, а затем возвращать этот объект. Благодаря этому упрощается прикладной код, а также сокращается место, резервируемое для объекта, которым мы так и не воспользуемся.

Шаблон IIFE открывает немало новых возможностей, включая применение библиотек или других функциональных средств по мере надобности. Круглые скобки в конце функционального выражения остаются теми же самыми, как и при вызове обычной функции. Следовательно, мы можем передавать аргументы немедленно вызываемому функциональному выражению, чтобы воспользоваться ими в его теле. В качестве примера в листинге 3.15 показано, как получить доступ к функциональным возможностям библиотеки jQuery из немедленно вызываемого функционального выражения.

Листинг 3.15. Передача аргументов немедленно вызываемому функциональному выражению

```
// Здесь $ обозначает библиотеку jQuery только для
// всей области действия модуля
var myModule = (function ($) {
  // Закрытая переменная
  var events = [];

  return {
    x : 10,
    addEmUp : function ( x, y ) {
      return x + y;
    },
    addEvent : function ( eventName, target, fn ) {
      events.push( {eventName : eventName, target : target, fn : fn} );
      $( target ).on( eventName, fn );
    },
    listEvents : function ( eventName ) {
      return events.filter( function ( evtObj ) {
```

```
        return evtObj.eventName === eventName
      } );
    }
  };
})(jQuery); // Предполагается, что библиотека jQuery включена раньше
```

Сначала мы передаем ссылку **$** на библиотеку jQuery немедленно вызываемому функциональному выражению, а затем обращаемся к ней по этой ссылке в теле данного выражения, где она применяется в функции `addEvent()` для ввода обработчика событий в модель DOM. (Не обращайте особого внимания на синтаксис, поскольку он не составляет основную суть данного примера!)

Опираясь на данный пример, нетрудно себе представить систему, где модули, формируемые в немедленно вызываемых функциональных выражениях, взаимодействуют друг с другом, обмениваясь аргументами и применяя библиотеки, но не переходя на глобальный уровень взаимодействия. В действительности это часть тех функциональных возможностей, которые рассматриваются в следующей главе.

Резюме

Вопрос, поставленный вначале этой главы, имеет отношение к управлению кодом и состоит в следующем: можно ли написать прикладной код таким образом, чтобы следовать рекомендациям ООП, и как инкапсулировать такой код для его повторного использования? Для ответа на первую часть этого вопроса мы уделили внимание прототипному характеру языка JavaScript, его применению для формирования чего-то похожего на классы и экземпляры, но с учетом особенностей JavaScript. И реализация такой возможности оказалась проще, чем попытка принудить JavaScript действовать подобно C# или Java. А для ответа на вторую часть упомянутого выше вопроса мы рассмотрели различные варианты решений, позволяющие инкапсулировать прикладной код: пространства имен, модули и немедленно вызываемые функциональные выражения. В конечном счете, сочетая эти три языковые средства, мы получили наилучшее решение — по крайней мере, для применения глобального контекста.

ГЛАВА 4

Отладка кода JavaScript

Иногда для нас важнее не написание кода, а управление им, поскольку оно выводит нас из себя, и чтобы успокоиться, мы обращаемся к своей излюбленной видеоигре. Но нас не перестают мучить следующие вопросы: почему прикладной код нормально работает на одной машине, но перестает работать на другой? Что на самом деле означает: пользоваться оператором тройного сравнения на равенство `===` предпочтительнее, чем оператором двойного сравнения на равенство `==`? Почему так хлопотно выполнять тесты? Как упаковать прикладной код для дальнейшего распространения? В поисках ответов на эти вопросы мы наталкиваемся на другие вопросы, отвлекающие наше внимание и не имеющие непосредственного отношения к написанному коду.

Разумеется, мы не должны пренебрегать этими вопросами. Ведь нам нужно написать высококачественный код, и если нам не удается этого сделать, мы прибегаем к простым в использовании инструментальным средствам отладки. Нам требуется хорошее покрытие проверяемого кода тестами как теперь, так и для реорганизации кода впоследствии. Мы должны также подумать о дальнейшем распространении прикладного кода. Весь круг подобных вопросов обсуждается в этой главе.

Сначала в этой главе будет показано, как разрешать затруднения, возникающие в прикладном коде. Всем нам хочется стать идеальными программистами и писать весь код правильно с первого раза. Но ведь мы хорошо знаем, что подобный идеал недостижим на практике. Итак, начнем с рассмотрения инструментальных средств, имеющихся для отладки прикладного кода.

Инструментальные средства отладки

Все современные браузеры оснащены в той или иной форме набором инструментальных средств разработки. Даже в неразвитом браузере Internet Explorer 8 имелся элементарный отладчик, хотя для его установки требовались привилегии администратора системы. Теперь отладка прикладного кода в браузерах стала намного более удобной по сравнению с теми временами, когда приходилось употреблять различные операторы с вызовом функции `alert()`, а иногда и элементы DOM в качестве единственных ресурсов для отладки.

В общем, набор инструментальных средств разработки должен состоять из следующих служебных программ.

- **Консоль.** Сочетание блокнотной памяти JavaScript и места регистрации приложений.
- **Отладчик.** Инструментальное средство, пользоваться которым разработчики прикладного кода JavaScript до сих пор всячески избегали.
- **Инспектор DOM.** Большая часть работы сосредоточена на манипулировании моделью DOM и выборе команды View Source (Просмотр источника) из контекстного меню, хотя это далеко не все, на что способен инспектор. В частности, инспектор должен отражать текущее состояние модели DOM, а не первоначального источника. Большинство инспекторов DOM сопровождаются древовидным представлением и предоставляют возможность выбрать элемент DOM щелчком на нем как в самом инспекторе, так и на отлаживаемой веб-странице.
- **Сетевой анализатор.** Показывает, какие именно файлы были запрошены, какие файлы были фактически найдены и сколько времени потребовалось для их загрузки.
- **Профилировщик.** Зачастую действует грубовато, но все же лучше, чем заключение пары вызовов в оператор `new Date().getTime()`.

Имеются также расширения, которые могут быть внедрены в браузеры, чтобы предоставить пользователю дополнительные возможности для отладки, помимо встроенных в браузер. Например, расширение Postman (`http://getpostman.com`) для Chrome позволяет сформировать любой HTTP-запрос и наблюдать ответ на него. Еще одно распространенное расширение Firebug (`http://getfirebug.com`) является проектом с открытым кодом для внедрения всех инструментальных средств разработки в браузер Firefox, включая и создание собственного набора подобных средств. В этой главе типичный набор инструментальных средств называется инструментальными средствами или набором инструментальных средств разработки, если речь не идет о наборе подобных средств в конкретном браузере.

Консоль

Консоль — это место, где мы проводим немало времени как разработчики. Интерфейс консоли смоделирован после следующих уже известных уровней протоколирования в большинстве приложений: отладки (`debug`), информирования (`info`), предупреждения (`warn`), сообщения об ошибках (`error`) и регистрации (`log`). Зачастую мы обращаемся к консоли в качестве замены операторам с вызовом функции `alert()` в нашем коде, особенно при отладке. В некоторых из прежних версий браузера Internet Explorer поддерживается только регистрация, тогда как в версии Internet Explorer 11 уже поддерживаются все пять упомянутых выше уровней функций. Кроме того, на консоли поддерживается функция `dir()`, предоставляющая рекурсивный, древовидный интерфейс для объекта. Если же консоль отсутствует на избранной вами консоли, что маловероятно, воспользуйтесь кодом из листинга 4.1 для полизаполнения. (Очевидно, что это лишь полизаполнение функции регистрации. Если же вам потребуются другие функции, введите их по отдельности.)

Листинг 4.1. Полизаполнение консоли

```
if (!window.console) {
  window.console = {
    log : alert
  }
}
```

Вывод результатов на разных уровнях функций несколько отличается. Так, в браузере Chrome или Firefox метод `console.error()` включает в себя автоматическую трассировку стека. А в других браузерах (и в исходной версии Firefox) просто вводится пиктограмма и изменяется цвет текста для различения разных уровней. Вероятно, главная причина для применения различных уровней функций состоит в том, что их можно отфильтровать во всех трех основных браузерах. В листинге 4.2 приведен тестовый код, отладка которого иллюстрируется моментальными снимками экранов в браузерах Chrome, Firefox и Internet Explorer (рис. 4.1–4.3).

Листинг 4.2. Уровни функций консоли

```
console.log( 'A basic log message.' );
console.debug( 'Debug level.' );
console.info( 'Info level.' );
console.warn( 'Warn level.' );
console.error( 'Error level (possibly with a stacktrace).' );

var person = {
  name : 'John Connelly',
  age : 56,
  title : 'Teacher',
  toString: function() {
    return this.name + ' is a ' + this.age + '-year-old ' +
           this.title + '.';
  }
};

console.log( 'A person: ' );
console.dir( person );

console.log( 'Person object (implicit call to toString()): ' + person );
console.log( 'Person object as argument, similar to console.dir: ', person
);
```

Эффективное использование консольных средств

Так как же лучше всего воспользоваться функциями консоли? Как и в отношении многих языковых средств JavaScript, самое главное в данном случае — согласованность. Вы и ваша команда должны согласовать применение шаблонов с учетом разных факторов. Прежде и важнее всего то, что все консольные операторы должны

```
Developer Tools - http://localhost:63342/PJC/Chapter04/console-levels.html
Elements Network Sources Timeline Profiles Resources Audits Console AngularJS
<top frame>
A basic log message.                                                    console-levels.js:1
Debug level.                                                            console-levels.js:2
Info level.                                                             console-levels.js:3
Warn level.                                                             console-levels.js:4
▼Error level (note stacktrace).                                         console-levels.js:5
    (anonymous function)                                                console-levels.js:5
A person:                                                               console-levels.js:16
▼Object                                                                 console-levels.js:17
    age: 56
    name: "John Connelly"
    title: "Teacher"
  ▼toString: function () {
      arguments: null
      caller: null
      length: 0
      name: ""
    ▶prototype: Object
    ▶__proto__: function Empty() {}
    ▶<function scope>
  ▶__proto__: Object
Person object (implicit call to toString()): John Connelly is a 56-year-old Teacher.   console-levels.js:19
Person object as an additional argument, similar to console.dir:  ▼Object
                                                                     age: 56
                                                                     name: "John Connelly"
                                                                     title: "Teacher"
                                                                   ▶toString: function () {
                                                                   ▶__proto__: Object
                                                                                console-levels.js:20
>
```

Рис. 4.1. Вид тестового кода в браузере Chrome 40.0

```
Console - http://localhost:63342/PJC/Chapter04/console-levels.html
Console  Inspector  Debugger  Style Editor  Profiler  Network
Net  CSS  JS  Security  Logging  Clear  Filter output
"A basic log message."                                          console-levels…:1
"Debug level."                                                  console-levels…:2
"Info level."                                                   console-levels…:3
"Warn level."                                                   console-levels…:4
"Error level (possibly with a stacktrace)."                     console-levels…:5
"A person: "                                                    console-level…:16
{name: "John Connelly", age: 56, title: "Teacher", toString: person.toString()}   console-level…:17
  age: 56
  name: "John Connelly"
  title: "Teacher"
  ◢toString: person.toString()
    arguments: null                                             N
    caller: null                                                N
    length: 0
    name: ""
    ▷prototype: Object
    ▷__proto__: function ()
  ▷__proto__: Object
"Person object (implicit call to toString()): John Connelly is a 56-year-old Teacher."   console-level…:19
"Person object as argument, similar to console.dir: " {name: "John Connelly", age: 56,   console-level…:20
title: "Teacher", toString: person.toString()}
»|
```

Рис. 4.2. Вид тестового кода в браузере Firefox 35.0.1

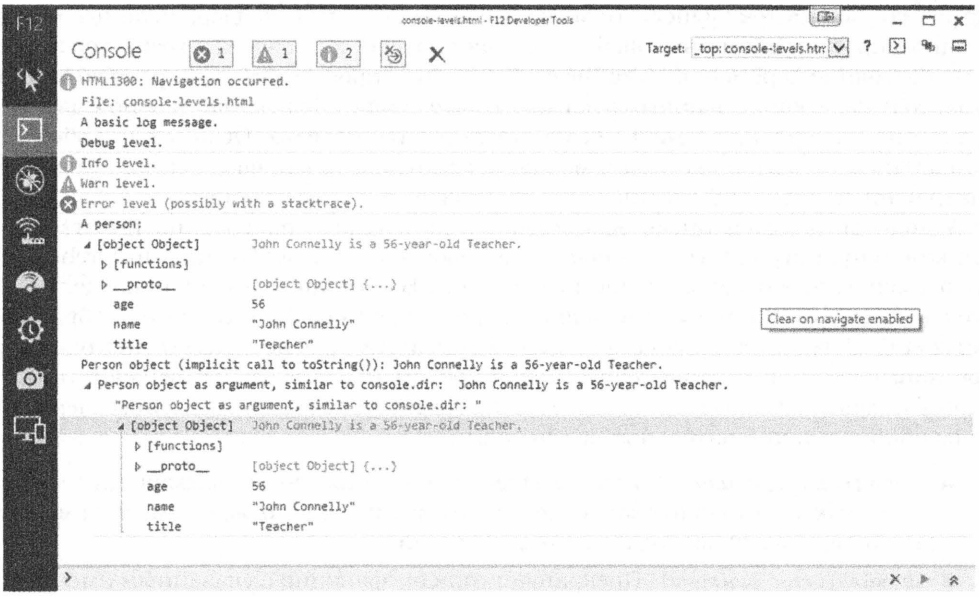


Рис. 4.3. Вид тестового кода в браузере Internet Explorer 11.0

быть удалены к моменту развертывания прикладного кода для внешнего просмотра. Включать в выходной код консольные операторы вряд ли нужно, а удалить вызовы функций консоли очень просто, как будет показано далее в главе. Не следует также забывать, что отладка, о которой речь пойдет ниже, может заменить регистрацию для одноразовых потребностей. В общем, регистрацией на консоли информации о состоянии приложения рекомендуется пользоваться, чтобы выяснить, когда регистрация была начата, позволила ли она обнаружить данные, как выглядят различные сложные объекты, и т.д. Регистрация дает хронологию жизненного цикла приложения, а также возможность просмотреть изменение его состояния. Если сравнить приложение с автострадой, то грамотную регистрацию — с километровыми знаками, указывающими достигнутый прогресс и место, с которого следует начать поиск неполадок, когда они неизбежно возникают.

Функции консоли не ограничиваются только регистрацией. Это блокнотная память JavaScript. Консоль начинает работать в однострочном режиме, в котором код JavaScript можно вводить построчно. Если же требуется ввести несколько строк кода, достаточно переключиться в многострочный режим работы, активизируемый с помощью соответствующих пиктограмм в браузерах Firefox и Internet Explorer, тогда как в Chrome ввод каждой строки кода следует завершить нажатием комбинации клавиш <Shift+Enter>. В однострочном режиме работы можно вводить различные операторы JavaScript, пользуясь функцией автозавершения простым нажатием клавиши <Tab> или <→>. Консоль снабжена простой предысторией, которая позволяет перемещаться по введенным строкам кода вверх и вниз с помощью клавиш <↑> и <↓>. На консоли поддерживается также текущее состояние, благодаря которому переменные, определенные в предыдущей строке (или нескольких строках) кода, остаются доступными до полной перезагрузки страницы.

Последняя возможность заслуживает особого внимания. На консоли доступно все состояние интерпретатора JavaScript, что очень удобно и полезно. Так, если вы

загрузили библиотеку jQuery, то можете ввести команды, поддерживаемые в прикладном программном интерфейсе этой библиотеки. Если же вам требуется проверить состояние переменной в конце страницы или просмотреть конкретную анимацию, то и здесь консоль пригодится как нельзя лучше. На консоли можно вызывать функции, проверять переменные, манипулировать моделью DOM и т.д. Любые команды, набираемые на консоли, можно рассматривать как вводимые в только что завершенный сценарий с полным доступом ко всему его состоянию.

У консоли имеется также расширенный прикладной программный интерфейс API командной строки. Первоначально он был создан разработчиками Firebug, но его элементы перенесены и в другие браузеры. В настоящее время он поддерживается в браузере Chrome и в исходной версии браузера Firefox, но не в браузере Internet Explorer. Этот прикладной программный интерфейс находит многочисленное применение, и поэтому настоятельно рекомендуется ознакомиться с ним по адресу `https://getfirebug.com/wiki/index.php/Command_Line_API`. Ниже перечислены наиболее примечательные его функции.

- **`debug(`*имя_функции*`)`**. Когда вызывается функция, обозначаемая аргументом *имя_функции*, отладчик запускается автоматически, прежде чем будет выполнена первая строка кода в данной функции.
- **`undebug(`*имя_функции*`)`**. Прекращает отладку функции с указанным именем.
- **`include(`*url*`)`**. Извлекает удаленный сценарий на веб-страницу. Эта функция очень удобна для извлечения другой отладочной библиотеки, какого-нибудь другого кода, иначе манипулирующего моделью DOM, и вообще всего, что угодно.
- **`monitor(`*имя_функции*`)`**. Активизирует регистрацию всех вызовов в функции с указанным именем. Не оказывает никакого влияния на вызовы **`console.*`**, а вводит специальный вызов функции `console.log()` при каждом вызове функции с указанным именем. В итоге регистрируется имя функции, ее аргументы и их значения.
- **`unmonitor(`*имя_функции*`)`**. Выключает регистрацию, активизированную функцией `monitor()`, при всех вызовах функции с указанным именем.
- **`profile([`*название*`])`**. Включает профилировщик JavaScript. Имеется возможность передать необязательное название данного профиля.
- **`profileEnd()`**. Завершает выполнение текущего профилировщика и выводит отчет, возможно, с названием указанного профиля.
- **`getEventListeners(`*элементы*`)`**. Получает обработчики событий для предоставляемого элемента.

Благодаря консоли разработчики получают полноценное инструментальное средство для взаимодействия с прикладным кодом. Консоль играет заметную роль и в работе следующего инструментального средства — отладчика.

Отладчик

Многие годы одним из препятствий для широкого применения JavaScript служило то обстоятельство, что он не считался "настоящим" языком программирования, поскольку ему недоставало таких инструментальных средств, как отладчик. Ныне

отладчик стал стандартным в арсенале средств всех разработчиков. Все современные браузеры оснащены инструментальными средствами отладки, которые позволяют обследовать приложение и отладить прикладной код. Рассмотрим, каким образом эти инструментальные средства действуют, начиная с отладчика.

Принцип действия отладчика довольно прост: разработчику нужно остановить выполнение отлаживаемого приложения и исследовать его текущее состояние. Несмотря на то что последнюю часть отладки можно выполнить, благоразумно применяя операторы с вызовом функции `console.log()`, первую часть нельзя осуществить без отладчика. Как только выполнение приложения приостановится, потребуется доступ к ряду других инструментальных средств, поскольку нужно каким-то образом активизировать отладчик. В самом прикладном коде можно ввести простой оператор `debugger`, чтобы активизировать отладчик в данной строке кода. Как упоминалось ранее, на консоли можно было бы также набрать команду **`debug`** вместе с именем функции, при вызове которой запустится отладчик. Но возобновить работу отладчика проще всего, установив точку прерывания в прикладном коде.

Точки прерывания позволяют выполнить код JavaScript до определенной точки, а затем приостановить в ней приложение. Дойдя до точки прерывания, мы можем приступить к анализу текущего состояния приложения. В этой точке можно просмотреть содержимое переменных, область их действия и пр. В нашем распоряжении имеется также меню навигации как минимум с четырьмя пунктами: для входа в текущую функцию (т.е. перехода на другой уровень в глубину стека), выхода из текущей функции (т.е. завершения текущего фрейма стека и возобновления отладки в точке возврата из этого фрейма), пропуска текущей функции (т.е. выполнения отлаживаемого кода без вхождения в эту функцию), а также возобновления выполнения (т.е. выполнения отлаживаемого кода до конца или следующей точки прерывания).

Инспектор DOM

Многие приложения JavaScript вносят обширные изменения в состояние модели DOM, причем настолько существенные, что зачастую бесполезно ссылаться на конкретные моменты в исходном коде HTML после загрузки страницы. Инспектор DOM отражает текущее состояние модели DOM (а не ее состояние, когда страница загружена). Он должен динамично и мгновенно обновить текущее состояние модели DOM всякий раз, когда в нее вносятся изменения. Инструментальные средства разработки входят в состав инспектора DOM как стандартная возможность.

Сетевой анализатор

С момента выхода в свет предыдущего издания данной книги технология Ajax превратилась из экзотического языкового средства JavaScript в стандартное средство в арсенале тех, кто профессионально программирует на JavaScript. Как правило, получать информацию по Ajax-запросам можно на консоли и в сетевом анализаторе. Это дает возможность сортировать конкретные типы запросов (XHR/Ajax, HTML, сценариев, изображений и т.д.). Каждый запрос должен получить свою точку входа, в которой обычно предоставляются сведения о состоянии запроса (а также код ответа и ответное сообщение), куда он направлен (полный URL), сколько данных было обменено и как долго продолжался запрос. Углубившись в анализ отдельного запроса, можно обнаружить заголовки запроса и ответа, предварительно просмотреть обработанные и необработанные данные в зависимости от их типа. Так, если

приложение делает запрос данных в формате JSON, сетевой анализатор сообщает о необработанных данных (в простой символьной строке) и потенциально передает эту строку, пропуская ее через средство форматирования JSON, чтобы показать конечный результат запроса. Вид сетевого анализатора в браузере Chrome 40.0 приведен на рис. 4.4, а в браузере Firefox 35.0.1 — на рис. 4.5.

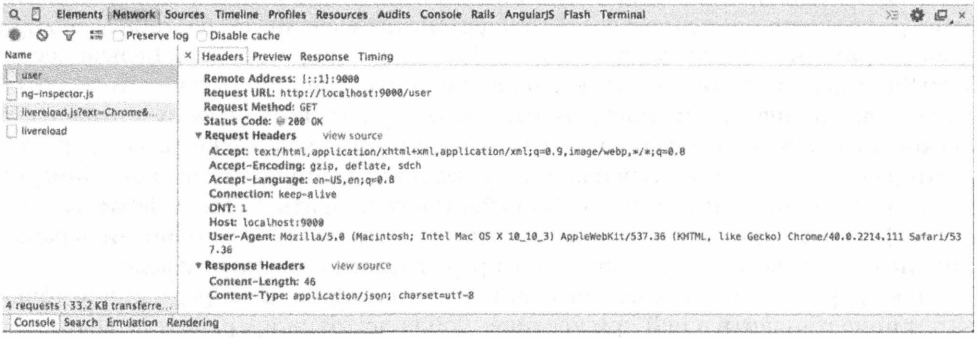


Рис. 4.4. Вид сетевого анализатора в браузере Chrome 40.0

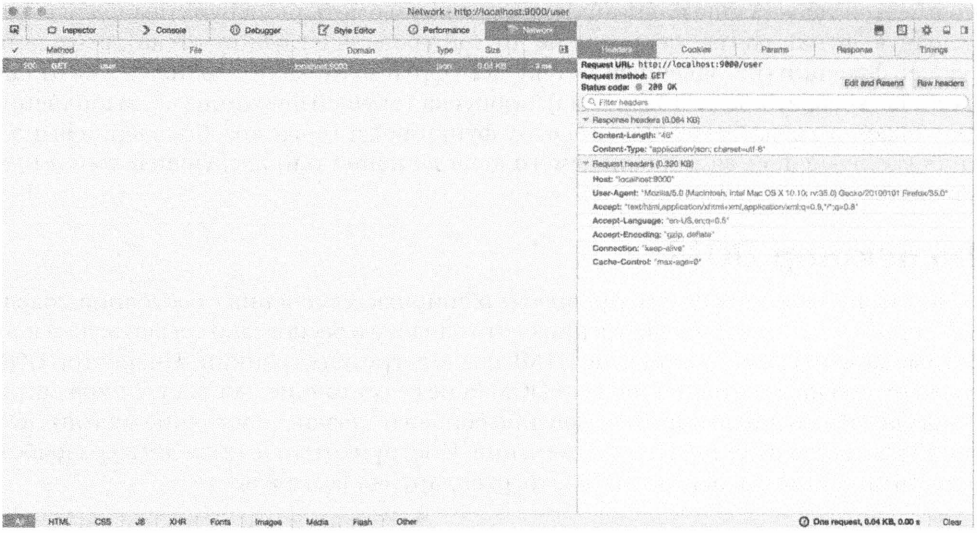


Рис. 4.5. Вид сетевого анализатора в браузере Firefox 35.0.1

Используя профилировщик динамической памяти и временн*ую* шкалу, можно выявить утечки памяти как в настольной системе, так и в мобильных устройствах. Рассмотрим сначала временн*ую* шкалу.

Временная шкала

Как только вы обнаружите, что ваша страница действует медленнее, временн*ая* шкала поможет вам выяснить, сколько оперативной памяти потребляет ваша страница во времени. Функциональные средства на временн*ой* шкале очень похожи на

те, что имеются у всех современных браузеров, поэтому ради краткости рассмотрим их на примере браузера Chrome.

Перейдите к панели Timeline (Временная шкала) и сбросьте флажок Memory (Память). Как только вы щелкнете слева на кнопке Record (Запись), начнется запись данных о потреблении оперативной памяти вашим приложением. Воспользуйтесь своим приложением во время записи как обычно, чтобы выявить утечки памяти. Остановите запись, и тогда появится график, наглядно показывающий, сколько оперативной памяти ваше приложение употребило во времени. Если вы обнаружите, что потребление оперативной памяти вашим приложением во времени не падает и остается на прежнем уровне после сборки "мусора", значит, имеет место утечка памяти.

Профилировщик

Если вы обнаружите утечку памяти, обратитесь к профилировщику и попытайтесь понять, что происходит. В частности, выясните, сколько оперативной памяти потребляет браузер и сколько ее освобождается после сборки "мусора". Сборка "мусора" выполняется в браузере автоматически. В ходе этого процесса браузер просматривает все созданные объекты. И те объекты, на которые больше не делаются ссылки, удаляются, освобождая занимаемую ими оперативную память.

Инструментальные средства профилирования встроены во все современные браузеры. Они позволяют выяснить, какие объекты потребляют больше оперативной памяти во времени. Вид панели Profiles (Профили) в браузере Chrome 40.0 приведен на рис. 4.6, а вид аналогичной ей вкладки Performance (Производительность) в браузере Firefox 35.0.1 — на рис. 4.7.

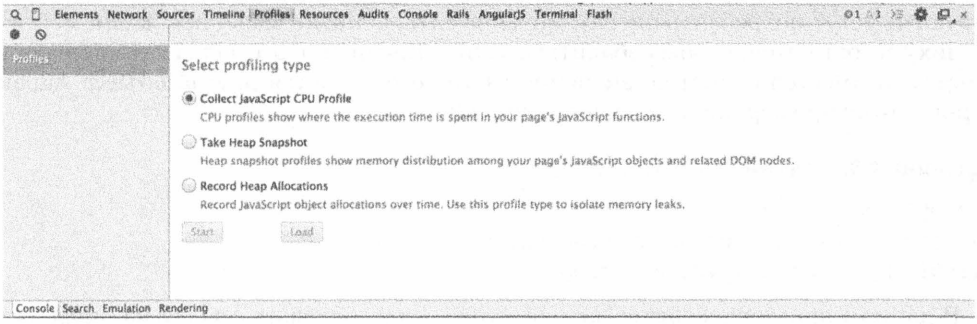


Рис. 4.6. Панель Profiles в браузере Chrome 40.0

Применение профилировщика аналогично временной шкале в том отношении, что для записи данных о приложении по ходу его действия требуется браузер. Но в данном случае делается так называемый *моментальный снимок*. Разработчики приложения Gmail рекомендуют сделать три таких снимка в следующем порядке.

1. Сделайте первый моментальный снимок.
2. Выполните действия, которые, на ваш взгляд, способны привести к утечке памяти.
3. Сделайте второй моментальный снимок.
4. Повторите те же самые действия.

5. Сделайте третий моментальный снимок.
6. Отберите объекты из первого и второго моментальных снимков в итоговом представлении третьего моментального снимка.

Рис. 4.7. Вкладка Performance в браузере Firefox 35.0.1

После этого можете приступить к просмотру всех объектов, которые еще существуют и могут потреблять оперативную память, а также выяснить, какие ссылки на них еще остаются, чтобы избавиться от них. Как правило, ссылка делается, когда у объекта имеется свойство, значением которого является другой объект. Характерный тому пример приведен в листинге 4.3.

Листинг 4.3. Создание ссылок на объекты

```
var myObject = {};
myObject.property = document.createElement('div');
mainDiv.appendChild(myObject.property);
```

В данном примере свойство `myObject.property` содержит ссылку на вновь созданный объект `div`. Метод `appendChild()` может воспользоваться этим объектом без всяких осложнений. Если в какой-то момент удалить объект `div` из модели DOM, в объекте `myObject` по-прежнему останется ссылка на объект `div`, не собранная в "мусор". Когда объекты больше не содержат ссылки, они автоматически собираются в "мусор". Для удаления ссылки можно, в частности, воспользоваться ключевым словом `delete`, как демонстрируется в листинге 4.4.

Листинг 4.4. Удаление ссылок на объекты

```
delete myObject.property;
```

Резюме

Как было показано в этой главе, современные браузеры оснащены инструментальными средствами, предоставляющими удобную среду, которая помогает полностью проанализировать работу приложения. Если вы обнаружите в своем приложении участки, где можно внести усовершенствования, временная шкала покажет вам, сколько оперативной памяти потребляется во времени. Отладчик поможет вам проанализировать значения переменных в любой момент времени. А с помощью профилировщика вы сможете выявить места, где происходят утечки памяти, а также найти способ их устранения.

ГЛАВА 5

Объектная модель документов

Работа с объектной моделью документов (DOM) является важной составляющей арсенала инструментальных средств тех, кто профессионально программирует на JavaScript. Ясное понимание особенностей написания сценариев по модели DOM дает преимущества не только в разнообразии разрабатываемых приложений, но и в их качестве. Подобно большинству других языковых средств JavaScript, модель DOM имеет непростую историю развития. Но в современных браузерах, как никогда прежде, стало проще манипулировать моделью DOM и ненавязчиво взаимодействовать с ней. Поэтому ясное представление о том, как пользоваться данной технологией и как овладеть ею, может послужить удобной отправной точкой для разработки очередного веб-приложения.

В этой главе обсуждается ряд вопросов, связанных с моделью DOM. Для тех читателей, которые только начинают осваивать работу с моделью DOM, в начале этой главы будут изложены основы, чтобы затем перейти к рассмотрению всех важных понятий. А для тех читателей, которые уже имеют некоторый опыт работы с моделью DOM, далее в этой главе будет представлен ряд эффективных приемов, которыми они могут воспользоваться в своих веб-приложениях.

Модель DOM находится на пересечении разных путей. Исторически сложилось так, что обновления интерфейса DOM не происходили синхронно с обновлениями браузеров или языка JavaScript, и поэтому возник разрыв в поддержке браузеров и модели DOM. Этот разрыв лишь усугубили ошибочные реализации. Для разрешения подобных затруднений и были созданы библиотеки вроде jQuery и Dojo. Но в современных браузерах модель DOM нормализована, а ее интерфейс немного устоялся. Поэтому разработчикам приходится решать вопрос, следует ли обращаться за помощью к библиотекам для доступа к модели DOM или делать все необходимое с помощью стандартного интерфейса DOM.

Введение в объектную модель документов

Первоначально модель DOM была создана как средство для представления *частей* HTML-документа в браузере. Используя язык JavaScript, разработчик мог анализировать формы, привязки, изображения и другие составляющие страницы, хотя и не обязательно всю страницу. Иногда такая модель DOM называется "устарев-

шей", или моделью DOM нулевого уровня. В конечном счете модель DOM превратилась в интерфейс под надзором консорциума W3C. С самого скромного начала модель DOM стала официальным интерфейсом не только для HTML-, но и для XML-документов. И хотя это далеко не самый быстрый и простой в употреблении интерфейс, он все же наиболее распространен и имеет свои реализации на большинстве языков программирования, включая Java, Perl, PHP, Ruby, Python и, разумеется, JavaScript. Работая с интерфейсом DOM, не стоит забывать следующее: все, что вы узнаете о нем из этой главы, относится к HTML и XML, хотя чаще всего мы будем в ней ссылаться только на HTML.

Консорциум Всемирной паутины (World Wide Web Consortium — W3C) надзирает за спецификацией DOM. По разным историческим причинам версии этой спецификации были обозначены как DOM Level *n*, т.е. модель DOM уровня *n*. Так, текущей (на момент написания данной книги) считалась версия DOM Level 4. Такое обозначение вызывает порой недоразумение, поскольку у дерева DOM могут быть свои уровни. Поэтому мы будем далее обозначать версии спецификации модели DOM как DOM Level с соответствующим порядковым номером, а уровни дерева DOM — просто как уровни.

Прежде всего следует вкратце обсудить структуру HTML-документов. Но поскольку эта книга посвящена языку JavaScript, а не HTML, то мы уделим основное внимание тому влиянию, которое HTML-документ оказывает на код JavaScript. С этой целью установим сначала ряд следующих простых принципов.

1. HTML-документы должны начинаться с обозначения типа документа по версии HTML 5. Это делается очень просто: `<!DOCTYPE html>`. Обозначение типа документа препятствует вхождению браузеров в режим совместимости, где браузер ведет себя менее последовательно.
2. Предпочтение следует отдавать включению в документ отдельных файлов JavaScript с помощью дескрипторов `<script>`, а не блоков сценариев или встроенных сценариев. Благодаря этому упрощается разработка (отделение кода JavaScript от HTML-разметки) и управление прикладным кодом. Блоки сценариев оказываются более удобными, чем включаемые файлы, лишь в редких случаях. Но, как правило, предпочтение следует отдавать включаемым файлам.
3. Дескрипторы `<script>` должны располагаться в нижней части HTML-документа, непосредственно перед закрывающим дескриптором `</body>`.

Третий из перечисленных выше принципов требует особого пояснения. Размещая дескрипторы `<script>` в конце страницы, мы получаем несколько преимуществ. Большая часть (если не все) HTML-документов должна содержать загружаемые файлы изображений, звука, видеозаписей, таблиц CSS и т.д. Почему это так важно? Потому что обработка кода JavaScript блокирует воспроизведение страницы! Браузеры не в состоянии воспроизвести другие элементы страницы во время синтаксического анализа, а иногда и выполнения кода JavaScript. Следовательно, загрузку кода JavaScript следует откладывать при всякой возможности до самого последнего момента. Кроме того, при соединении с мобильными устройствами или с низкой скоростью обмена данными извлечение и загрузка кода JavaScript происходит медленнее, чем в браузеры настольных систем. А загрузка сначала остальной части страницы означает, что пользователям не придется ждать, уставившись на пустую страницу, где не видно ничего, кроме вертушки, обозначающей процесс

загрузки. Данный принцип состоит в том, чтобы пользователь мог видеть в качестве ответной реакции как можно более скорую загрузку некоторой части страницы.

Так как же выглядит идеальная HTML-страница? Характерный ее пример приведен в листинге 5.1.

Листинг 5.1. Простой HTML-файл

```
<!DOCTYPE html>
<html>
<head>
  <title>Introduction to the DOM</title>
</head>
<body>
<h1>Introduction to the DOM</h1>

<p id="intro" class="test">There are a number of reasons
       why the DOM is awesome; here are some:</p>
<ul id="items">
  <li id="everywhere">It can be found everywhere.</li>
  <li class="test">It's easy to use.</li>
  <li class="test">It can help you to find what you want,
      really quickly.</li>
</ul>
<script src="01-sample.js"></script>
</body>
</html>
```

В рассматриваемых здесь примерах будут порой присутствовать встроенные блоки сценариев. В таком случае блок сценария будет появляться на странице в соответствии с его функциональными возможностями. Если местоположение блока сценария не связано с его функциональными возможностями, он будет размещаться в конце страницы аналогично включаемым сценариям.

Структура DOM

Структура HTML-документа представлена в модели DOM в виде навигационного дерева. Вся применяемая здесь терминология взята из генеалогического дерева (родители, потомки, родственники и т.д.). Для рассматриваемых здесь целей мы будем рассматривать как ствол дерева узел документа, называемый также элементом документа. Этот элемент содержит указатели на своих потомков, а каждый подчиненный узел (т.е. узел-потомок) — обратные указатели на его родительские, родственные и подчиненные узлы.

Для обозначения разных объектов HTML-дерева в модели DOM употребляется особая терминология. Практически все в модели DOM является узлом: элементы HTML-разметки, текст в этих элементах, комментарии, элемент разметки `DOCTYPE` и даже атрибуты элементов разметки! Очевидно, что мы должны каким-то образом различать все эти узлы, и для этой цели у каждого узла имеется свойство его типа, соответственно называемое `nodeType` (табл. 5.1). Мы можем запросить это свойство, чтобы выяснить тип анализируемого узла. Если получить ссылку на узел, она будет

экземпляром типа `Node`, реализующего все методы и обладающего всеми свойствами данного типа.

Таблица 5.1. Типы узлов и их постоянные значения

Наименование узла	Значение типа узла
ELEMENT_NODE	1
ATTRIBUTE_NOTE (не рекомендован к употреблению)	2
TEXT_NODE	3
CDATA_SECTION_NODE (не рекомендован к употреблению)	4
ENTITY_REFERENCE_NODE (не рекомендован к употреблению)	5
ENTITY_NODE (не рекомендован к употреблению)	6
PROCESSING_INSTRUCTION_NODE	7
COMMENT_NODE	8
DOCUMENT_NODE	9
DOCUMENT_TYPE_NODE	10
DOCUMENT_FRAGMENT_NODE	11
NOTATION_NODE (не рекомендован к употреблению)	12

Типы узлов, обозначенные как не рекомендованные к употреблению, считаются устаревшими и могут быть удалены, хотя это маловероятно. Но вполне возможно, что они по-прежнему действуют теперь, как и прежде.

Как следует из табл. 5.1, узлы имеют разную специализацию, которая соответствует интерфейсам в спецификации DOM. Особый интерес представляют документы, элементы и атрибуты разметки, а также текст. У каждого из узлов имеется свой тип реализации: `Document`, `Element`, `Attr` и `Text` соответственно.

На заметку Особая роль принадлежит атрибутам. В версиях спецификации DOM Level 1, 2 и 3 интерфейс **`Node`** был реализован в интерфейсе **`Attr`**. Но для версии DOM Level 4 это уже не так. Правда, это более благоразумное, чем любое другое решение. Более подробно об этом речь пойдет далее в разделе, посвященном атрибутам.

В общем, тип `Document` служит для управления HTML-документом в целом. Каждый дескриптор в документе относится к типу `Element`, который, в свою очередь, разделяется на отдельные типы элементов HTML-разметки (например, `HTMLLIElement`, `HTMLFormElement`). Атрибуты элемента разметки представлены экземплярами типа `Attr`. Любой простой текст в элементе разметки типа `Element` является текстовым узлом, представленным типом `Text`. Не все эти подтипы наследуют от типа `Node`, но именно с ними чаще всего приходится взаимодействовать.

А теперь рассмотрим структуру документа из листинга 5.1. Весь этот документ от дескриптора `<!DOCTYPE html>` до дескриптора `</html>` относится к типу `Document`. Сам же тип документа является экземпляром типа `Doctype`, а элемент разметки `<html>...</html>` — первым и главным элементом типа `Element`. Он содержит порожденные элементы разметки типа `Element` для дескрипторов `<head>` и `<body>`. Углубляясь дальше в документ, мы обнаруживаем, что у элемента разметки `<p>` в дескрипторе `<body>` имеются два атрибута: `id` и `class`. Тот же самый элемент разметки `<p>` содержит единственный узел типа `Text`. Иерархическая структура данного

документа дублируется во взаимосвязях между экземплярами различных типов DOM. Поэтому мы должны рассмотреть эти взаимосвязи более подробно.

Взаимосвязи в модели DOM

Рассмотрим следующий очень простой фрагмент документа, чтобы выявить взаимосвязи между его узлами:

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

Каждая часть этого фрагмента разделяется на узлы модели DOM с указателями на другие узлы (родительские, порожденные, родственные). Если наглядно отобразить взаимосвязи, существующие в данном фрагменте, они будут выглядеть так, как показано на рис. 5.1. Каждая часть данного фрагмента, где прямоугольниками со скругленными углами обозначены элементы разметки, а обычными прямоугольниками — текстовые узлы, показана на рис. 5.1 вместе с имеющимися в ней ссылками.

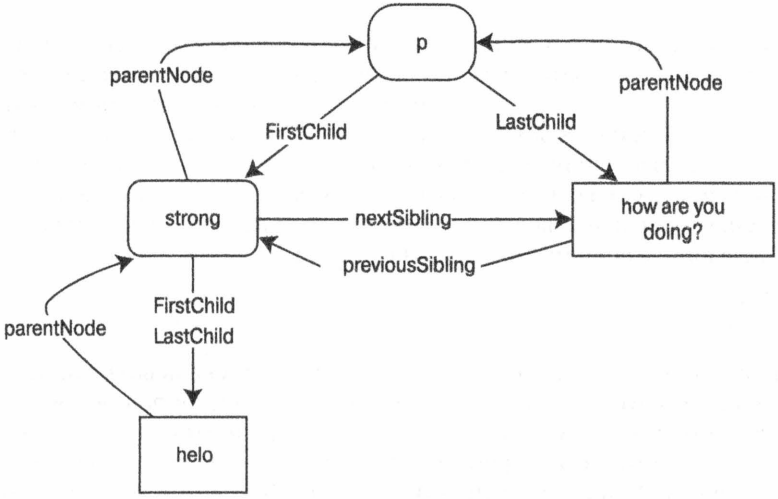


Рис. 5.1. Взаимосвязи между узлами

Каждый узел DOM содержит совокупность указателей, которые могут быть использованы для ссылки на родственные узлы. С их помощью можно научиться перемещаться по модели DOM. Все имеющиеся указатели приведены на рис. 5.2. Каждое из свойств, доступных во всяком узле DOM, является указателем на другой экземпляр класса `Node` или его подкласс. Единственным исключением из этого правила является свойство `childNodes`, обозначающее совокупность всех узлов, порожденных от текущего узла. И, разумеется, если одна из рассматриваемых здесь взаимосвязей не определена, значение свойства будет пустым. Так, в дескрипторе `<img>` не будет определено ни свойство `firstChild`, ни свойство `lastChild`.

Если пользоваться разными указателями, то можно перейти к любому элементу разметки или текстовому блоку на странице. Напомним, что в листинге 5.1 приведен пример типичной HTML-страницы. Если прежде мы рассматривали ее с точки зрения типов JavaScript, то теперь — с точки зрения модели DOM.

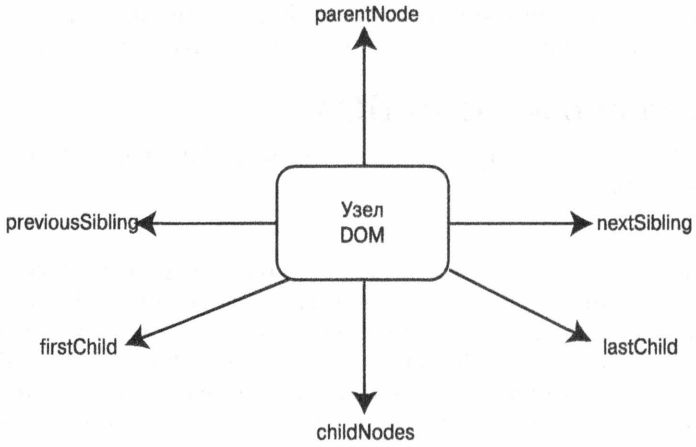


Рис. 5.2. Перемещение по модели DOM с помощью указателей

В рассматриваемом здесь примере HTML-документа имеется элемент разметки `<html>`. Для доступа к этому элементу в коде JavaScript достаточно обратиться к свойству `document.documentElement`, непосредственно ссылающемуся на элемент разметки `<html>`. Как и в любом другом узле DOM, в корневом узле имеются все указатели, применяемые для перемещения по документу. С помощью этих указателей можно начать просмотр всего документа, переходя к любому требующемуся элементу. Например, чтобы добраться до элемента разметки `<h1>`, можно было бы воспользоваться следующим фрагментом кода JavaScript:

```
// Не пройдет!
document.documentElement.firstChild.nextSibling.firstChild
```

Но дело в том, что указатели в модели DOM могут указывать как на текстовые узлы, так и на элементы разметки. В приведенном выше фрагменте кода JavaScript на самом деле происходит указание не на элемент разметки `<h1>`, а на элемент разметки `<title>`. Почему же это происходит? Все дело в пробеле — самом неприятном и спорном свойстве XML. Как следует из листинга 5.1, между элементами разметки `<html>` и `<head>` фактически находится знак перевода строки, который считается пробелом. Это, по существу, означает, что сначала следует текстовый узел, а не элемент разметки `<head>`. Из данного примера можно извлечь следующие уроки.

- Написание ясной и понятной HTML-разметки может фактически привести к недоразумению, если попытаться просматривать модель DOM, пользуясь только указателями.
- Применение только указателей DOM для перемещения по документу может оказаться весьма многословным и непрактичным.
- В действительности указатели DOM не совсем надежны, поскольку они слишком тесно связывают логику JavaScript с HTML-разметкой.
- Нередко требуется доступ не к самими текстовым узлам, а только к окружающим их элементам разметки.

В связи с изложенным выше возникает следующий вопрос: имеется ли более совершенный способ поиска элементов в документе? Да, имеется, причем не один!

Для доступа к элементам на странице имеются два основных способа. С одной стороны, можно продолжить относительный доступ вглубь документа, иногда называемый обходом дерева DOM. Но по перечисленным выше причинам такой способ не годится для общего доступа к модели DOM. Впрочем, мы еще вернемся к обходу дерева DOM, когда станут понятнее особенности доступа к конкретным элементам документа. А с другой стороны, можно пойти по иному пути, обратившись к различным функциям извлечения элементов, предоставляемых в современном интерфейсе DOM.

Доступ к элементам DOM

Все современные реализации модели DOM содержат несколько методов, упрощающих поиск элементов на странице. С помощью этих методов и некоторых специальных функций можно намного удобнее перемещаться по дереву DOM. Рассмотрим сначала порядок доступа к одному элементу.

- Метод `document.getElementById('everywhere')`, который можно вызывать только для одного объекта документа, находит все элементы с одинаковым идентификатором *everywhere*. Это очень эффективный метод, поскольку он обеспечивает самый быстрый доступ непосредственно к элементу. Метод `getElementById()` возвращает ссылку на элемент HTML-разметки по указанному идентификатору, а иначе — пустое значение. Возвращаемый объект является экземпляром типа `Element`. О том, как обращаться с этим экземпляром, речь пойдет несколько позже.

Внимание! Как и следовало ожидать, метод **`getElementById()`** предназначен для обработки документов, а точнее — для поиска среди всех элементов единственного элемента разметки, имеющего атрибут **`id`** с указанным значением. Но если загрузить удаленный XML-документ и вызвать метод **`getElementById()`** (или воспользоваться реализацией модели DOM на любом другом языке, кроме JavaScript), то следует иметь в виду, что в таком документе атрибут **`id`** не применяется по умолчанию. Это сделано намеренно, чтобы явно указать в XML-документе, каким должен быть атрибут **`id`**. Для этой цели обычно используется определение или схема XML-разметки.

Продолжим рассмотрение методов доступа к элементам. Следующие два метода обеспечивают доступ к коллекциям элементов.

- Метод `getElementsByTagName('li')`, который можно вызывать для любого элемента, находит все подчиненные элементы по указанному имени дескриптора *li* и возвращает их в виде активного списка узлов типа `NodeList`, очень похожего на массив.
- Метод `getElementsByClassName('test')` действует аналогично методу `getElementsByTagName()` и может быть вызван из любого экземпляра типа `Element`. Он возвращает актуальную коллекцию типа `HTMLCollection` совпадающих элементов.

Оба эти метода обеспечивают доступ сразу к нескольким элементам. Если не принимать во внимание отличие в возвращаемом типе, то возвращаемая коллекция элементов в любом случае оказывается *актуальной*. Это означает, что если происходит модификация модели DOM, то она будет учтена в возвращаемой коллекции элементов вплоть до их исключения из данной коллекции. Следовательно, возвра-

щаемая коллекция элементов автоматически обновляется с учетом внесенных изменений, что очень удобно!

Несмотря на то что оба эти метода действуют одинаково, они возвращают разные типы данных. Начнем с самого простого: оба типа имеют позиционный доступ аналогично массиву. Так, в следующей строке кода:

```
var lis = document.getElementsByTagName('li');
```

в переменной `lis` сохраняется список возвращаемых элементов, а доступ ко второму элементу этого списка осуществляется по ссылке `lis[1]`. У коллекций, возвращаемых обоими методами, имеется свойство `length`, обозначающее количество элементов в коллекции. У них имеется также метод `item()` с аргументом, обозначающим позицию для доступа и возвращающим элемент, находящийся на данной позиции. И наконец, ни в одной из возвращаемых коллекций не поддерживаются методы более высокого порядка вроде `push()`, `pop()`, `map()` или `filter()`, относящиеся к типу `Array`. Если вы предпочитаете вызывать методы, относящиеся к типу `Array`, для своей коллекции элементов типа `HTMLCollection` или `NodeList`, то можете всегда сделать это так, как демонстрируется в листинге 5.2.

Листинг 5.2. Вызов методов, относящихся к типу `Array`, для коллекции элементов типа `HTMLCollection` или `NodeList`

```
// Простая функция фильтрации. Свойство nodeName типа Element
// всегда содержит имя исходного дескриптора
function filterForListItems(el) {
  return el.nodeName === 'LI';
}

var testElements = document.getElementsByClassName( 'test' );
console.log( 'There are ' + testElements.length +
             ' elements in testElements.');

// Формирование массива из элементов, собранных из
// коллекции testElements, в зависимости от того, проходят
// ли они процесс фильтрации, заданный в функции filterForListItems()
var liElements = Array.prototype.filter.call(
    testElements, filterForListItems);
console.log( 'There are ' + liElements.length +
             ' elements in liElements.');
```

Эти методы отличаются возвращаемым типом, что обусловлено особенностями реализации модели DOM в браузерах. В будущем оба метода должны возвращать экземпляры типа `HTMLCollection`, но это будущее еще не настало. Шаблоны доступа к коллекциям типа `NodeLists` и `HTMLCollections` практически одинаковы, и поэтому нам не стоит особенно беспокоиться о том, какой именно тип данных возвращает каждый из рассматриваемых здесь методов.

Применяя метод `getElementsByClassName()` или `getElementsByTagName()`, не стоит забывать, что оба они принадлежат не только к экземплярам типа `Document`, но и к экземплярам типа `Element`. Когда они вызываются из документа, то производят поиск по документу в целом. Допустим, что в разделе под дескриптором `<head>` осуществляется поиск дескрипторов `<li>` или элементов разметки с классом `foo`.

Нетрудно представить, что такой поиск столь же неэффективен, как и поиск ключей по всему дому. Вряд ли вы будете искать ключи в холодильнике или в ванной, поскольку маловероятно, чтобы вы их там оставили. Скорее всего, вы будете искать ключи в прихожей, гостиной, спальне и т.д. Поэтому старайтесь при всякой возможности ограничивать область поиска соответствующими элементами, содержащими искомые элементы. В качестве примера в листинге 5.3 демонстрируется код, возвращающий такие же результаты, как и код из листинга 5.2, но в то же время ограничивающий поиск конкретным родительским элементом.

Листинг 5.3. Ограничение области поиска

```
var ul = document.getElementById( 'items' );
var liElements = ul.getElementsByClassName( 'test' );
console.log( 'There are ' + liElements.length +
             ' elements in liElements.');
```

■ ***На заметку*** В отличие от метода **getElementsByClassName()** или **getElementsByTagName()**, метод **document.getElementById()** *недоступен* для экземпляров типа **Element**. Он доступен *только* для документа или экземпляра типа **Document**.

Рассмотренные здесь три метода доступны во всех современных браузерах и очень удобны для поиска конкретных элементов. Если вернуться к приведенному ранее примеру, где предпринималась попытка найти элемент разметки `<h1>`, то теперь такой поиск можно организовать так, как показано ниже. Код в следующей строке гарантированно работоспособен и всегда возвращает первый элемент разметки `<h1>` в документе:

```
document.getElementsByTagName('h1')[0];
```

Поиск элементов по CSS-селектору

Вам как веб-разработчику, должно быть, известны CSS-селекторы в качестве альтернативного способа выбора элементов HTML-разметки. *CSS-селектор* — это выражение, предназначенное для применения стилей CSS к ряду элементов разметки. В результате пересмотра стандарта CSS (его версии 1, 2 и 3 иногда обозначаются как CSS Level 1, Level 2 или Level 3 соответственно) в спецификацию селектора были внедрены дополнительные возможности, чтобы разработчикам было легче находить нужные им элементы разметки в документе. В связи с тем что полная реализация CSS-селекторов версии 2 и 3 в браузерах происходила медленно, вам, возможно, неизвестны некоторые новые примечательные возможности, которые они предоставляют, хотя в современных браузерах отсутствие поддержки подобных возможностей в основном устранено. Если вас интересуют все новые примечательные возможности в CSS, рекомендуем обратиться по данному вопросу к следующим страницам веб-сайта консорциума W3C:

- CSS-селекторы версии 1: `http://www.w3.org/TR/CSS1/#basic-concepts`.
- CSS-селекторы версии 2.1: `http://www.w3.org/TR/CSS21/selector.html`.
- CSS-селекторы версии 3: `http://www.w3.org/TR/css3-selectors/`.

Возможности, предоставляемые по спецификации CSS-селекторов, обычно похожи в том отношении, что каждая последующая версия содержит, помимо новых, все возможности из предыдущих версий. Например, версия CSS 2.1 содержит селекторы атрибутов и порожденных узлов, тогда как версия CSS 3 обеспечивает дополнительную языковую поддержку, выбор по типу атрибута и отказ от выбора. Ниже приведены CSS-селекторы, пригодные для современных браузеров.

- Выражение `#main <div> p` находит элемент разметки с идентификатором `main`, все производные от него элементы разметки, а также все элементы, производные от элемента разметки `<p>`. Это подходящий CSS-селектор версии 1.
- Выражение `div.items > p` находит все элементы разметки `<div>` с классом `items`, а также все элементы, производные от элемента разметки `<p>`. Это достоверный CSS-селектор версии 2.
- Выражение `div:not(.items)` находит все элементы разметки `<div>` без класса `items`. Это достоверный CSS-селектор версии 3.

Доступ к элементам через CSS-селекторы обеспечивают методы `querySelector()` и `querySelectorAll()`. Достаточно предоставить методу `querySelector()` достоверный CSS-селектор, и он возвратит ссылку на первый элемент, совпадающий с данным селектором. Единственное отличие метода `querySelectorAll()` состоит в том, что он возвращает неактуальную коллекцию типа `NodeList` совпавших элементов. (Эта коллекция неактуальна потому, что для получения актуальной коллекции требуется немало ресурсов.) Аналогично методам `getElementsByTagName()` и `getElementsByClassName()`, методы `querySelector()` и `querySelectorAll()` можно вызывать из любого экземпляра типа `Element`. Область поиска следует ограничивать при всякой возможности ради повышения эффективности и ускорения возврата результатов поиска.

Итак, мы рассмотрели четыре способа доступа к элементам разметки. Какой же из них выбрать? Для доступа к одиночным элементам разметки лучше выбрать метод `document.getElementById()`, поскольку он делает это быстрее остальных методов. А для доступа ко многим элементам разметки или же к одному элементу без идентификатора лучше выбрать сначала метод `getElementsByTagName()`, затем метод `getElementsByClassName()` и, наконец, метод `querySelectorAll()`. Следует, однако, иметь в виду, что такой выбор учитывает только быстродействие. Ведь иногда большее значение имеет простота запроса, точность совпадения элементов или даже актуальность возвращаемой коллекции. Поэтому выбирайте тот метод, который в наибольшей степени отвечает вашим потребностям.

Ожидание загрузки HTML-документов, простроенных по модели DOM

Одна из трудностей работы с HTML-документами, построенными по модели DOM, состоит в том, что код JavaScript может быть выполнен перед полной загрузкой такого документа, а это способно привести к ряду осложнений в прикладном коде. Порядок выполнения операций в браузере выглядит следующим образом.

1. Синтаксический анализ HTML-разметки.
2. Загрузка внешних таблиц стилей.

3. Выполнение сценариев по мере их синтаксического анализа в документе.
4. Полное построение HTML-документа по модели DOM.
5. Загрузка изображений и внешнего содержимого.
6. Завершение загрузки страницы.

Разумеется, все это зависит в основном от структуры HTML-документа. Если в нем дескриптор `<script>` оказывается перед дескриптором `<link>`, по которому загружается таблица CSS, то сценарий JavaScript будет загружен прежде таблицы CSS. (Между прочим, такой порядок загрузки не рекомендуется, поскольку он неэффективен.) Сценарии, находящиеся в заголовке документа и загружаемые из внешнего файла, выполняются перед построением HTML-документа по модели DOM. Как упоминалось ранее, такое расположение сценариев вызывает серьезные трудности, поскольку модель DOM будет недоступна всем сценариям, выполняемым в этих двух местах. И это одна из причин, по которым дескрипторы `<script>` не рекомендуется размещать в разделе с дескриптором `<head>`. Но даже если придерживаться норм передовой практики, размещая дескрипторы `<script>` перед закрывающим дескриптором `<body>`, то все равно существует вероятность того, что документ, построенный по модели DOM, не будет готов для обработки по сценарию JavaScript. Правда, это препятствие можно обойти целым рядом способов.

Ожидание загрузки страницы

Наиболее распространенный способ состоит в том, чтобы просто ожидать полной загрузки страницы перед выполнением операций с моделью DOM. Чтобы реализовать такой способ, достаточно связать функцию, которая будет запускаться на выполнение при загрузке страницы, с событием загрузки в оконном объекте. Более подробно события обсуждаются в главе 6. Пример выполнения кода, связанного с моделью DOM, после загрузки страницы демонстрируется в листинге 5.4.

Листинг 5.4. Функция `addEventListener()`, связывающая обратный вызов со свойством `load` оконного объекта

```
// Ожидать до тех пор, пока не загрузится страница
// (Использовать функцию addEventListener(),
// описываемую в следующей главе)
window.addEventListener('load', function() {
  // Выполнить операции с HTML-документом,
  // построенным по модели DOM
  var theSquare = document.getElementById('square');
    theSquare.style.background = 'blue';
});
```

И хотя такая операция может быть самой простой, она всегда будет самой медленной. Из приведенного выше порядка операций загрузки следует, что страница загружается в последнюю очередь. Событие загрузки не наступает до тех пор, пока не загрузятся все файлы, указанные в элементах разметки с атрибутами `src`. Это означает, что если на странице имеется значительное количество изображений, видеозаписей и прочего мультимедийного содержимого, пользователям такой страницы придется ждать до тех пор, пока не загрузится весь сценарий JavaScript. С дру-

гой стороны, такое решение является наиболее пригодным с точки зрения обратной совместимости.

Ожидание подходящего события

Если вы пользуетесь современным браузером, то можете проверить наступление события `DOMContentLoaded`. Данное событие наступает по завершении загрузки и синтаксического анализа документа. Это соответствует полному построению HTML-документа по модели DOM. Следует, однако, иметь в виду, что изображения, таблицы стилей, видео- и звукозаписей и прочее мультимедийное содержимое не может быть загружено полностью к тому моменту, когда наступит данное событие. Если требуется запустить прикладной код после загрузки конкретного файла изображения или видеозаписи, то рекомендуется воспользоваться событием загрузки для данного конкретного дескриптора. А если требуется подождать до тех пор, пока не загрузятся все файлы, указанные в атрибуте `src`, то лучше воспользоваться событием загрузки в окно. Характерный тому пример приведен в листинге 5.5.

Листинг 5.5. Применение события `DOMContentLoaded`

```
document.addEventListener('DOMContentLoaded' functionHandler);
```

Событие `DOMContentLoaded` не поддерживается в браузере Internet Explorer 8, но в то же время можно проверить, изменилось ли состояние готовности в документе. В листинге 5.6 показано, как определить, был ли HTML-документ, построенный по модели DOM, загружен с учетом межбраузерной совместимости.

Листинг 5.6. Применение события `DOMContentLoaded` с учетом межбраузерной совместимости

```
if(document.addEventListener){
    document.addEventListener('DOMContentLoaded', function(){
      document.removeEventListner('DOMContenLoded',arguments.callee);
})else if(document.attachEvent){
    document.attachEvent('onreadystatechange', function(){
      document.detachEvent('onreadystatechange', arguments.callee,);
}
```

Получение содержимого элемента разметки

Все элементы разметки в модели DOM могут содержать одно из трех: текст, другие элементы разметки или и то и другое в определенном сочетании. Для большинства типичных ситуаций характерно первое и последнее. В этом разделе будут рассмотрены наиболее употребительные способы извлечения содержимого элементов разметки.

Извлечение текста из элемента разметки

Извлечение текста из элемента разметки относится к тем задачам, которые чаще всего сбивают с толку тех, кто только начинает работать с моделью DOM. Но в то же время это задача, которую приходится решать тем, кто работает как с HTML-документами, так и с XML-документами, построенными по модели DOM, и поэтому нужно знать способы оптимального решения подобной задачи. В примере структу-

ры DOM, приведенной на рис. 5.3, корневой элемент разметки <p> содержит элемент разметки и текстовый блок, а сам элемент разметки — текстовый блок.

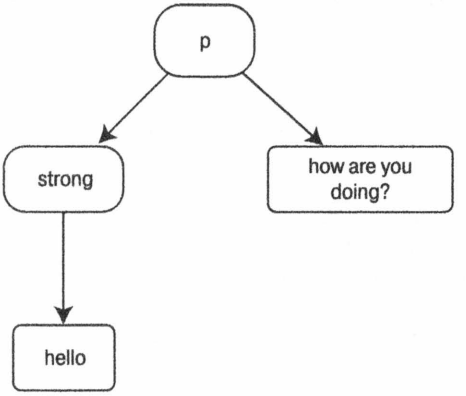


Рис. 5.3. Пример структуры DOM, состоящей из элементов разметки и текста

Посмотрим, как извлечь текст из каждого упомянутого выше элемента разметки. Для этого проще всего начать с элемента разметки , поскольку он содержит только один текстовый узел.

Следует заметить, что существует свойство `innerText`, в котором фиксируется текст из элемента разметки во всех браузерах, несовместимых с Mozilla. В этом отношении данное свойство очень удобно. Но, к сожалению, это свойство непригодно для заметной части остальных браузеров, а также для XML-документов, построенных по модели DOM, и поэтому приходится искать другие жизнеспособные альтернативы.

Особенность извлечения текстового содержимого элемента разметки состоит в том, что текст содержится не в самом элементе разметки, а в производном от него текстовом узле, что может показаться странным, хотя об этом не следует забывать. В примере, приведенном в листинге 5.7, показано, каким образом текст извлекается из элемента разметки средствами DOM. При этом предполагается, что переменная `strongElem` содержит ссылку на элемент разметки .

Листинг 5.7. Извлечение текстового содержимого из элемента разметки

```
// Все браузеры, несовместимые с Mozilla:
strongElem.innerText

// Все платформы, включая и браузеры, несовместимые с Mozilla:
strongElem.firstChild.nodeValue
```

А теперь, когда стало понятно, как извлечь текстовое содержимое из одиночного элемента разметки, необходимо выяснить, как извлечь из элемента разметки <p> смешанное текстовое содержимое. С этой целью можно также разработать обобщенную функцию, чтобы извлекать текстовое содержимое из любого элемента разметки, независимо от того, что именно он содержит (листинг 5.8). Так, в результате вызова функции `text(`*элемент*`)` возвращается символьная строка, содержащая

смешанное текстовое содержимое элемента разметки и всех производных от него элементов.

Листинг 5.8. Обобщенная функция для извлечения текстового содержимого из элемента разметки

```
function text(e) {
  var t = '' ;
  // Если был передан элемент, получить производные от него
  // элементы, а иначе — допустить, что это массив
  e = e.childNodes || e;

  // Обойти все производные узлы
  for ( var j = 0; j < e.length; j++ ) {
    // Если же это не элемент разметки, присоединить
    // его текстовое значение, а иначе — обойти все
    // элементы, производные от данного элемента
    t += e[j].nodeType != 1 ?
    e[j].nodeValue : text(e[j].childNodes);
  }

  // Возвратить совпадающий текст
  return t;
}
```

Используя приведенную выше обобщенную функцию, можно извлечь текстовое содержимое из элемента разметки `<p>`, упоминавшегося в предыдущем примере. Ниже приведен фрагмент кода, который позволяет это сделать.

```
// Извлечь текстовое содержимое из элемента разметки <p> типа Element
var pElm = document.getElementsByTagName ('p');
console.log(text( pElem ));
```

Данная функция примечательна тем, что она вполне пригодна для обработки как HTML-, так и XML-документов, построенных по модели DOM. Это означает, что теперь в нашем распоряжении имеется надежный способ извлечения текстового содержимого из любого элемента разметки.

Извлечение HTML-содержимого из элемента разметки

В отличие от извлечения текстового содержимого, извлечение HTML-содержимого из элемента разметки является одной из самых простых задач, которые могут быть выполнены над документом, построенным по модели DOM. Благодаря функциональному средству, внедренному разработчиками браузера Internet Explorer, теперь все современные браузеры включают дополнительное свойство `innerHTML` в каждый HTML-документ, построенный по модели DOM. С помощью этого свойства можно извлечь все HTML-содержимое и текстовое содержимое из элемента разметки. Кроме того, свойство `innerHTML` позволяет делать это, как правило, намного быстрее, чем рекурсивный поиск всего текстового содержимого в элементе разметки. Но этот путь не усыпан розами. Ведь браузеру дано самому решать, как реализовать свойство `innerHTML`, а поскольку для этого не существует никаких норм, то браузер

может возвратить любое содержимое, какое он сочтен нужным. В качестве примера ниже перечислен ряд необычных программные ошибок, которые могут возникнуть в результате применения свойства `innerHTML`.

- Браузеры, несовместимые с Mozilla, не возвращают элементы разметки `<style>` в операторе со свойством `innerHTML`.
- В версии 8 и прежних версиях браузер Internet Explorer возвращает элементы разметки, выделенные всеми прописными буквами, что неудобно из соображений совместимости.
- Свойство `innerHTML` постоянно доступно лишь как свойство элементов разметки в HTML-документах, построенных по модели DOM. Попытка воспользоваться им в XML-документах, построенных по модели DOM, приведет к извлечению пустых значений.

Пользоваться свойством `innerHTML` очень просто. В результате доступа к свойству предоставляется символьная строка, содержащая HTML-содержимое элемента разметки. Если же элемент разметки не содержит никаких подчиненных элементов разметки, а только текст, то возвращается символьная строка, которая содержит только текст. Чтобы продемонстрировать, как это происходит, рассмотрим два элемента разметки, приведенных на рис. 5.2 и ниже.

```
// Получить свойство innerHTML элемента разметки <strong>.
// В итоге должна быть возвращена строка "Hello"
strongElem.innerHTML
// Получить свойство innerHTML элемента разметки <p>.
// В итоге должна быть возвращена строка
// "<strong>Hello</strong> how are you doing?"
pElem.innerHTML
```

Если вы уверены, что ваш элемент разметки не содержит ничего, кроме текста, такой способ может служить очень простой заменой тех сложностей, которые связаны с извлечением текста из элемента разметки. С другой стороны, возможность извлечь HTML-содержимое из элемента разметки позволяет построить привлекательные динамические приложения, в которых выгодно используются функции редактирования по месту. Более подробно этот вопрос обсуждается в главе 10.

Обращение с атрибутами элементов разметки

Получение и установка значения в атрибуте элемента разметки относятся к наиболее часто выполняемым операциям с HTML-документами после извлечения содержимого элементов разметки. Как правило, список атрибутов, которые содержит элемент разметки, предварительно заполняется данными, собранными из XML-представления самого элемента разметки и сохраненными в ассоциативном массиве для последующего доступа, как показано в следующем примере кода HTML на веб-странице:

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

Как только элемент разметки HTML-формы будет загружен в документ, построенный по модели DOM, а также в переменную `formElem`, он будет иметь ассоциативный массив, из которого можно извлечь пары "имя–значение" из атрибутов. Полученный результат будет выглядеть следующим образом:

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

Выяснить, существует ли атрибут в элементе разметки, не составит большого труда, если воспользоваться массивом атрибутов. Но дело в том, что такая возможность не поддерживается в браузере Safari, какова ни была для этого причина. Начиная с версии 8, в браузере Internet Explorer эта возможность поддерживается, при условии, что он работает в стандартном режиме. Так как же выяснить, существует ли искомый атрибут? С этой целью можно, в частности, вызвать метод `getAttribute()`, рассматриваемый в следующем разделе, а затем проверить, является ли возвращаемое значение пустым, как демонстрируется в листинге 5.9. Имея в своем распоряжении такую функцию и зная, как обращаться с атрибутами, можно смело приступить к получению и установке значений атрибутов.

Листинг 5.9. Выяснение, содержит ли элемент разметки определенный атрибут

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

Получение и установка значений атрибутов

Для получения и установки значений в атрибутах элементов разметки имеются два способа, выбор которых зависит от типа используемого документа, построенного по модели DOM. Для надежности рекомендуется всегда пользоваться обобщенными методами `getAttribute()` и `setAttribute()`, совместимыми с XML-документами, построенными по модели DOM. Ниже показано, как воспользоваться этими методами.

```
// Получить атрибут
document.getElementById('everywhere').getAttribute('id');
// Установить атрибут
document.getElementsByTagName('input')[0].setAttribute(
        'value', 'Your Name');
```

Помимо стандартной пары методов `getAttribute()` и `setAttribute()`, для HTML-документов, построенных по модели DOM, имеются дополнительные свойства, действующие подобно кратким методам получения и установки значений атрибутов. Эти свойства повсеместно доступны в современных реализациях модели DOM и дают немало преимуществ при написании краткого кода, хотя они гарантируются только для HTML-документов, построенных по модели DOM. В следующем примере кода демонстрируется, каким образом можно воспользоваться этими свойствами DOM для получения и установки атрибутов в HTML-документах, построенных по модели DOM:

```
// Быстро получить значение атрибута
document.getElementsByTagName('input')[0].value;

// Быстро установить значение атрибута
document.getElementsByTagName('div')[0].id = 'main';
```

Следует, однако, иметь в виду пару необычных случаев употребления атрибутов. Первый случай чаще всего возникает при доступе к атрибуту с именем класса. Если сделать ссылку непосредственно на имя класса, то свойство `elem.className` позволит установить и получить это имя. Но если воспользоваться методом `getAttribute()` или `setAttribute()`, то можно сослаться на это имя как, например, при вызове `getAttribute('class')`. Для согласованного обращения с именами классов во всех браузерах необходимо получить доступ к атрибуту `className`, используя свойство `elem.className` вместо более удобного по наименованию вызова метода `getAttribute('class')`. То же самое затруднение возникает, когда, например, требуется переименовать атрибут `for` на `htmlFor`. А второй случай связан со следующими атрибутами CSS: `cssFloat` и `cssText`. Данное конкретное соглашение об именовании возникло потому, что такие слова, как `class`, `for`, `float` и `text`, зарезервированы в JavaScript.

Чтобы обойти все эти необычные случаи и упростить процесс получения и установки значений нужных атрибутов в элементах разметки, следует воспользоваться функцией, которая возьмет на себя все эти хлопоты. Пример такой функции приведен в листинге 5.10. Если вызвать эту функцию с двумя параметрами, как, например, `attr(element, id)`, она возвратит значение указанного атрибута. А если вызвать эту же функцию с тремя параметрами, как, например, `attr(element, class, test)`, она возвратит значение указанного атрибута и его новое значение.

Листинг 5.10. Получение и установка значений атрибутов в элементах разметки

```
function attr(elem, name, value) {
  // Убедиться, что предоставлено достоверное имя
  if ( !name || name.constructor != String ) return '' ;

  // Выяснить, относится ли имя к одному из двух
  // упомянутых выше необычных случаев именования
  name = { 'for': 'htmlFor', 'className': 'class' }[name] || name;

  // И в том случае, если пользователь устанавливает значение
  if ( typeof value != 'undefined' ) {
    // Установить сначала значение кратким способом
    elem[name] = value;

    // А если возможно, то воспользоваться методом setAttribute()
    if ( elem.setAttribute )
      elem.setAttribute(name,value);
  }

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

Стандартный способ доступа и изменения атрибутов независимо от их реализации является весьма эффективным средством. В листинге 5.11 демонстрируются некоторые примеры применения функции `attr()` в ряде типичных случаев для упрощения процесса обращения с атрибутами.

Листинг 5.11. Применение функции `attr()` для установки и получения значений атрибутов из элементов разметки документа, построенного по модели DOM

```
// Установить класс для первого элемента разметки <h1>
attr( document.getElementByTagName('h1')[0], 'class', 'header' );

// Установить значение для каждого элемента разметки <input>
var input = document.getElementByTagName('input');
for ( var i = 0; i < input.length; i++ ) {
  attr( input[i], 'value', '' );
}

// Добавить обрамление к элементу разметки <input>,
// имеющему имя 'invalid'
var input = document.getElementByTagName('input');
for ( var i = 0; i < input.length; i++ ) {
  if ( attr( input[i], 'name' ) == 'invalid' ) {
    input[i].style.border = '2px solid red';
  }
}
```

До сих пор мы обсуждали только получение и установку атрибутов, чаще всего применяемых в документах, построенных по модели DOM (т.е. идентификатора, имени, класса и т.д.). Но имеется также очень удобный способ получения и установки нетрадиционных атрибутов. Например, новый атрибут можно ввести в вариант элемента разметки по модели DOM, а затем извлечь его, не прибегая к модификации физических свойств документа. Допустим, что требуется составить список определений элементов, где каждый термин превращается щелчком в развернутое определение. В листинге 5.12 приведена HTML-разметка такого списка.

Листинг 5.12. HTML-документ со списком скрытых определений

```
<html>
<head>
  <title>Expandable Definition List</title>
  <style>dd { display: none; }</style>
</head>
<body>
  <h1>Expandable Definition List</h1>

  <dl>
    <dt>Cats</dt>
    <dd>A furry, friendly, creature.</dd>
    <dt>Dog</dt>
    <dd>Like to play and run around.</dd>
    <dt>Mice</dt>
```

```
    <dd>Cats like to eat them.</dd>
  </dl>
</body>
</html>
```

Более подробно особенности обработки событий рассматриваются в главе 6, а до тех пор постараемся сделать как можно более простым код обработки событий. В листинге 5.13 приведен краткий сценарий, позволяющий построить развертываемый список определений, где можно щелкнуть на термине и показать (или скрыть) его определение.

Листинг 5.13. Динамическое переключение состояния определений в списке

```
// Ждать до тех пор, пока не будет готов документ,
// построенный по модели DOM
document.addEventListener('DOMContentLoaded', addEventClickToTerms);

// Отслеживать щелчки пользователя на термине в списке
function addEventClickToTerms(){
  var dt = document.getElementsByTagName('dt');
  for ( var i = 0; i < dt.length; i++ ) {
    dt[i].addEventListener('click', checkIfOpen);
  }
}

// Выяснить, развернуто ли уже определение.
// Требуются два родственных узла, поскольку первый
// родственный узел является текстовым (он содержит слова,
// на которых произведен щелчок). Если щелчок еще не был
// произведен, то выбрать пустой стиль '', а иначе — стиль 'none'.
// Таким образом, с помощью условного оператора if проверяются
// оба возможных варианта.
function checkIfOpen(e){
  if(e.target.nextSibling.nextSibling.style.display ==
       '' || e.target.nextSibling.nextSibling.style.display == 'none'){
    e.target.nextSibling.nextSibling.style.display = 'block';
  }else{
    e.target.nextSibling.nextSibling.style.display = 'none';
  }
}
```

Итак, мы рассмотрели способы обхода дерева DOM, исследования и модификации атрибутов элементов разметки. А теперь можно перейти к обсуждению способов создания новых элементов DOM, их ввода там, где они требуются, а также удаления тех элементов, которые больше не нужны.

Модификация модели DOM

Зная, как модифицировать модель DOM, можно выполнять любые операции: от оперативного создания специальных XML-документов до построения динамиче-

ских форм, приспосабливающихся к данным, вводимым пользователями. Открывающиеся при этом возможности безграничны. Модификация модели DOM происходит в три этапа: сначала нужно создать новый элемент, затем ввести его в DOM и, наконец, удалить его снова. Порядок выполнения всех этих этапов рассматривается в последующих разделах.

Создание узлов средствами DOM

Модификация модели DOM осуществляется в основном с помощью метода `createElement()`, который позволяет оперативно создавать элементы. Но вновь созданный элемент не сразу вводится в модель DOM, что нередко вводит в заблуждение тех, кто только начинает работать с этой моделью. Рассмотрим сначала порядок создания элемента DOM.

В качестве единственного параметра метод `createElement()` принимает имя дескриптора элемента разметки и возвращает виртуальное представление этого элемента в модели DOM без атрибутов или стилевого оформления. Если вы разрабатываете приложения, в которых применяются XHTML-страницы, формируемые средствами XSLT или снабжаемые конкретным типом содержимого, вы должны помнить, что фактически пользуетесь XML-документом и что его элементы разметки должны находиться в нужном пространстве имен XML. В качестве изящного выхода из этого затруднения можно создать простую функцию, негласно проверяющую, допускается ли в используемом HTML-документе, построенном по модели DOM, создавать новые элементы в данном пространстве имен, что характерно для XHTML-документов, построенных по модели DOM. Если такая возможность допускается, то новый элемент DOM можно создать в нужном пространстве имен XHTML, как демонстрируется в листинге 5.14.

Листинг 5.14. Обобщенная функция для создания нового элемента DOM

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

Например, используя приведенную выше функцию, можно создать простой элемент разметки `<div>` и присоединить к нему дополнительные данные следующим образом:

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

Следует также заметить, что в модели DOM имеется также метод `createTextNode()`, предназначенный для создания новых текстовых узлов. Этот метод принимает в качестве единственного аргумента текст, который требуется разместить в узле, и возвращает созданный текстовый узел. Вновь созданные элементы DOM и текстовые узлы можно теперь ввести в нужном месте документа, построенного по модели DOM.

Ввод элементов в модель DOM

Ввод элементов в модель DOM может иногда вызвать затруднение даже у тех, кто имеет опыт работы с этой моделью. Для выполнения этой операции имеются два метода. Первый метод называется `insertBefore()` и позволяет ввести элемент перед другим порожденным элементом. Так, в следующей строке кода первый элемент разметки вводится перед вторым элементом:

```
parentOfBeforeNode.insertBefore( nodeToInsert, beforeNode );
```

Итак, данный метод позволяет вводить одни узлы перед другими, включая элементы разметки и текст. А как же ввести в родительский узел последний порожденный узел? Для этой цели служит другой метод, называемый `appendChild()`. Этот метод вызывается для элемента разметки, присоединяя указанный узел в конце списка узлов, как показано ниже.

```
parentElem.appendChild( nodeToInsert );
```

В примере из листинга 5.15 демонстрируется применение обоих методов, `insertBefore()` и `appendChild()`, в прикладном коде.

Листинг 5.15. Функция ввода одного элемента разметки перед другим

```
document.addEventListener(DOMContentLoaded, 'addElement');

function addElement(){
  // Извлечь упорядоченный список из документа.
  // Напомним, что метод getElementById() возвращает массив как объект

  var orderedList = document.getElementById('myList');

  // Создать элемент разметки <li>, ввести текстовый узел,
  // а затем присоединить его к элементу разметки <li>
  var li = document.createElement('li');
    li.appendChild(document.createTextNode('Thanks for visiting'));

  // Элемент [0] обозначает порядок доступа к элементам
  // упорядоченного списка orderedList
  orderedList.insertBefore(li, orderedList[0]);
}
```

В тот момент, когда эти данные вводятся в модель DOM методом `insertBefore()` или `appendChild()`, они сразу же воспроизводятся и видны пользователю. Благодаря этому может быть обеспечена мгновенная обратная связь, что особенно удобно в интерактивных приложениях, где требуются данные, вводимые пользователями. А теперь, когда было показано, как создаются и вводятся узлы методами, поддерживаемыми в самой модели DOM, было бы особенно полезно рассмотреть альтернативные методы ввода содержимого в DOM.

Вставка HTML-разметки в модель DOM

Вставка HTML-разметки непосредственно в документ еще более распространен, чем создание и ввод обычных элементов в модель DOM. Для этого проще всего вос-

пользоваться упоминавшимся ранее свойством innerHTML. Такой прием дает возможность не только извлечь HTML-разметку из элемента, но и установить ее в элементе. Чтобы показать, насколько просто это делается, допустим, что имеется пустой элемент разметки <ol>, в который требуется ввести элемент разметки <li>. В приведенном ниже примере кода показано, как этого добиться.

```
// Ввести ряд элементов разметки <li> в элемент разметки <ol>
document.getElementsByTagName('ol')[0].innerHTML =
    "<li>Cats.</li><li>Dogs.</li><li>Mice.</li>";
```

Согласитесь, что это намного проще, чем создавать элементы DOM и связанные с ними текстовые узлы. А согласно данным, приведенным по адресу http://www.quirksmode.org, это также заметно повышает быстродействие по сравнению с применением методов из модели DOM. Тем не менее такой способ далеко не идеален. Ниже перечислен ряд трудностей, связанных с вводом элементов разметки с помощью свойства innerHTML.

- Как упоминалось ранее, у XML-документов, построенных по модели DOM, отсутствует свойство innerHTML. Это означает, что для создания элементов разметки придется и далее пользоваться традиционными методами из модели DOM.
- Свойство innerHTML отсутствует и у XHTML-документов, создаваемых клиентскими средствами XSLT, поскольку они являются только XML-документами.
- Свойство innerHTML полностью удаляет любые узлы, уже имеющиеся в элементе разметки. Это означает, что ввести узел перед элементом разметки или присоединить к нему узел не удастся так же удобно, как это можно сделать только методами из модели DOM.

Последнее вызывает особые трудности, поскольку ввод узла перед элементом разметки или присоединение к нему узла является очень удобной возможностью. В примере из листинга 5.16 показано, как этого добиться, используя те же самые методы, что и прежде.

Листинг 5.16. Ввод новых узлов DOM в существующий упорядоченный список

```
document.addEventListener('DOMContentLoaded', activateButtons);

function activateButtons(){
  // присоединить приемники событий к кнопкам
  var appendBtn = document.querySelector('#appendButon');
    appendBtn.addEventListener('click', appendNode);

  var addBtn = document.querySelector('#addButton');
    addBtn.addEventListener('click', addNode);
}

function appendNode(e){
  // получить существующие элементы разметки <li> и создать новые
  var listItems = document.getElementsByTagName('li');
  var newListItem = document.createElement('li');
    // присоединить новый текстовый узел
```

```
    newListItem.appendChild(document.createTextNode('Mouse trap.'));

    // присоединить в качестве четвертого элемента
    // к существующему списку
    listItems[2].appendChild(newListItem);
}

function addNode(e){
  // получить весь список
  var orderedList = document.getElementById('myList');

  // получить все элементы разметки <li>
  var listItems = document.getElementsByTagName('li');
  // создать новый элемент разметки <li> и присоединить
  // к нему текстовый узел
  var newListItem = document.createElement('li');
    newListItem.appendChild(document.createTextNode('Zebra.'));
    // ввести новый элемент в список, опустив второй элемент
    // на место третьего
    orderedList.insertBefore(newListItem,listItems[1]);
}
```

Как следует из данного примера, внести изменения в существующий документ совсем не трудно. Но что, если требуется пойти другим путем и удалить узлы из модели DOM? И на это, как всегда, найдется свой способ.

Удаление узлов из модели DOM

Удаление узлов из модели DOM происходит едва ли не так же часто, как и их создание и ввод. Если, например, создается динамическая форма, запрашивающая неограниченное количество элементов, то очень важно предоставить пользователю возможность удалять части страницы, которые ему больше не требуются. Возможность удалить узел инкапсулирована в метод `removeChild()`. Этот метод применяется аналогично методу `appendChild()`, но дает противоположный результат. Ниже показано, каким образом этот метод вызывается в прикладном коде.

```
NodeParent.removeChild( NodeToRemove );
```

Принимая во внимание все сказанное выше, можно создать две отдельные функции для быстрого удаления узлов. Первая функция удаляет один узел, как показано в листинге 5.17.

Листинг 5.17. Функция для удаления узла из модели DOM

```
// Удалить один узел из модели DOM
function remove( elem ) {
  if ( elem ) elem.parentNode.removeChild( elem );
}
```

В листинге 5.18 приведена функция для удаления всех порожденных узлов из элемента разметки с помощью единственной ссылки на элемент DOM.

Листинг 5.18. Функция для удаления всех порожденных узлов из элемента разметки

```
// Удалить все порожденные узлы из элемента разметки
function empty( elem ) {
  while ( elem.firstChild )
    remove( elem.firstChild );
}
```

Допустим, что требуется удалить элемент разметки `<li>`, который был введен в примере из предыдущего раздела, при условии, что пользователю предоставлено достаточно времени для просмотра содержимого этого элемента разметки, а следовательно, он может быть беспрепятственно удален. В листинге 5.19 демонстрируется код JavaScript, которым можно воспользоваться для выполнения подобной операции, чтобы достичь нужного результата.

Листинг 5.19. Удаление одного или всех элементов разметки из модели DOM

```
// Удалить последний элемент разметки <li> из элемента разметки <ol>
var listItems = document.getElementsByTagName('li');
remove(listItems[2]);

// При выполнении приведенного выше кода следующий фрагмент разметки:
<ol>
  <li>Learn Javascript.</li>
  <li>???</li>
  <li>Profit!</li>
</ol>

// преобразуется в такой фрагмент разметки:
<ol>
  <li>Learn Javascript.</li>
  <li>???</li>
</ol>

// Если выполнить функцию empty() вместо функции remove(),
var orderedList = document.getElementById('myList');
empty(orderedList);
// она просто опорожнит элемент разметки <ol>, оставив следующее:
<ol></ol>
```

Обработка пробелов в модели DOM

Вернемся к рассмотренному ранее примеру HTML-документа. Ранее при попытке обнаружить один элемент разметки `<h1>` мы испытали определенные трудности из-за наличия лишних текстовых узлов. Но если это еще допустимо для одного элемента разметки, то совсем не годится, когда требуется найти следующий элемент после элемента разметки `<h1>`, поскольку приходится каким-то образом обрабатывать пробелы. В частности, для пропуска знаков конца строки между элементами разметки `<h1>` и `<p>` придется обращаться по ссылке `.nextSibling.nextSibling`. Впрочем, не все еще потеряно. Для обработки пробелов имеется один обходной при-

ем, демонстрируемый в листинге 5.20. Этот прием позволяет удалить все текстовые узлы, содержащие только пробелы, из документа, построенного по модели DOM, чтобы упростить его обход. Это не окажет заметного влияния на воспроизведение HTML-документа, но упростит перемещение по нему вручную. Следует, однако, иметь в виду, что результаты выполнения функции из листинга 5.20 непостоянны, и поэтому ее требуется повторно выполнять после каждой загрузки HTML-документа.

Листинг 5.20. Обходной прием обработки пробелов в HTML-документах

```
function cleanWhitespace( element ) {
  // Если конкретный элемент разметки не предоставлен,
  // обработать весь HTML-документ
  element = element || document;
  // Использовать первый порожденный узел в качестве отправной точки
  var cur = element.firstChild;
  // Продолжить до тех пор, пока не останутся порожденные узлы
  while ( cur != null ) {
    // Если узел является текстовым и не содержит ничего,
    // кроме пробелов,
    if ( cur.nodeType == 3 && ! /\S/.test(cur.nodeValue) ) {
      // удалить текстовый узел,
      element.removeChild( cur );
    // а если это элемент разметки,
    } else if ( cur.nodeType == 1 ) {
      // рекурсивно обработать пробелы до конца документа
      cleanWhitespace( cur );
    }
    cur = cur.nextSibling; // Обойти порожденные узлы
  }
}
```

Допустим, что приведенной выше функцией требуется воспользоваться для поиска в документе элемента, расположенного после первого элемента разметки `<h1>`. Код для выполнения этой операции выглядит следующим образом:

```
cleanWhitespace();
// Найти элемент разметки <h1>
document.documentElement
  .firstChild      // Найти элемент разметки <h1>
  .nextSibling     // Найти элемент разметки <body>
  .firstChild      // Получить элемент разметки <h1>
  .nextSibling     // Получить смежный абзац
```

У такого приема имеются свои преимущества и недостатки. Самое главное преимущество этого приема заключается в том, что он позволяет благоразумно перемещаться по документу, построенному по модели DOM. Но в то же время этот прием оказывается довольно медленным, если учесть, что приходится обходить каждый элемент DOM и текстовый узел в поисках текстовых узлов, содержащих только пробелы. Так, если имеется документ, содержащий немало текста, его загрузка может быть существенно замедлена. И всякий раз, когда новая HTML-разметка вводится

в документ, эту часть модели DOM приходится просматривать снова, чтобы проверить наличие в ней появившихся текстовых узлов с пробелами.

Важной особенностью рассматриваемой здесь функции является применение типов узлов. Тип узла можно определить, проверив значение его свойства `nodeType`. Все типы узлов и их значения были перечислены в начале этой главы, но чаще всего употребляются следующие.

- **`Element (nodeType = 1)`**. Это проверка на совпадение с большинством элементов разметки в XML-файле. Например, элементы разметки `<li>`, `<a>`, `<p>` и `<body>` имеют значение **1** своего свойства `nodeType`.
- **`Text (nodeType = 3)`**. Это проверка на совпадение со всеми текстовыми узлами в документе. При перемещении по структуре DOM с помощью стандартных свойств `previousSibling` и `nextSibling` нередко встречаются фрагменты текста как в элементах разметки, так и между ними.
- **`Document (nodeType = 9)`**. Это проверка на совпадение с корневым элементом документа. Так, в HTML-документе это элемент разметки `<html>`.

Кроме того, для обращения к разным типам узлов в модели DOM можно воспользоваться константами, начиная с версии 9 браузера Internet Explorer. Например, вместо того чтобы запоминать значение **1**, **3** или **9** соответствующего типа узла, достаточно воспользоваться константой `document.ELEMENT_NODE`, `document.TEXT_NODE` или `document.DOCUMENT_NODE`. Постоянная очистка модели DOM от пробелов может вызвать определенные трудности, и поэтому следует искать другие способы перемещения по структуре DOM.

Простое перемещение по модели DOM

Следуя принципу перемещения только по модели DOM, т.е. имея указатели в каждом доступном для перемещения направлении, можно разработать функции, более пригодные для перемещения по HTML-документу, построенному по модели DOM. Данный конкретный принцип отражает тот факт, что большинству веб-разработчиков требуется перемещение только по элементам DOM и очень редко по родственным текстовым узлам. Для этой цели имеется целый ряд полезных функций, которыми можно воспользоваться вместо стандартных свойств `previousSibling`, `nextSibling`, `firstChild`, `lastChild` и `parentNode`. В качестве примера в листинге 5.21 демонстрируется функция, возвращающая, аналогично свойству `previousSibling`, предыдущий по отношению к текущему элемент разметки или пустое значение, если такой элемент не найден.

Листинг 5.21. Функция для поиска предыдущего элемента разметки, родственного текущему элементу

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

В листинге 5.22 демонстрируется функция, возвращающая, аналогично свойству `nextSibling`, следующий по отношению к текущему элемент разметки, или же пустое значение, если такой элемент не найден.

Листинг 5.22. Функция для поиска следующего элемента разметки, родственного текущему элементу

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

В листинге 5.23 демонстрируется функция, возвращающая, аналогично свойству `firstChild`, первый производный от текущего элемент разметки, или же пустое значение, если такой элемент не найден.

Листинг 5.23. Функция для поиска первого производного от текущего элемента разметки

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

В листинге 5.24 демонстрируется функция, возвращающая, аналогично свойству `lastChild`, последний производный от текущего элемент разметки, или же пустое значение, если такой элемент не найден.

Листинг 5.24. Функция для поиска последнего производного от текущего элемента разметки

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

В листинге 5.25 демонстрируется функция, возвращающая, аналогично свойству `parentNode`, родительский по отношению к текущему элемент разметки. Дополнительно можно предоставить числовое значение для перемещения одновременно вверх по нескольким родительским элементам разметки. Например, вызов функции `parent(elem, 2)` равнозначен вызову `parent(parent(elem))`.

Листинг 5.25. Функция для поиска родительского элемента разметки по отношению к текущему

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

С помощью всех этих функций можно быстро перемещаться по документу, построенному по модели DOM, не беспокоясь о тексте, содержащемся в каждом элементе. Например, чтобы найти элемент разметки, следующий после элемента разметки `<h1>`, теперь можно сделать следующее:

```
// Найти элемент разметки, следующий после элемента разметки <h1>
next( first( document.body ) )
```

В приведенном выше фрагменте кода необходимо обратить внимание на следующее. Во-первых, в нем присутствует новая ссылка `document.body`. Все современные браузеры предоставляют ссылку на элемент разметки `<body>` в параметре тела HTML-документа, построенного по модели DOM. Благодаря этому можно сделать прикладной код более кратким и понятным. И во-вторых, вызовы функций кажутся написанными нелогично. Как правило, мы собираемся перемещаться по документу, начиная с элемента разметки `<body>`, получив сначала первый элемент разметки, а затем следующий элемент, тогда как вызовы функций на самом деле следуют в обратном порядке.

Резюме

В этой главе обсуждалась модель DOM и ее структура. В ней были также рассмотрены взаимосвязи между узлами, типами узлов, а также доступ к элементам разметки средствами JavaScript. Имея доступ к этим элементам, можно изменять их атрибуты, вызывая методы `element.getAttribute()` и `element.setAttribute()`. В этой главе обсуждался также порядок создания и ввода новых узлов в документы, построенные по модели DOM, обработки пробелов и перемещения по модели DOM. А в следующей главе речь пойдет об обработке событий средствами JavaScript.

ГЛАВА 6

События

За время своего существования Интернет переживал и худшие, и лучшие времена. Сначала была эпоха браузера Netscape, а затем — браузера Internet Explorer. Но вместе с новыми обработчиками событий появилась и весна надежды нашей. Мы все еще переживали зиму отчаяния из-за того, что обработка в браузерах реализована по-разному. Но за последние годы солнышко стало светить ярче и появилась надежда на долгожданное тепло, когда был наконец-то стандартизирован прикладной программный интерфейс API обработки событий в браузерах — по крайней мере, большая его часть. Конечная цель написания пригодного для применения кода на JavaScript всегда состояла в получении веб-страницы, доступной для всех пользователей, независимо от того, с каким браузером или платформой они работают. Но слишком долго это означало написание кода для обработки событий по двум разным моделям. И только с появлением современных браузеров разработчикам веб-приложений больше не нужно об этом беспокоиться.

Понятие событий в JavaScript за последние годы получило прочное основание, на которое можно теперь опереться. Как только в версии 8 браузера Internet Explorer была реализована модель обработки событий, предложенная консорциумом W3C, отпала необходимость в написании целых библиотек, в которых учитывались отличия в браузерах, что дало возможность сосредоточить основное внимание на более интересных и примечательных свойствах событий. В конечном счете это привело к внедрению эффективного шаблона "модель–представление–контроллер" (MVC) в JavaScript, обсуждаемого далее в этой главе.

В начале этой главы представлен механизм действия событий в языке JavaScript. После этих теоретических положений следует рассмотрение практических приемов привязки событий к элементам разметки. Затем рассматривается информация, предоставляемая моделью событий, а также порядок ее наилучшего контроля. Попутно мы, безусловно, должны рассмотреть типы доступных нам событий. И в заключение обсуждается делегирование событий и даются рекомендации по наилучшим приемам их обработки.

Представление о событиях в JavaScript

Если посмотреть в корень любого прикладного кода, написанного на JavaScript, то можно обнаружить, что события являются тем склеивающим материалом, который держит все вместе. Независимо от того, применяется ли одностраничное приложение, полностью построенное по шаблону MVC, или только сценарий JavaScript для ввода функциональных возможностей на одной или двух страницах, взаимодействие пользователей с прикладным кодом осуществляется с помощью обработчиков событий. В коде JavaScript данные, вероятнее всего, привязываются в виде литералов объектов. Эти данные затем размещаются в модели DOM, которая служит их представлением. События, возникающие из модели DOM, обрабатываются в прикладном коде JavaScript, фиксируя взаимодействие с пользователем и направляя ход выполнения приложения. Применение модели DOM в сочетании с событиями в JavaScript составляет прочное единство, на которое опирается разработка всех современных веб-приложений.

Стек, очередь и цикл ожидания событий

Во многих языках программирования, включая и JavaScript, употребляются метафоры для описания потока управления в программе, элементов в оперативной памяти и планирования того, что должно произойти дальше. Независимо от того, где именно выполняется код, будь то глобальный контекст, тело функции или вызов одной функции из другой, он размещается в так называемом *стеке*. Так, если выполняется функция `foo()`, вызывающая функцию `bar()`, то стек состоит из трех *фреймов* в глубину: глобальный контекст, функция `foo()` и функция `bar()`. Что же произойдет в результате выполнения такого кода? Это уже сфера действия *очереди*, которая определяет порядок выполнения следующего фрагмента кода после того, как будет разрешено текущее содержимое стека. В любой момент, когда освобождается стек, его содержимое переносится в очередь, после чего стек заполняется новым фрагментом кода для исполнения. Оба эти элемента очень важны для правильного понимания событий. Но имеется еще и третий элемент, называемый *динамической памятью* (или просто *"кучей"*). Именно здесь и размещаются переменные, функции и прочие именованные объекты. Если в прикладном коде JavaScript требуется доступ к объекту, функции или переменной, он обращается к динамической памяти для доступа к нужной ему информации. Динамическая память для нас не так важна, поскольку она не играет такой же большой роли в обработке событий, как стек и очередь.

Каким же образом стек и очередь учитываются в обработке событий? Для ответа на этот вопрос необходимо ввести понятие цикла ожидания событий. Такой цикл служит для взаимодействия двух потоков исполнения в браузере: потока отслеживания событий и потока исполнения кода JavaScript.

■ ***На заметку*** Напомним, что JavaScript является однопоточным языком программирования, за исключением рабочих веб-процессов, где код JavaScript выполняется в фоновом режиме.

Эти потоки исполнения действуют совместно, перехватывая пользовательские события и сортируя их в соответствии с теми типами событий, для которых зарегистрированы обработчики. Весь этот процесс и называется *циклом ожидания событий*. Всякий раз, когда выполняется цикл ожидания событий, пользовательские

события проверяются на наличие для них зарегистрированных обработчиков. Если такие обработчики не зарегистрированы, то ничего и не происходит. Если же соответствующие обработчики событий имеются, то они переносятся в голову очереди, чтобы обработать события в JavaScript при первой же удобной возможности.

Но если очередь действует по принципу "первой же удобной возможности", то это, как правило, означает, что она вступает в действие после того, как будет разрешено текущее содержимое стека. Это может придать очереди асинхронный характер работы, особенно если стек состоит из многих фреймов в глубину или содержит долго выполняемый код. События разрешается переносить в голову очереди, но они не могут прерывать работу стека. Чаще всего эта особенность событий несущественна для разработчиков, поскольку промежуток между наступлением события, разрешением стековых фреймов и выполнением кода обработки событий может оказаться пренебрежимо малым для восприятия человеком. Тем не менее очень важно иметь в виду, что в цикле ожидания событий происходит лишь перенос событий на передний план, но не прерывание текущего выполнения кода.

Теперь должно быть понятно, каким образом браузер, очередь и стек взаимодействуют вместе, чтобы определить момент для запуска обработчика событий на выполнение. Далее мы рассмотрим механизм привязки событий к их обработчикам. Но прежде нам нужно обсудить следующие архитектурные вопросы: если щелкнуть на ссылке, которая делается на элемент в неупорядоченном списке, размеченном элементом `<div>` в теле HTML-документа, то в каких из трех упомянутых выше компонентов рассматриваемого здесь процесса будет обработано данное событие? Может ли событие быть обработано в более чем одном компоненте, и если это возможно, то какие компоненты получают событие в первую очередь? Чтобы ответить на эти вопросы, мы должны рассмотреть стадии обработки событий.

Стадии обработки событий

События в JavaScript обрабатываются в две стадии, называемые *перехватом* и *всплыванием*. Это означает, что когда в элементе разметки наступает событие (например, от щелчка, произведенного пользователем на ссылке), то варьируются элементы, которым разрешается его обработать, а также порядок его обработки. В примере, демонстрирующем на рис. 6.1 порядок обработки события, показано, какие обработчики событий и в каком именно порядке запускаются на выполнение всякий раз, когда пользователь щелкает на первом элементе разметки `<a>` веб-страницы.

Данный простой пример выполнения щелчка на ссылке наглядно показывает порядок обработки событий. Допустим, что пользователь щелкнул сначала на элементе разметки `<a>`. В таком случае первым на выполнение запускается обработчик событий от щелчка на документе, затем обработчик событий в элементе разметки `<body>`, далее обработчик событий в элементе разметки `<div>` и так далее вплоть до элемента разметки `<a>`, т.е. выполняется цикл, называемый стадией перехвата. По завершении этой стадии происходит обратное перемещение вверх по дереву и по порядку запускаются на выполнение обработчики событий в элементах разметки `<li>`, `<ul>`, `<div>`, `<body>` и от щелчка на документе.

Для построения такого механизма обработки событий существуют вполне определенные исторические причины. Когда обработка событий была впервые внедрена в браузере Netscape, то было решено применить перехват событий. Далее в браузере Internet Explorer был реализован свой вариант обработки событий по принципу их

всплывания. Это была эпоха войн браузеров, когда нередко принимались диаметрально противоположные архитектурные решения. Все это послужило серьезным препятствием для развития языка JavaScript на долгие годы, поскольку программистам приходилось тратить немало времени на сопровождение библиотек, предназначавшихся для нормализации механизма обработки событий, а также некоторых особенностей DOM, Ajax и прочих технологий.

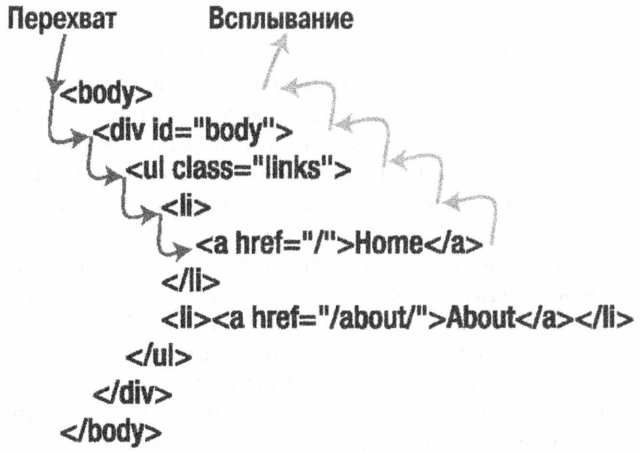


Рис. 6.1. Две стадии обработки событий

Правда, положение дел с тех пор изменилось к лучшему. Современные браузеры дают пользователям возможность выбирать стадию для перехвата событий. В действительности обработчики событий можно назначить на обеих стадия обработки событий, а это уже совсем другое дело.

Независимо от того, к какой именно стадии привязаны события, сразу становятся очевидными следующие вопросы. Прежде всего, если щелкнуть на дескрипторе привязки в пределах элемента списка, как пояснялось выше, то произойдет ли переход к атрибуту `href`, на который делается ссылка? Вероятно, такое поведение можно каким-то образом изменить. Кроме того, общий замысел стадий обработки событий таков: независимо от перехвата или всплывания событие передается по иерархии DOM. А что, если передавать событие не требуется? Можно ли предотвратить передачу события вверх (или вниз) по иерархии?

Но мы немного забегаем вперед. Ведь мы еще не обсудили порядок привязки обработчиков событий. Сделаем это теперь.

Привязка обработчиков событий

Выбор самого лучшего способа привязки обработчиков событий к элементам разметки служил предметом постоянных поисков в JavaScript. Этот поиск начался с того, что браузеры вынуждали пользователей встраивать свой код для обработчиков событий в HTML-документы. Недаром первые попытки сделать это считались черновым кодом или альфа-версией! Как оказалось впоследствии, применение встраиваемых обработчиков событий было неоптимальным и даже проблематичным с точки зрения норм передовой практики вроде отделения логики от представ-

ления. Нетрудно представить, насколько усложнится сопровождение кодовой базы, где половина критических путей зависит от кода, встраиваемого на уровне представления. Профессионально программирующие на JavaScript вряд ли пойдут на такое! Правда, этого удалось избежать благодаря развитию прикладных программных интерфейсов API в браузерах, а также внедрению в веб-разработку норм передовой практики.

В ходе острой конкуренции между браузерами Netscape и Internet Explorer для каждого из них были разработаны отдельные, но очень похожие модели регистрации событий. В конечном итоге была выбрана модель, которая применялась в браузере Netscape и стала в видоизмененном виде стандартом консорциума W3C, тогда как в браузере Internet Explorer аналогичная модель осталась прежней. И только в версии 9 браузера Internet Explorer, когда корпорация Microsoft пошла наконец-то на уступки, была реализована обработка событий по стандарту консорциума W3C. На самом деле корпорация Microsoft пошла на еще большие уступки, не рекомендовав к употреблению свой прежний прикладной программный интерфейс API для обработки событий. И это было сущим благом для разработчиков. Ведь с тех пор они могли больше не создавать и не сопровождать библиотеки, специально предназначавшиеся для учета отличий в обработке событий в разных браузерах.

Ныне события надежно регистрируются двумя способами. Традиционный способ происходит от прежнего способа присоединения обработчиков событий путем встраивания, но он действует надежно и устойчиво даже в прежних версиях браузеров. А другой способ состоит в применении стандарта консорциума W3C для регистрации событий. Далее будут рассмотрены оба способа, поскольку в вашей практике, вероятнее всего, встретится и тот и другой.

Традиционная привязка событий

Традиционная привязка событий является простейшим способом привязки обработчиков событий. В этом способе выгодно используется то обстоятельство, что обработчики событий являются свойствами элементов в модели DOM. Чтобы воспользоваться этим способом, достаточно присоединить функцию в качестве свойства к элементу DOM, за поведением которого требуется наблюдать. Нужный элемент извлекается с помощью метода `document.getElementById()` или любых других функций извлечения элементов, обсуждавшихся в главе 5. Допустим, что требуется наблюдать за событиями от щелчка. Для этого достаточно присвоить соответствующую функцию свойству `onclick` извлеченного элемента, и дело с концом!

В примерах из этой главы применяется стандартная HTML-страница со многими размеченными элементами. Содержимое этой страницы приведено в листинге 6.1.

Листинг 6.1. Пример разметки HTML-страницы, применяемой для демонстрации обработки событий

```
<!DOCTYPE html>
<html>
<head lang="en">
  <meta charset="UTF-8">
  <title>Event Handling</title>
  <link rel="stylesheet" href="school.css"/>
</head>
<body>
```

```
<div id="main">
  <nav id="navbar">
    <ul>
      <li>Students
        <ul>
          <li id="Academics">Academics</li>
          <li id="Athletics">Athletics</li>
          <li id="Extracurriculars">Extracurriculars</li>
        </ul>
      </li>
      <li>Faculty
        <ul>
          <li id="Frank Walsh">Frank Walsh</li>
          <li id="Diane Walsh">Diane Walsh</li>
          <li id="John Mullin">John Mullin</li>
          <li id="Lou Garaventa">Lou Garaventa</li>
          <li id="Dan Tully">Dan Tully</li>
          <li id="Emily Su">Emily Su</li>
        </ul>
      </li>
    </ul>
  </nav>
  <div id="welcome">
    <h1>Welcome to the School of JavaScript</h1>
    <h3 id="welcome-header">Click here for a welcome message!</h3>
    <p id="welcome-content">Welcome to the School of JavaScript.
          Here, you will find many
      <a href="/examples" id="examples-link">examples</a>
              of JavaScript, taught by our most esteemed
      <a href="/faculty">faculty</a>.
      <span id="disclaimer">Please note that these are only examples,
              and are not necessarily
      <a href="/production-ready">production-ready code</a>.</span></p>
  </div>
  <hr/>
  <div id="form-container">
    <h2>Contact Form</h2>

    <p>Thank you for your interest in the School of JavaScript.
       Please fill out the form below so we can send you even
       more materials!</p>

    <form id="main-form">
      <ul>
        <li><label for="firstName">First Name:
            </label><input id="firstName" type="text"/></li>
        <li><label for="lastName">Last Name:
            </label><input id="lastName" type="text"/></li>
        <li><label for="city">City:
```

```
            </label><input id="city" type="text"/></li>
        <li><label for="state">State:
            </label><input id="state" type="text"/></li>
        <li><label for="postCode">Postal Code:
            </label><input id="postCode" type="text"/></li>
        <li><label for="comments">Comments: </label><br/>
          <textarea name="" id="comments" cols="30"
                    rows="10"></textarea>
        </li>
        <li><input type="submit"/> <input type="reset"/></li>
      </ul>
    </form>
  </div>
</div>

</body>
</html>
```

Как видите, на главной странице имеется немало элементов для представления воображаемой школы программирования на JavaScript. Навигационная панель в конечном итоге будет снабжена подходящими средствами обработки событий, чтобы функционировать как меню. Эти средства будут введены в форму для простой проверки достоверности вводимых данных (более сложные средства проверки достоверности вводимых данных рассматриваются в главе 8), а для большей интерактивности данная страница будет дополнена приветственным сообщением.

Итак, начнем с самого простого. Когда производится щелчок на поле `firstName`, зарегистрируем это событие на консоли, а затем выделим данный элемент желтым цветом фона. Привяжем это событие рассматриваемым здесь традиционным способом, как показано в листинге 6.2.

Листинг 6.2. Привязка события от щелчка традиционным способом

```
// Извлечь элемент firstName
var firstName = document.getElementById('firstName');

// Присоединить к нему обработчик событий
firstName.onclick = function() {
  console.log('You clicked in the first name field!');
  firstName.style.background = 'yellow';
};
```

Самое замечательное, что этот код работает! Но ему недостает гибкости. Поэтому нам придется написать отдельную функцию обработки событий для каждого поля формы. Но ведь это потребует немало труда! Нельзя ли найти какой-нибудь способ получить ссылку на элемент, инициировавший событие?

В действительности таких способов два! Первый и самый простой состоит в том, чтобы предоставить аргумент функции обработки событий, как демонстрируется в листинге 6.3. В качестве этого аргумента служит объект события, содержащий сведения о только что наступившем событии. Более подробно объект события будет рассматриваться несколько позже. А до тех пор достаточно сказать, что свойство `target` этого объекта содержит ссылку на элемент DOM, инициировавший событие.

Листинг 6.3. Привязка события по аргументу

```
// Извлечь элемент firstName
var firstName = document.getElementById('firstName');

// Присоединить обработчик событий
firstName.onclick = function(e) {
  console.log('You clicked in the ' + e.target.id + ' field!');
  e.target.style.background = 'yellow';
};
```

Свойство `e.target` указывает на поле `firstName`, а по существу, на элемент DOM для этого поля, и поэтому можно проверить его свойство `id`, чтобы выяснить, на каком именно поле был произведен щелчок. Но еще важнее, что можно изменить и его свойство `style`! Это означает, что действие данного обработчика событий можно распространить практически на любые текстовые поля в форме.

Вместо явного обращения к объекту события можно также воспользоваться ключевым словом `this`, как демонстрируется в примере функции из листинга 6.4. В контексте функции обработки событий это означает ссылку на инициатор события. Иными словами, ссылки `event.target` и `this` являются синонимами или, по крайней мере, указывают на одно и то же.

Листинг 6.4. Привязка события по ссылке `this`

```
// Извлечь элемент firstName
var firstName = document.getElementById('firstName');

// Присоединить обработчик событий
firstName.onclick = function() {
  console.log('You clicked in the ' + this.id + ' field!');
  this.style.background = 'yellow';
};
```

Какой же способ привязки событий предпочесть? Объект события представляет все необходимые сведения о событии, тогда как ссылка `this` лишь указывает на элемент DOM, инициировавший событие. По затратам ресурсов ни один из этих способов не имеет никаких преимуществ, и поэтому, как правило, рекомендуется пользоваться объектом события, чтобы сразу узнать во всех подробностях о наступившем событии.

Но иногда и ссылка `this` оказывается удобной. Она всегда указывает на ближайший элемент, инициировавший событие. Вернемся к разметке страницы в листинге 6.1 и рассмотрим элемент разметки `<div>` с атрибутом `id = "welcome"`. Допустим, что он дополнен обработчиком событий от мыши для изменения цвета фона при наведении курсора мыши на этот элемент и возврата к исходному цвету фона при отведении курсора мыши. С одной стороны, если изменить стиль оформления этого элемента по ссылке `e.target`, то событие будет наступать и для каждого подчиненного элемента разметки (`welcome-header`, `welcome-content` и т.д.)! А с другой стороны, если изменить стиль оформления элемента разметки `<div>` с атрибутом `id = "welcome"` по ссылке `this`, это изменение коснется только его. Мы рассмотрим данное отличие более подробно при обсуждении особенностей делегирования событий далее в этой главе.

Преимущества традиционной привязки событий

Традиционной привязке событий присущи следующие преимущества.

- Главное преимущество традиционного способа привязки событий заключается в его простоте и согласованности. Этим гарантируется, что он будет действовать одинаково в любом браузере.
- При обработке события ключевое слово `this` ссылается на текущий элемент разметки, что может быть очень удобно, как показано в листинге 6.4.

Недостатки традиционной привязки событий

Но у традиционной привязки событий имеются и недостатки.

- Традиционный способ привязки не позволяет контролировать перехват или всплывание событий. Все события всплывают, а изменить порядок их перехвата нельзя.
- Обработчики событий можно привязывать к элементам разметки только по очереди. Это может привести к неясным результатам при обращении к распространенному свойству `window.onload`. По существу, это приводит к перезаписи других фрагментов кода, где применяется тот же самый способ привязки событий. Пример такого затруднения приведен в листинге 6.5, где следующий обработчик событий перезаписывает предыдущий обработчик событий.

Листинг 6.5. Обработчики событий, перезаписывающие друг друга

```
// Привязать первый обработчик событий
window.onload = myFirstHandler;

// Первый обработчик событий перезаписывается где-то
// в другой подключаемой библиотеке, а по завершении загрузки
// вызывается только второй обработчик 'mySecondHandler'
window.onload = mySecondHandler;
```

- В браузере Internet Explorer 8 и прежних его версиях нельзя передать объект события в качестве аргумента функции. Вместо этого приходится пользоваться свойством `window.event`.

Зная, что одни обработчики событий могут быть слепо перезаписаны другими, пользоваться традиционными средствами привязки событий рекомендуется только в простых случаях, когда можно вполне доверять другому коду, выполняющемуся вместе со своим кодом. Но подобных хлопот можно избежать, воспользовавшись способом привязки событий по стандарту консорциума W3C, внедренному в современных браузерах.

Привязка событий к элементам DOM по стандарту консорциума W3C

Стандартные средства привязки событий к элементам DOM обеспечивает только способ, предложенный консорциумом W3C. Такой способ привязки событий поддерживается во всех современных браузерах, в том числе и Internet Explorer, начиная с версии 9. А в прежних версиях этого браузера он не поддерживается, хотя они

и не считаются современными. Поэтому при разработке веб-приложений для этих версий следует рассмотреть возможность применения традиционного способа привязки событий.

Код для привязки функции обработки событий этим новым способом довольно прост. Он представлен в виде функции, доступной в каждом элементе DOM. Эта функция называется `addEventListener()` и принимает следующие три аргумента: имя события (например, `click`; обратите внимание на отсутствие префикса в этом имени), функцию обработки события, а также логический признак для разрешения или запрета перехвата событий. Характерный пример применения функции `addEventListener()` приведен в листинге 6.6.

Листинг 6.6. Пример кода, в котором применяется способ привязки событий по стандарту консорциума W3C

```
// Извлечь элемент firstName
var firstName = document.getElementById( 'firstName' );

// Присоединить к нему обработчик событий
firstName.addEventListener( 'click', function ( e ) {
  console.log( 'You clicked in the ' + e.target.id + ' field!' );
  e.target.style.background = 'yellow';
} );
```

Следует заметить, что в данном примере функции `addEventListener()` не передается третий аргумент. По умолчанию третий ее аргумент принимает логическое значение `false`, а это означает, что вместо перехвата применяется всплывание событий. Если бы потребовался перехват событий, то в качестве третьего аргумента данной функции следовало бы явным образом передать логическое значение `true`.

Преимущества привязки по стандарту консорциума W3C

Способу привязки событий стандарту консорциума W3C присущи следующие преимущества.

- Этот способ поддерживает обе стадии обработки событий: перехват и всплывание. Конкретная стадия выбирается установкой соответствующего логического значения в последнем параметре функции `addEventListener()`: `false` (всплывание по умолчанию) или `true` (перехват).
- В функции обработки событий ключевое слово `this` служит для ссылки на текущий элемент разметки, как и при традиционной привязке событий.
- Объект события всегда доступен в качестве первого аргумента функции обработки событий.
- К элементу разметки можно привязать сколько угодно событий, не перезаписывая привязанные ранее обработчики событий. Все привязываемые обработчики событий размещаются во внутреннем стеке JavaScript и выполняются в том порядке, в каком они были зарегистрированы.

Недостатки привязки по стандарту консорциума W3C

У способа привязки событий стандарту консорциума W3C имеется лишь один недостаток.

- Он не поддерживается в браузере Internet Explorer 8 и прежних его версиях, где для этой цели применяется функция `attachEvent()` с аналогичным синтаксисом.

Отвязка событий

Итак, мы рассмотрели способы привязки событий. А что, если потребуется отвязать события? Например, в том случае, если недоступна кнопка, к которой привязан обработчик событий от щелчка, или же если элемент разметки `<div>` больше не требуется выделять при наведении на него курсора. Отвязать событие и его обработчик относительно просто. Если событие привязано традиционным способом, то достаточно присвоить обработчику события пустую символьную строку или пустое значение, как показано ниже.

```
document.getElementById('welcome-content').onclick = null;
```

Как видите, сделать это совсем не трудно. Но дело несколько усложняется, если событие привязано по стандарту консорциума W3C. Для этой цели служит функция `removeEventListener()`, принимающая те же самые три аргумента: тип удаляемого события, связанный с ним обработчик и логическое значение `true` или `false` для режима перехвата или всплывания событий соответственно. Но особенность отвязки событий в данном случае состоит в том, что функция `removeEventListener()` должна ссылаться на ту же самую функцию обработки событий, которая была назначена с помощью функции `addEventListener()`, а не на те же самые строки кода. Так, если с помощью функции `addEventListener()` была назначена анонимная встраиваемая функция, то она не может быть удалена подобным способом.

Совет Если в дальнейшем предполагается удаление обработчика событий, то в качестве такого обработчика следует всегда пользоваться именованной функцией.

Аналогично, если указать третий аргумент при первоначальном вызове функции `addEventListener()`, то его придется снова задать и при вызове функции `removeEventListener()`. Если же не сделать этого или передать функции `removeEventListener()` неверное значение данного аргумента, ее выполнение негласно завершится неудачно. В листинге 6.7 приведен пример отвязки обработчика событий подобным способом.

Листинг 6.7. Отвязка обработчика событий

```
// Допустим, что имеются две кнопки, 'foo' и 'bar'
var foo = document.getElementById( 'foo' );
var bar = document.getElementById( 'bar' );

// Когда производится щелчок на кнопке foo, на консоль
// требуется вывести сообщение "Clicked on foo!"
function fooHandler() {
  console.log( 'Clicked on the foo button!' );
}

foo.addEventListener( 'click', fooHandler );
```

```
// Когда же производится щелчок на кнопке bar, требуется
// удалить обработчик событий для кнопки foo и вывести
// на консоль соответствующее сообщение
function barHandler() {
  console.log( 'Removing event handler for foo....' );
  foo.removeEventListener( 'click', fooHandler );
}

bar.addEventListener( 'click', barHandler );
```

Типичные средства обработки событий

Для обработки событий в JavaScript имеется целый ряд относительно согласованных средств, предоставляющих больше возможностей управлять данным процессом в ходе разработки веб-приложений. Самым простым и старым средством служит объект события, предоставляющий ряд метаданных и зависящий от контекста функции для обработки таких событий, как, например, при перемещении мыши или нажатии клавиш. Кроме того, имеются функции для видоизменения обычного хода перехвата или всплывания событий. Знание особенностей этих средств обработки событий намного упрощает разработку веб-приложений.

Объект события

Одним из стандартных средств обработки событий служит доступ к объекту события, содержащему зависящую от контекста информацию о текущем событии. Этот объект служит весьма ценным ресурсом для определенных событий. Например, при обработке событий от нажатия клавиш можно обратиться к свойству `keyCode` объекта события, чтобы получить сведения о конкретной нажатой клавише. У объектов событий имеются некоторые отличия, обсуждаемые далее в этой главе. А до тех пор рассмотрим следующие вопросы: распространение событий и поведение по умолчанию.

Отмена всплывания событий

Выше пояснялось, каким образом происходит перехват или всплывание событий. А теперь выясним, как взять этот процесс под контроль. Из предыдущего примера можно сделать следующий важный вывод: если требуется, чтобы событие произошло только в целевом элементе, а не в его родительских элементах, то остановить его распространение никак нельзя. Ведь если остановить ход всплывания события, то произойдет нечто, подобное показанному на рис. 6.2, где результат наступления события перехватывается в первом элементе разметки `<a>`, а дальнейшее его всплывание отменяется.

Прекращение всплывания (или перехвата) события может оказаться очень удобным в сложных приложениях, а реализовать его совсем не трудно. Достаточно вызвать метод `stopPropagation()` для объекта события, чтобы предотвратить дальнейшее распространение события вверх (или вниз) по иерархии. Характерный тому пример приведен в листинге 6.8.

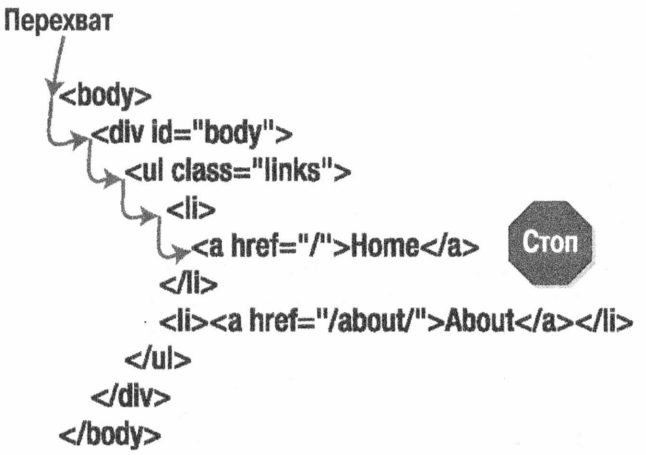


Рис. 6.2. Результат наступления события перехватывается в первом элементе разметки `<a>`

Листинг 6.8. Пример, демонстрирующий прекращение всплывания события

```
document.getElementById( 'disclaimer' ).addEventListener(
        'click', function ( e ) {
  // Выделить полужирным отказ от права, когда
  // на нем производится щелчок
  e.target.style.fontWeight = 'bold';

  // Родительский элемент стремится скрыться, когда производится
  // щелчок на текущем элементе. Мы должны воспрепятствовать этому
  e.stopPropagation();
} );

document.getElementById( 'welcome-content' ).addEventListener(
        'click', function ( e ) {
  e.target.style.visibility = 'hidden';
} );
```

В листинге 6.9 приведен краткий фрагмент кода для ввода красного обрамления вокруг элемента, на который наведен курсор. С этой целью в каждый элемент DOM вводятся обработчики событий `mouseover` и `mouseout`. Если не остановить всплывание событий, то при наведении курсора на элемент красное обрамление появится вокруг данного элемента и всех его родительских элементов, что нежелательно.

Листинг 6.9. Недопущение изменения цвета всех элементов с помощью метода `stopPropagation()`

```
// Функции обработки событий
function mouseOverHandler( e ) {
  e.target.style.border = '1px solid red';
  e.stopPropagation();
}
```

```
function mouseOutHandler( e ) {
  this.style.border = '0px';
  e.stopPropagation();
}

// Найти и обойти все элементы DOM
var all = document.getElementsByTagName( '*' );
for ( var i = 0; i < all.length; i++ ) {

  // Проследить за тем, когда пользователь наведет курсор на элемент,
  // а затем ввести красное обрамление вокруг этого элемента
  all[i].addEventListener( 'mouseover', mouseOverHandler );

  // Проследить за тем, когда пользователь отведет курсор от элемента,
  // а затем удалить введенное ранее красное обрамление
  all[i].addEventListener( 'mouseout', mouseOutHandler );
}
```

Возможность останавливать всплывание событий дает полный контроль над теми элементами, где можно обнаружить и обработать событие. Это основополагающее средство обработки событий необходимо для исследования путей разработки динамических веб-приложений. В конечном итоге требуется отменить действие, выполняемое в браузере по умолчанию, полностью заменив его реализацией новых функциональных возможностей.

Отмена действия, выполняемого в браузере по умолчанию

В ответ на большинство происходящих событий по умолчанию в браузере всегда выполняется некоторое действие. Так, если щелкнуть на элементе разметки `<a>`, произойдет переход на связанную с ним веб-страницу. Это и есть действие, выполняемое в браузере по умолчанию. Оно всегда происходит после стадий перехвата и всплывания событий. Так, на рис. 6.3 показаны результаты щелчка, произведенного

Рис. 6.3. Весь срок действия события

пользователем на элементе разметки <a> веб-страницы. Событие начинает распространяться по модели DOM на обеих стадиях перехвата и всплывания, как пояснялось ранее. Но как только распространение события прекратится, браузер попытается выполнить действие по умолчанию для этого события и элемента. В данном случае это переход на начальную веб-страницу.

Действия по умолчанию можно подытожить как все, что браузер делает из того, что ему не предписывается делать явным образом. Ниже перечислены разные виды действий, происходящих по умолчанию, а также связанные с ними события.

- В результате щелчка на элементе разметки <a> происходит переадресация по URL, предоставляемому в его атрибуте `href`.
- При нажатии комбинации клавиш <Ctrl+S> браузер попытается сохранить физическое представление веб-сайта.
- Передача формы, размеченной элементом <form>, вместе с данными запроса по указанному URL и переадресация браузера в данное место.
- Наведение курсора на элемент разметки <img> с атрибутом `alt` или `title` в зависимости от конкретного браузера приводит к появлению всплывающей подсказки, предоставляющей значение атрибута.

Все перечисленные выше действия выполняются браузером даже в том случае, если остановить всплывание событий, или же в том случае, если ни один из обработчиков событий не привязан. Это может привести к серьезным затруднениям в сценариях на JavaScript. Что, если, например, от передаваемых форм или от элементов разметки <a> потребуется поведение, отличающееся от предписанного? Отмены всплывания событий недостаточно, чтобы предотвратить действие по умолчанию, и поэтому для непосредственного управления данным процессом потребуется особый код. Такие функциональные возможности предоставляет метод `preventDefault()`, вызываемый для объекта события из прикладного программного интерфейса API, предложенного консорциумом W3C для обработки событий, как показано в листинге 6.10. В качестве альтернативы во многих браузерах может быть выбран простой возврат логического значения `false` из обработчика событий. Реализацию именно такого поведения можно наблюдать в некоторых примерах кода и библиотеках. Но предпочтение все же следует отдавать методу `preventDefault()`, поскольку он самодокументирован, в отличие от не совсем ясного приема периодического возврата логического значения `false` из обработчика событий.

Листинг 6.10. Обобщенная функция, предотвращающая выполнение в браузере действия по умолчанию

```
document.getElementById('examples-link').addEventListener(
        'click', function(e) {
  e.preventDefault();
  console.log("examples-link clicked");
});
```

Используя метод `preventDefault()`, можно теперь остановить выполнение в браузере любого действия по умолчанию. Это, например, дает возможность выгодно воспользоваться преимуществами событий `mouseover` для правильного перехода по ссылке, не беспокоясь о том, что пользователь может случайно щелкнуть на ссылке и направить браузер не туда, куда нужно. С одной стороны, можно отменить поведе-

ние по умолчанию, показав в строке состояния, куда именно направляет ссылка, а с другой стороны — предоставить кнопку Submit (Передать), чтобы начать проверку достоверности формы. И если форма не пройдет проверку достоверности, то можно отказаться от ее передачи на рассмотрение (т.е. изменить поведение по умолчанию).

Делегирование событий

Итак, мы рассмотрели почти все средства для манипулирования обработчиками событий, теперь осталось лишь выбрать подходящую методику их применения. Допустим, что имеется неупорядоченный список, состоящий из 20 элементов, каждый из которых требуется снабдить обработчиком событий. Точнее говоря, требуется по-разному обрабатывать события от щелчков на отдельных элементах списка. С этой целью можно было бы извлечь все элементы из списка с помощью метода `document.querySelectorAll()`, обойти в цикле полученные элементы и присоединить к ним обработчики отдельных событий. Но это было бы неэффективно как в самом процессе, так и в браузере. Ведь нам пришлось бы устанавливать 20 обработчиков событий по очереди, когда это можно было бы сделать сразу, даже если бы они указывали на одну и ту же функцию обработки событий.

Все элементы списка содержатся в дескрипторе неупорядоченного списка, так почему бы не воспользоваться с выгодой тем обстоятельством, что события от щелчка можно перехватывать на уровне элемента разметки `<ul>`? Нужно лишь каким-то образом различать элементы списка. Ранее при обсуждении ссылки `this` в разделе, посвященном традиционной привязке событий, было замечено, что она делается на тот элемент разметки, в котором перехватывается событие, тогда как ссылка `event.target` делается на тот элемент разметки, который фактически инициирует данное событие. Очевидно, что мы могли бы воспользоваться ссылками `this` и `event.target` в определенном сочетании, но для этой цели в спецификации на обработку событий предоставляется специальное свойство `event.currentTarget`.

В рассматриваемом здесь примере обработчик событий от щелчка присоединяется к неупорядоченному списку, в самом обработчике событий целевой элемент разметки `<ul>` указывается в свойстве `event.currentTarget`, а каждый элемент списка — в свойстве `event.target`. Следовательно, мы можем проверить содержимое свойства `event.target`, чтобы выяснить, на каком именно элементе списка был произведен щелчок, чтобы передать его соответствующей функции на обработку. Характерный пример делегирования событий демонстрируется в листинге 6.11, где функция `clickHandler()` обрабатывает события на уровне элемента разметки `<nav>`, но получает события, инициируемые различными элементами списка, находящимися ниже элемента разметки `<nav>` по иерархии.

Листинг 6.11. Делегирование событий

```
function clickHandler(e) {
  console.log( 'Handled at ' + e.currentTarget.id );
  console.log( 'Emitted by ' + e.target.id );
}

var navbar = document.getElementById('navbar');
navbar.addEventListener( 'click', clickHandler );
```

Объект события

Объект события предоставляется или доступен в теле каждой функции обработки событий. В общем, свойства объекта события содержат все подробности, которые требуется знать о событии: его типе, происхождении, координатах места, где был произведен щелчок, или нажатых клавишах. Но у разных браузеров имеются некоторые отличия в передаче этой информации.

Общие свойства

У объекта события имеется определенное количество свойств для каждого перехватываемого события. Все эти свойства связаны непосредственно с самим событием, но никак — с конкретным браузером. Далее по очереди рассматриваются все свойства и методы объекта события с краткими пояснениями и примерами кода, демонстрирующими их применение.

Свойство `type`

Содержит наименование наступающего события, например `click` или `mouseover`. Оно может служить для предоставления обобщенной функции обработки событий, которая затем детерминированно выполняет соответствующий код. В листинге 6.12 демонстрируется пример применения свойства `type` для обработки события в зависимости от его типа.

Листинг 6.12. Применение свойства `type` для обработки событий при наведении курсора на элемент

```
function mouseHandler(e){
  // Смена цвета фона в элементе разметки <div> в зависимости
  // от типа наступившего события от мыши
  this.style.background = (e.type === 'mouseover') ? '#EEE' : '#FFF';
}

// Найти элемент разметки <div>, на который нужно навести курсор мыши
var div = document.getElementById('welcome');

// Привязать одну и ту же функцию к обоим событиям, mouseover и mouseout
div.addEventListener( 'mouseover', mouseHandler );
div.addEventListener( 'mouseout', mouseHandler );
```

Свойство `target`

Содержит ссылку на элемент, инициировавший событие. Например, привязка обработчика событий от щелчка к элементу разметки `<a>` приведет к появлению свойства `target`, равнозначного самому элементу разметки `<a>`.

Метод `stopPropagation()`

Останавливает процесс всплывания (или перехвата) событий. В итоге текущий элемент разметки становится последним элементом, получившим конкретное событие.

Метод preventDefault()

В результате вызова этого метода во всех современных браузерах, совместимых со стандартами консорциума W3C, предотвращается выполнение действия по умолчанию. Аналогичного результата можно добиться, просто возвратив логическое значение false из обработчика событий.

Свойства мыши

Свойства мыши присутствуют в объекте события только в том случае, если инициируется событие от мыши (например, click, mousedown, mouseup, mouseover, mousemove, mouseout, mouseenter или mouseleave). Но в остальное время можно считать, что значения, возвращаемые этими свойствами, не существуют, а если и существуют, то ненадежны. Ниже рассматриваются по очереди свойства объекта события, существующие в течение события от мыши.

Свойства pageX и pageY

Содержат координаты *x* и *y* положения курсора мыши относительно начала координат в левом верхнем углу окна браузера. Они не изменяются во время прокрутки.

Свойства clientX и clientY

Содержат координаты *x* и *y* положения курсора мыши относительно окна браузера. Так, если документ прокручивается по вертикали (или горизонтали), числовые значения этих свойств указывают на положение курсора относительно краев окна браузера. Эти значения изменяются по ходу прокрутки документа.

Свойства layerX/layerY и offsetX/offsetY

Должны содержать координаты *x* и *y* положения курсора мыши относительно целевого элемента для наступившего события. Свойства offsetX/offsetY поддерживаются в браузерах Chrome и Internet Explorer, но не в Firefox, где поддерживаются свойства layerX/layerY, хотя они содержат другие сведения. Эти свойства в большей степени соответствуют свойствам pageX/pageY.

Свойство button

Доступно только при наступлении событий click, mousedown и mouseup, а его числовое значение представляет кнопку мыши, на которой в настоящий момент произведен щелчок. В частности, значение 0 обозначает щелчок левой кнопкой мыши; значение 1 — щелчок средней кнопкой мыши; значение 2 — щелчок правой кнопкой мыши.

Свойство relatedTarget

Содержит ссылку на элемент разметки, с которого только что был перемещен курсор мыши. Чаще всего свойство relatedTarget применяется в тех случаях, когда требуется навести или отвести курсор от элемента разметки, но, кроме того, нужно знать, где он только что находился или куда будет перемещен. В листинге 6.13 приведен пример древовидного меню, где одни элементы разметки <ol> содержат другие аналогичные элементы, а поддеревья отображаются только при наведении курсора мыши на подчиненный элемент <li> в первый раз.

Листинг 6.13. Применение свойства `relatedTarget` при построении доступного для перемещения дерева

```
// Если событие DOMContent наступило, получить ссылки на элементы
document.addEventListener('DOMContentLoaded', init);

function init(){
  var top = document.getElementById("top");
  var bottom = document.getElementById("bottom");

  top.addEventListener("mouseover", onMouseOver);
  top.addEventListener("mouseout", onMouseOut);

  bottom.addEventListener("mouseover", onMouseOver);
  bottom.addEventListener("mouseout", onMouseOut);
}

function onMouseOut(event) {
console.log("exited " + event.target.id + " for " + event.relatedTarget.id);
}
function onMouseOver(event) {
  console.log("entered " + event.target.id + " from " +
              event.relatedTarget.id);
}

// Образец HTML-разметки документа:
<style>
div > div {
  height: 128px;
  width: 128px;
}
#top { background-color: red; }
#bottom { background-color: blue; }
</style>
<title>Untitled Document</title>
</head>

<body>

<div id="outer">
  <div id="top"></div>
  <div id="bottom"></div>
</div>
```

Свойства клавиатуры

Свойства клавиатуры обычно существуют в объекте события, когда инициируется событие от клавиатуры (например, `keydown`, `keyup` или `keypress`). Исключением из этого правила являются свойства `ctrlKey` и `shiftKey`, доступные и в течение событий от мыши, когда пользователь может нажать клавишу <Ctrl> и щелкнуть кноп-

кой мыши на элементе разметки. Но в остальное время можно считать, что значения, возвращаемые этими свойствами, не существуют, а если и существуют, то ненадежны.

Свойство `ctrlKey`

Возвращает логическое значение, представляющее нажатие клавиши <Ctrl>. Свойство `ctrlKey` доступно как для событий от клавиатуры, так и для событий от мыши.

Свойство `keyCode`

Содержит числовое значение, соответствующее разным клавишам на клавиатуре. Доступность определенных клавиш (например, <PageUp> или <Home>) может разниться, но все остальные клавиши, как правило, распознаются надежно. Все наиболее употребительные клавиши и их коды перечислены в табл. 6.1.

Таблица 6.1. Наиболее употребительные клавиши и их коды

Клавиша	Код клавиши	Клавиша	Код клавиши	Клавиша	Код клавиши
<Backspace>	8	<←>	37	0-9	48-57
<Tab>	9	<↑>	38	A-Z	65-90
<Enter>	13	<→>	39		
<Пробел>	32	<↓>	40		

Свойство `shiftKey`

Возвращает логическое значение, представляющее нажатие клавиши <Shift>. Свойство `shiftKey` доступно как для событий от клавиатуры, так и для событий от мыши.

Типы событий

Типичные для JavaScript события могут быть разделены на несколько категорий. Чаще всего употребляются события, связанные с операциями мышью, затем события от клавиатуры и далее события, наступающие в заполняемой форме. Ниже дается краткий обзор различных категорий событий, которые наступают и могут быть обработаны в веб-приложении.

- **События при загрузке и ошибках.** События этой категории связаны с самой страницей и отражают состояние ее загрузки. Они происходят, когда пользователь загружает страницу в первый раз (событие `load`) и когда он покидает страницу (события `unload` и `beforeunload`). Кроме того, ошибки в коде JavaScript отслеживаются с помощью события `error`, что дает возможность обрабатывать ошибки по отдельности.
- **События в пользовательском интерфейсе.** События этой категории служат для отслеживания моментов взаимодействия пользователей со страницей, когда они переходят от одного ее элемента к другому. С их помощью можно, например, надежно выяснить, когда пользователь начал вводить данные в

элемент формы. Для отслеживания этого момента предназначены события `focus` и `blur`. Они определяют, когда пользователь получает и когда теряет фокус ввода соответственно.

- **События от мыши.** Эти события разделяются на следующие категории: события, отслеживающие текущее местоположение курсора мыши (`mouseover`, `mouseout`), а также события, отслеживающие место, где был произведен щелчок кнопкой мыши (`mouseup`, `mousedown`, `click`).
- **События от клавиатуры.** Эта категория событий отвечает за отслеживание нажатий клавиш в определенном контексте, например, отслеживание нажатий клавиш в элементах формы, в отличие от нажатий клавиш на странице в целом. Как и для мыши, для отслеживания состояний клавиш на клавиатуре служат следующие события: `keyup`, `keydown` и `keypress`.
- **События в форме.** События этой категории связаны со взаимодействиями, которые происходят только в формах и их элементах для ввода данных. В частности, событие `submit` служит для отслеживания момента передачи формы, событие `change` — для наблюдения за вводом пользовательских данных в элементе, а событие `select` наступает в результате обновления элемента разметки `<select>`.

События на странице

Все события на странице имеют непосредственное отношение к функционированию и состоянию всей страницы в целом. Большинство типов событий данной категории наступают при загрузке и выгрузке страницы, т.е. всякий раз, когда пользователь посещает страницу и затем покидает ее.

Событие `load`

Наступает, как только страница полностью загрузится, включая все изображения, внешние файлы JavaScript и CSS. Оно доступно также в большинстве элементов разметки с атрибутом `src`, принимающим значение `img`, `script`, `audio`, `video` и т.д. Событие `load` не всплывает.

Событие `beforeunload`

Это не совсем обычное событие, поскольку оно нестандартное, хотя и широко поддерживается. Оно ведет себя аналогично событию `unload`, но за одним существенным исключением. Если обработчик событий `beforeunload` возвращает символьную строку, то в ней присутствует сообщение о подтверждении, запрашивающее у пользователей, желают ли они покинуть текущую страницу. Если они отклонят такое предложение, то могут остаться на текущей странице. В таких динамических веб-приложениях, как Gmail, это обстоятельство используется с целью предотвратить потерю пользователями любых не сохраненных данных.

Событие `error`

Инициируется всякий раз, когда в коде JavaScript возникает ошибка. Оно может служить в качестве средства для перехвата сообщений об ошибках, их отображения или корректной обработки. Обработчик подобных событий ведет себя иначе, чем

обработчики других событий, в том отношении, что вместо объекта события ему передается сообщение, поясняющее характер возникшей ошибки.

Событие `resize`

Наступает всякий раз, когда пользователь изменяет размер окна браузера. Когда пользователь корректирует размер окна браузера, событие `resize` наступит, как только процесс изменения размера завершится полностью, а не частично.

Событие `scroll`

Наступает, когда пользователь перемещается по документу в окне браузера. Оно может наступить при нажатии клавиши (например, клавиш со стрелками, <Page Up>, <Page Down> или клавиши пробела) или при прокрутке документа.

Событие `unload`

Наступает всякий раз, когда пользователь покидает текущую страницу (например, щелкнув на ссылке, нажав клавишу Back (Назад) или даже закрыв окно браузера). Но это событие не позволяет предотвратить выполнение действия по умолчанию, для чего лучше инициировать событие `beforeunload`.

События в пользовательском интерфейсе

События в пользовательском интерфейсе имеют отношение к взаимодействию пользователя с браузером или элементами страницы. Такие события помогают определить те элементы на странице, с которыми пользователь взаимодействует в настоящий момент, а также представить для них дополнительный контекст (например, выделение или вспомогательные меню).

Событие `focus`

Служит для определения места на странице, где в настоящий момент находится курсор. По умолчанию фокус ввода находится в пределах всего документа, но всякий раз, когда ссылка или элемент ввода данных в форме выбирается с помощью мыши или клавиатуры, именно туда и перемещается фокус ввода. (Пример обработки события `focus` приведен в листинге 6.14.)

Событие `blur`

Наступает, когда пользователь перемещает фокус ввода от одного элемента к другому (в контексте ссылок, элементов ввода данных или самой страницы). (Пример обработки события `blur` приведен в листинге 6.14.)

События от мыши

Наступают, когда пользователь перемещает курсор или производит щелчок кнопками мыши.

Событие `click`

Наступает, когда пользователь производит щелчок левой кнопкой мыши на элементе (см. далее описание события `mousedown`) и отпускает кнопку мыши (см. далее описание события `mouseup`), удерживая курсор на том же самом элементе.

Событие dblclick

Наступает, когда пользователь совершает двойной щелчок, т.е. когда быстро следуют подряд два события `click`. Скорость двойного щелчка зависит от конкретных настроек на уровне операционной системы.

Событие mousedown

Наступает, когда пользователь нажимает кнопку мыши. В отличие от события `keydown`, это событие инициируется лишь один раз, когда нажимается кнопка мыши.

Событие mouseup

Наступает, когда пользователь отпускает кнопку мыши. Если кнопка мыши отпускается на том же самом элементе, где она была нажата, то наступает также событие `click`.

Событие mousemove

Наступает, когда пользователь перемещает указатель мыши (курсор) хотя бы на один пиксель на экране. Количество инициируемых событий `mousemove` для полного перемещения мыши зависит от того, насколько быстро пользователь двигает мышью и как часто браузер успевает обновлять положение курсора.

Событие mouseover

Наступает, когда пользователь наводит курсор на один элемент, перемещая его от другого элемента. Чтобы выяснить, от какого именно элемента пользователь переместил курсор, следует обратиться к свойству `relatedTarget`. Это событие является ресурсоемким, поскольку оно может инициироваться при перемещении по экрану на каждый пиксель или наведении курсора на каждый подчиненный элемент. Поэтому вместо него следует предпочесть рассматриваемое далее событие `mouseenter`.

Событие mouseout

Наступает, когда пользователь перемещает курсор за пределы элемента, включая перемещение курсора от родительского элемента к порожденному элементу, что может показаться не совсем логичным. Чтобы выяснить, к какому именно элементу пользователь перемещает курсор, следует обратиться к свойству `relatedTarget`. Это событие является ресурсоемким, поскольку оно может неоднократно инициироваться вместе с событием `mouseover`. Поэтому вместо него следует предпочесть рассматриваемое далее событие `mouseleave`.

Событие mouseenter

Действует аналогично событию `mouseover`, но более тонко, принимая во внимание местоположение курсора в пределах элемента. Оно не наступает до тех пор, пока курсор не покинет пределы элемента.

Событие mouseout

Действует аналогично событию mouseout, но более тонко, принимая во внимание момент, когда курсор покидает пределы элемента.

В листинге 6.14 приведен пример привязки двух событий к элементам, чтобы сделать веб-страницу доступной для взаимодействия с помощью клавиатуры и мыши. Всякий раз, когда пользователь перемещает курсор, наводя его на ссылку с помощью мыши или клавиатуры, эта ссылка выделяется другим цветом.

Листинг 6.14. Создание эффекта наведения курсора с помощью событий mouseover и mouseout

```
// Обработчик событий mouseenter
function mouseEnterHandler() {
  this.style.backgroundColor = 'blue';
}

// Обработчик событий mouseleave
function mouseLeaveHandler() {
  this.style.backgroundColor = 'white';
}

// Найти все элементы разметки <a>, чтобы
// привязать к ним обработчики событий
var a = document.getElementsByTagName('a');
for ( var i = 0; i < a.length; i++ ) {
  // Привязать обработчики событий mouseover и focus к
  // элементу разметки <a>, чтобы изменить на синий цвет его фон,
  //  когда пользователь наводит курсор на ссылку или перемещает
  // на нее фокус ввода, используя клавиатуру
  a[i].addEventListener('mouseenter', mouseEnterHandler);
  a[i].addEventListener('focus', mouseEnterHandler);

  // Привязать обработчики событий mouseout и blur к элементу
  // разметки <a>, чтобы вернуться к исходному белому цвету его фона,
  // когда пользователь отводит курсор от ссылки
  a[i].addEventListener('mouseleave', mouseLeaveHandler);
  a[i].addEventListener('blur', mouseLeaveHandler);
}
```

События от клавиатуры

События от клавиатуры инициируются во всех случаях, когда пользователь нажимает клавиши на клавиатуре, будь то в самой области ввода текста или за ее пределами.

События keydown/keypress

Событие keydown наступает первым из всех событий от клавиатуры, когда пользователь нажимает клавишу. Если пользователь продолжает удерживать клавишу нажатой, то событие keydown продолжает инициироваться. А событие keypress действует аналогично событию keydown. Оба эти события практически одинаковы, за одним исключением: если требуется предотвратить выполнение в браузере

действия по умолчанию при нажатии клавиши, это лучше сделать по событию `keypress`.

Событие `keyup`

Наступает последним из всех событий от клавиатуры (после события `keydown`). В отличие от события `keydown`, это событие инициируется лишь один раз при отпускании клавиши, поскольку отпускать клавишу долго невозможно.

События в форме

События в форме связаны в основном с элементами разметки `<form>`, `<input>`, `<select>`, `<button>` и `<textarea>`, наиболее характерными для HTML-форм.

Событие `select`

Наступает всякий раз, когда пользователь выбирает другой блок текста в области ввода данных с помощью мыши. С помощью этого события можно переопределить порядок взаимодействия пользователя с заполняемой формой.

Событие `change`

Наступает, когда значение элемента ввода, включая элементы разметки `<select>` и `<textarea>`, видоизменяется пользователем. Оно наступает только после того, как пользователь покинет элемент, и тот потеряет фокус ввода.

Событие `submit`

Наступает только в формах и только в том случае, если пользователь щелкнет на кнопке Submit (Передать) в самой форме или нажмет клавишу <Enter> или <Return>, когда курсор находится на одном из элементов ввода. Привязав обработчик событий `submit` к форме вместо привязки обработчика событий `click` к кнопке Submit, можно надежно перехватывать все попытки передать на рассмотрение форму, заполненную пользователем.

Событие `reset`

Наступает только в том случае, если пользователь щелкнет на кнопке Reset (Сброс) в заполняемой форме, в отличие от кнопки Submit, выбор которой дублируется нажатием клавиши <Enter> или <Return>.

Доступность событий для специальных возможностей

И наконец, при разработке совершенно ненавязчивых веб-приложений следует уделить особое внимание тому, чтобы события действовали даже в том случае, если мышь не используется. Благодаря этому учитываются интересы двух групп пользователей: тех, кому требуются в помощь специальные возможности (например, пользователей со слабым зрением), а также тех, кто не любит пользоваться мышью. (Попробуйте отключить однажды мышь от своего компьютера и перемещаться по веб-страницам только с помощью клавиатуры. Этот опыт станет для вас настоящим откровением.)

Чтобы сделать события в JavaScript более доступными, рассмотрите возможность привязки событий, альтернативных событиям от мыши, всякий раз, когда вы пользуетесь в своем коде событиями `click`, `mouseover` и `mouseout`. Ниже перечислены способы, позволяющие быстро и просто выйти из данного положения.

- **Событие `click`.** Одним из полезных нововведений в браузеры стало приведение в действие события `click` при нажатии клавиши <Enter>. Благодаря этому отпала необходимость предоставлять альтернативу данному событию. Следует, однако, иметь в виду, что некоторые разработчики предпочитают привязывать обработчики событий `click` к кнопкам Submit в формах, чтобы следить за тем, когда пользователь передаст заполненную форму с веб-страницы. С этой целью вместо события `click` в качестве разумной и надежной альтернативы следует привязать событие `submit` в объекте формы.
- **Событие `mouseover`.** При перемещении по веб-странице с помощью клавиатуры фактически происходит смещение фокуса ввода с одних элементов на другие. Присоединяя обработчики к событиям `mouseover` и `focus`, можно получить равнозначное решение для пользователей клавиатуры и мыши.
- **Событие `mouseout`.** Аналогично событию `focus`, служащему альтернативой событию `mouseover`, событие `blur` наступает всякий раз, когда пользователь перемещает фокус ввода из элемента. В таком случае событием `blur` можно воспользоваться для имитации события `mouseout` с помощью клавиатуры.

В действительности внедрение возможности обрабатывать события от клавиатуры в дополнение к типичным событиям от мыши не таит в себе ничего особенного. Но помимо прочего, это помогает зависящим от клавиатуры пользователям удобнее пользоваться веб-сайтом, от чего в выигрыше остаются все.

Резюме

В начале этой главы было разъяснено, каким образом события действуют в JavaScript по сравнению с моделями событий в других языках программирования. Затем в ней была рассмотрена информация, которую предоставляет модель событий, а также показано, как лучше всего управлять ею. Далее были представлены способы привязки событий к элементам DOM и различные типы доступных событий. И в завершение главы обсуждались свойства объекта события, типы событий, а также способы обработки событий для внедрения специальных возможностей в веб-приложения.

ГЛАВА 7

JavaScript и проверка достоверности форм

Имея дело с формой, неизбежно приходится решать судьбу данных, введенных в этой форме. Одно из первых практических применений JavaScript состояло в предоставлении способа проверки достоверности данных на стороне клиента вместо того, чтобы обращаться за этой услугой к серверу, посылая ему данные и получая результаты их проверки обратно. В то время проверка достоверности форм была чем-то особенным, не имела никакой практической поддержки в прикладном программном интерфейсе API и никакого реального внедрения в браузерах. Вместо этого программистам приходилось связывать вместе события и основные операции над текстом, чтобы предоставить удобный пользовательский интерфейс.

Ныне проверка достоверности форм находится в намного лучшем состоянии. В современные браузеры внедрен прикладной программный интерфейс API, предоставляющий обширные средства проверки достоверности форм в HTML и CSS. В распоряжении разработчиков веб-приложений имеются также регулярные выражения, которые намного более эффективны для проверки достоверности данных, несмотря на всю сложность их синтаксиса, чем, например, последовательный перебор строки по символам.

В этой главе основное внимание уделяется возможностям JavaScript в отношении форм. И хотя она посвящена в основном проверке достоверности форм, в ней также рассматриваются основные усовершенствования во взаимодействии Java Script с формами и новые прикладные программные интерфейсы API, доступные для обращения с формами.

Проверка достоверности форм в HTML и CSS

Как упоминалось выше, проверка достоверности форм прошла долгий путь развития с тех пор, как появился язык JavaScript. Чтобы оценить текущее состояние проверки достоверности форм, нужно рассмотреть не только возможности Java Script, но и HTML5 и CSS. Начнем со стороны HTML. За последние несколько лет язык HTML получил дальнейшее развитие, и в него было внедрено немало средств благодаря трудам группы WHATWG (Web Hypertext Application Technology Working Group — Рабочая группа по разработке гипертекстовых приложений для Интернета).

Эта организация способствовала дальнейшей эволюции и обновлению языка HTML до того, что теперь называется HTML5. Но обсуждение спецификации HTML5 выходит за рамки этой главы, и поэтому мы отсылаем читателей за дополнительными сведения к книге *HTML5 Programmer's Reference* Джонатана Рейда (Jonathan Reid), вышедшей в издательстве Apress в 2015 г.

Особо следует отметить усовершенствования HTML5 в отношении элементов управления форм. Эти усовершенствования можно разделить на следующие две обширные категории: внедрение новых элементов управления или стилей оформления элементов управления (полей ввода URL, селекторов данных и прочих), а также проверка достоверности форм. Сначала мы уделим внимание последней категории. Простая проверка достоверности форм была перенесена в язык HTML без необходимости прибегать к средствам JavaScript. Такая проверка достоверности доступна благодаря внедрению некоторых атрибутов в элементы управления формы. В качестве простого примера служит атрибут `required`, который применяется вместе с элементами ввода и требует обязательного наличия значения в поле перед тем, как форма будет передана на рассмотрение. Характерный пример разметки простой формы приведен в листинге 7.1.

Листинг 7.1. Простая форма

```
<!DOCTYPE html>
<html>
<head>
  <title>A basic form</title>
</head>
<body>
<h2>A basic form</h2>
<p>Please provide your first and last names.</p>

<form>
  <fieldset>
    <label for="firstName">First Name:</label><br/>
    <input id="firstName" name="firstName" type="text" required/><br/>
    <label for="lastName">Last Name:</label><br/>
    <input type="text" name="lastName" id="lastName"/><br/>
  </fieldset>
  <input type="submit" value="Submit the form"/>
  <input type="reset" value="Reset the form"/>
</form>

</body>
</html>
```

В данной форме обратите внимание на поле ввода с идентификатором `firstName`, где введен упоминавшийся ранее атрибут `required`. Если попытаться передать эту форму без заполнения данного поля, то получится результат, аналогичный приведенному на рис. 7.1.

Этот результат приблизительно одинаков в браузерах Chrome и Internet Explorer 11 (в браузере Chrome пустое поле не обрамлено красной рамкой, тогда как в Internet Explorer такая рамка имеется и хорошо заметна). Если сделать поля

`firstName` и `lastName` обязательными для заполнения, то обрамляющая рамка появится вокруг каждого поля, но всплывающая подсказка будет связана только с первым полем, где не введены данные. А как насчет специализации всплывающей подсказки? Мы рассмотрим вскоре данный вопрос, но для этого нам придется обратиться к средствам JavaScript.

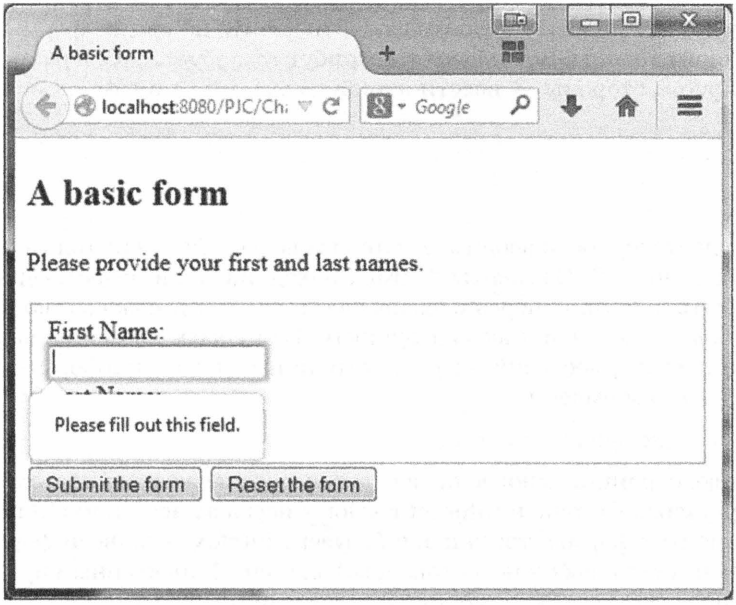


Рис. 7.1. В этой простой форме отсутствует имя того, кто ее заполнял

С помощью следующих атрибутов HTML можно активизировать и ряд других видов проверки достоверности.

- **Атрибут `pattern`.** Принимает регулярное выражение в качестве аргумента. Заключать регулярное выражение в знаки косой черты необязательно. Языковые средства HTML для регулярных выражений такие же, как и для Java Script (со всеми их достоинствами и недостатками). Этот атрибут присоединяется к элементу ввода. Следует, однако, иметь в виду, что типы ввода `email` и `url` подразумевают ввод значений, согласующихся с адресами электронной почты и URL соответственно. Проверка достоверности по шаблону не действует в версиях браузеров Safari 8, iOS Safari 8.1 и Opera Mini.
- **Атрибут `step`.** Требует, чтобы введенное значение было кратно указанной величине шага. Его действие ограничивается типами ввода даты и времени, а также `number`, `range`. Проверка достоверности по шагу действует в версиях браузеров Chrome 6.0, Firefox 16.0, IE 10, Opera 10.62 и Safari 5.0.
- **Атрибуты `min`/`max`.** Обозначают минимальное и максимальное значения, которые соответствуют не только числам, но и датам и времени. Такая проверка достоверности действует в версиях браузеров Chrome 41, Opera 27 и Chrome 41 для Android.

- **Атрибут `maxlength`.** Обозначает максимальную длину в символах для ввода данных в поле. Действителен только для типов ввода `text`, `email`, `search`, `password`, `tel` и `url`. Такая проверка достоверности обычно не особенно препятствует пользователю вводить слишком много данных в том поле, к которому присоединен данный атрибут. Она действует во всех современных браузерах.

Проверку достоверности в целом можно отключить на уровне формы двумя способами. С одной стороны, можно ввести атрибут `formnovalidate` в разметку кнопки Submit, а с другой стороны — ввести атрибут `novalidate` в сам элемент разметки формы.

CSS

Проверка достоверности вводимых данных была внедрена не только в HTML5, но и в спецификации CSS. Элементы формы, находящиеся в недостоверном состоянии, могут быть доступны через псевдокласс `:invalid`. К сожалению, реализация этого псевдокласса оставляет желать лучшего. Во-первых, элементы формы проверяются на их достоверность при загрузке страницы. Так, если в форме имеется следующее стилевое оформление:

```
:invalid { background-color: yellow }
```

то при загрузке страницы многие поля будут выделены желтым цветом фона. И во-вторых, в браузерах Chrome и Internet Explorer псевдокласс `:invalid` применяется только в элементах формы, тогда как в браузере Firefox — ко всей форме в целом, если в форме имеется любой недостоверный элемент. Характерный пример применения псевдокласса `:invalid` демонстрируется в листинге 7.2.

Листинг 7.2. Применение псевдокласса `:invalid`

```
<!DOCTYPE html>
<html>
<head>
  <title>A basic form</title>
  <style>
    :invalid {
      background-color: yellow
    }
  </style>
</head>
<body>
<h2>A basic form</h2>

<p>Please provide your first and last names.</p>


<form>
  <fieldset>
    <label for="firstName">First Name:</label><br/>
    <input id="firstName" name="firstName" type="text" required/><br/>
    <label for="lastName">Last Name:</label><br/>
```

```
    <input type="text" name="lastName" id="lastName"/><br/>
  </fieldset>
  <input type="submit" value="Submit the form"/>
  <input type="reset" value="Reset the
form"/>
</form>

</body>
</html>
```

В примере из листинга 7.2 вся форма отображается в окне браузера Firefox на желтом фоне, поскольку один из ее элементов находится в недействительном состоянии. Чтобы устранить этот недостаток, достаточно изменить стилевое оформление с `:invalid` на `input:invalid`. Это позволит добиться согласованного поведения формы во всех браузерах.

Для проверки достоверности в CSS предоставляются и другие псевдоклассы, в том числе следующие.

- **`:valid`**. Охватывает элементы, находящиеся в действительном состоянии.
- **`:required`**. Получает элементы разметки, в атрибуте `required` которых установлено логическое значение `true`.
- **`:optional`**. Получает элементы разметки, в которых отсутствует установленный атрибут `required`.
- **`:in-range`**. Предназначен для тех элементов разметки, которые находятся в своих минимальных или максимальных пределах. Не поддерживается в браузере Internet Explorer.
- **`:out-of-range`**. Предназначен для тех элементов разметки, которые выходят за свои минимальные или максимальные пределы. Не поддерживается в браузере Internet Explorer.

И наконец, рассмотрим эффекты красного свечения и всплывающего сообщения. В частности, красное свечение появляется вокруг недостоверного элемента в окне браузера Firefox после передачи формы на рассмотрение. (А в браузере Internet Explorer вокруг недостоверного элемента появляется красная рамка, но без свечения.) Эффект красного свечения воспроизводится в браузере Firefox с помощью псевдокласса `:-moz-ui-invalid`, но такое поведение можно переопределить следующим образом:

```
:-moz-ui-invalid { box-shadow: none }
```

К сожалению, в браузере Internet Explorer такой эффект не воспроизводится с помощью псевдокласса. Это означает, что мы достигли предела в проверке достоверности форм только средствами HTML и CSS. Но ведь у проверки достоверности форм имеются одни возможности, которые хотелось быть реализовать, а другие — не реализовать. И для этой цели нам, увы, придется снова обратиться к услугам JavaScript.

Проверка достоверности форм в JavaScript

Благодаря главным образом действующему стандарту HTML5 в языке JavaScript теперь появился согласованный прикладной программный интерфейс API для проверки достоверности форм. Он опирается на относительно простой критерий

проверки достоверности: имеется ли у данного элемента формы процедура проверки достоверности? Если таковая имеется, то проходит ли элемент проверку достоверности? Если он не проходит такую проверку, то почему? С этим процессом тесно связаны точки логического доступа для кода JavaScript через вызовы методов или перехват событий. И хотя эта система надежна, нельзя сказать, что она не требует некоторого усовершенствования. Но не будем забегать вперед.

Чтобы проверить элемент формы на достоверность, проще всего вызвать для него метод `checkValidity()`. Такой метод теперь имеется у объекта JavaScript, поддерживающего каждый элемент формы. Этот метод получает доступ к ограничению на проверку достоверности, установленному в HTML-разметке проверяемого элемента. Текущее значение элемента проверяется по каждому ограничению. Если оно не соответствует любому из ограничений, то метод `checkValidity()` возвращает логическое значение `false`, а иначе — логическое значение `true`. Вызовы метода `checkValidity()` не ограничиваются отдельными элементами. Они могут быть сделаны и относительно дескриптора формы. В таком случае вызов метода `checkValidity()` делегируется каждому из элементов формы. Если в результате всех подчиненных вызовов возвращается логическое значение `true`, то форма признается в целом достоверной. А если в результате любого из подчиненных вызовов возвращается логическое значение `false`, то форма считается недостоверной.

Помимо получения простого логического ответа о достоверности проверяемого элемента формы, можно выяснить причину, по которой он не прошел проверку достоверности. Свойство `validity` любого элемента формы содержит объект `ValidityState` со сведениями обо всех возможных причинах, по которым этот элемент не прошел проверку достоверности. Перебрав свойства этого объекта, можно выяснить конкретную причину непрохождения элементом проверки достоверности. Эти свойства перечислены в табл. 7.1.

Таблица 7.1. Свойства объекта `ValidityState`

Свойство	Пояснение
`valid`	Обозначает, является ли значение элемента достоверным. Проверку достоверности элемента формы следует начинать именно с этого свойства
`valueMissing`	Обозначает, что у элемента формы отсутствует обязательное значение
`patternMismatch`	Обозначает непрохождение проверки по шаблону, указанному в регулярном выражении
`rangeUnderflow`	Обозначает, что проверяемое значение оказывается меньше минимальной величины
`rangeOverflow`	Обозначает, что проверяемое значение оказывается больше максимальной величины
`stepMismatch`	Обозначает, что проверяемое значение не соответствует достоверной величине шага
`tooLong`	Обозначает, что проверяемое значение оказывается больше допустимой максимальной длины в символах
`typeMismatch`	Обозначает, что проверяемое значение не соответствует типу ввода **`email`** или **`url`**
`customError`	Принимает логическое значение **`true`**, если возникла специальная ошибка
`badInput`	Обозначает универсальную причину, когда браузер считает, что значение недостоверно ни по одной из перечисленных выше причин; не реализовано в браузере Internet Explorer

Процесс проверки достоверности элемента формы, собственно, состоит в проверке свойства `validity`. Рассмотрим этот процесс на конкретном примере. В листинге 7.3 приведена подходящая для этой цели часть HTML-разметки формы.

Листинг 7.3. HTML-разметка формы

```
<body>
<h2>A basic form</h2>

<p>Please fill in the requested information.</p>

<form id="nameForm">
  <div id="fields">
    <label for="firstName">First Name:</label><br/>
    <input id="firstName" name="firstName" type="text"
           class="foo" required/><br/>
    <label for="lastName">Last Name:</label><br/>
    <input type="text" name="lastName" id="lastName" required/><br/>

    <label for="phone">Phone</label><br/>
    <input type="tel" id="phone"/><br/>
    <label for="age">Age (must be over 13):</label><br/>
    <input type="number" name="age" id="age" step="2"
           min="14" max="100"/><br/>
    <label for="email">Email</label><br/>
    <input type="email" id="email"/><br/>
    <label for="url">Website</label><br/>
    <input type="url" id="url"/><br/>
  </div>

  <div id="buttons">
    <input id="overallBtn" value="Check overall validity"
           type="button"/>
    <input id="validBtn" type="button" value="Display validity"/>
    <input id="submitBtn" type="submit" value="Submit the form"/>
    <input type="reset" id="resetBtn" value="Reset the form"/>
  </div>
</form>

<div>
  <h2>Validation results</h2>
  <div id="vResults"></div>
  <div id="vDetails"></div>
</div>

</body>
```

Обратите внимание на то, что в данной форме кнопки проверки, размеченные элементами `submit`, `reset` и `validity`, находятся в отдельном дескрипторе `<div>`. Благодаря этому упрощается применение метода `document.querySelectorAll()` для извлечения только нужных полей данной формы, также находящихся в отдельном

дескрипторе . А теперь перейдем непосредственно к коду JavaScript, приведенному в листинге 7.4.

Листинг 7.4. Проверка достоверности формы

```
window.addEventListener( 'DOMContentLoaded', function () {
  var validBtn = document.getElementById( 'validBtn' );
  var overAllBtn = document.getElementById( 'overallBtn' );
  var form = document.getElementById( 'nameForm' );
    // или document.forms[0]
  var vDetails = document.getElementById( 'vDetails' );
  var vResults = document.getElementById( 'vResults' );

  overallBtn.addEventListener( 'click', function () {
    var formValid = form.checkValidity();
    vResults.innerHTML = 'Is the form valid? ' + formValid;
  } );

  validBtn.addEventListener( 'click', function () {
    var output = '';
    var inputs = form.querySelectorAll( '#fields > input' );

    for ( var x = 0; x < inputs.length; x++ ) {
      var el = inputs[x];
      output += el.id + ' : ' + el.validity.valid;
      if (! el.validity.valid) {
        output += ' [';
        for (var reason in el.validity) {
          if (el.validity[reason]) {
            output += reason
          }
        }
        output += ']';
      }
      output += '<br/>'
    }

    vDetails.innerHTML = output;
  } );
} );
```

Весь блок кода в данном примере связан с событием, наступающим по завершении загрузки документа, построенного по модели DOM. Напомним, что в данном случае не предпринимается попытка ввести обработчики событий в те элементы формы, которые еще не были созданы. Сначала извлекаются нужные элементы в пределах страницы, а именно: две кнопки проверки достоверности, области вывода div и форма. Затем устанавливается обработка событий для общей проверки достоверности. Но на этот раз ради простоты проверяется достоверность всей формы в целом. А результаты этой проверки выводятся в области vResults.

Второй обработчик событий охватывает проверку отдельного состояния достоверности каждого элемента формы. Соответствующие элементы извлекаются с помощью метода `querySelectorAll()` для получения всех полей по их идентификаторам в дескрипторе `<div>`. (Сделать это намного проще, чем писать обширный CSS-селектор для обнаружения тех типов ввода, которые не включают в себя кнопки передачи, сброса и проверки достоверности формы.) После получения нужных элементов остается лишь перебрать их и проверить содержимое свойства `valid`, подчиненного их свойству `validity`. Если данное свойство содержит логическое значение `false`, т.е. элемент формы недостоверный, то выводится причина его недостоверности. Опробуйте данный пример с самыми разными значениями, вводимыми в форме.

Данный пример выявляет ряд интересных особенностей. Прежде всего, если загрузить страницу и щелкнуть на кнопке Display validity (Показать достоверность), то поля `firstName` и `lastName` окажутся недостоверными, как и следовало ожидать, поскольку они пусты. Но поля `phone`, `age`, `email`, и `url` окажутся достоверными, хотя они также пусты! Если заполнение поля не является обязательным, то его пустое значение оказывается достоверным.

Следует также заметить, что поле `email` проходит две проверки достоверности: подразумеваемую проверку достоверности адреса электронной почты и проверку по шаблону. Попробуйте ввести неверный адрес электронной почты, в котором отсутствуют такие составляющие, как, например, `@foo.com`, и обнаружите непрохождение сразу нескольких проверок достоверности. А браузер Firefox сообщит, что значение не прошло проверку на соответствие типу ввода `typeMismatchbadInput`, если вы введете неполный адрес электронной почты (например, только имя пользователя без указания домена). При проверке достоверности формы нельзя полагаться только на свойство `valid`, поскольку нужно также знать причины, по которым элемент формы не прошел проверку достоверности, чтобы сообщить эту важную информацию пользователю. Ведь он не сможет в конечном итоге передать заполненную форму на рассмотрение, не соблюдя различные ограничения, накладываемые на проверку достоверности формы.

Проверка достоверности и пользователи

До сих пор мы уделяли основное внимание техническим вопросам проверки достоверности форм. Но нам нужно также обсудить, когда именно должна выполняться такая проверка. Для этого имеется целый ряд возможностей. Простое применение прикладного программного интерфейса API для проверки достоверности означает, что такая проверка автоматически происходит в момент передачи формы на рассмотрение. А благодаря методу `checkValidity()` имеется возможность начать проверку достоверности в любом заданном элементе именно тогда, когда это потребуется. Но, как показывает практика, проверку достоверности лучше всего начинать как можно раньше, хотя все зависит от конкретного проверяемого элемента формы. Начнем с того, что проверку достоверности следует начинать при изменении значения в поле форме. Обработчик событий, связанный с подобным изменением, достаточно присоединить к элементу управления формой и вызвать для него метод `checkValidity()`. Работая с прикладным программным интерфейсом API, совсем нетрудно ответить на вопрос, когда именно следует начинать проверку достоверности формы.

Но что делать, если мы не работаем с прикладным программным интерфейсом API для проверки достоверности форм? Одно из значительных ограничений на работу с этим прикладным интерфейсом состоит в том, что в нем не предусмотрены возможности для специальных проверок достоверности форм. В частности, фрагмент специального кода нельзя привязать к функции для выполнения в виде процедуры проверки достоверности. Но в какой-то момент такая потребность может, без сомнения, возникнуть. В таком случае имеет общий и практический смысл связать проверку достоверности с обработчиком событий, наступающих при изменениях в элементах формы. Из данного правила возможны исключения. Рассмотрим поле, в котором требуется сделать Ajax-вызов для проверки достоверности значения, возможно, исходя из первых нескольких символов, введенных в данном поле. В этом случае проверку достоверности следует связать с событием от клавиатуры, внедрив также функциональные возможности автоматического заполнения вводимых данных. Характерный тому пример рассматривается в следующей главе, посвященной технологии Ajax.

На какой бы стадии ни было решено начать проверку достоверности, нужно учитывать также интересы пользователей. Ведь пользователю будет очень неприятно обнаружить после заполнения формы, что большая часть введенных им данных оказалась недостоверной по самым разным причинам. Пользователям удобнее устранять ошибки непосредственно в полях ввода при заполнении формы, чем получать список ошибок после передачи формы на рассмотрение.

События проверки достоверности

Еще одно дополнение прикладного программного интерфейса API для проверки достоверности форм состоит в том, что недостоверные элементы формы способны теперь генерировать событие в связи с непрохождением проверки достоверности. Такое событие генерируется лишь в ответ на вызов метода `checkValidity()`. Этот метод может быть вызван как для самого проверяемого элемента, так и для содержащей его формы. Событие в связи с непрохождением проверки достоверности не всплывает. У форм такие события отсутствуют, несмотря на то, что формы могут быть недостоверными.

Событие проверки достоверности можно, как обычно, перехватить, вызвав метод `addEventListener()` для элемента управления, инициировавшего данное событие. Оказавшись в обработчике событий, сам объект события не предоставляет никакой информации о проверке достоверности. Для этого придется извлечь проверяемый элемент по ссылке `event.target`, а затем обратиться к его свойству `validity`, чтобы выяснить конкретную причину, по которой элемент оказался недостоверным. Но любопытно, что если вызвать метод `preventDefault()` для объекта события, то поведение стилевого оформления в браузере не будет затрагивать недостоверные элементы. Следует иметь в виду, что изменения в стилевом оформлении согласованно делаются только после передачи формы на рассмотрение. (Так, в браузере Firefox изменения в стилевом оформлении происходят лишь в том случае, если изменяется значение в элементе управления формы или фокус ввода переносится от него на другой элемент.) В разных браузерах это означает следующее.

- В браузере Chrome стилевое оформление не распространяется на недостоверные элементы формы, и хотя элементы снабжаются всплывающим сообщением, оно будет подавляться для недостоверного элемента.

- В браузере Firefox будет подавляться всплывающее сообщение, но не эффект красного свечения вокруг недостоверного элемента.
- В браузере Internet Explorer будет подавляться как всплывающее сообщение, так и красная рамка вокруг недостоверного элемента.

Рассмотрим пример, в котором демонстрируется такое поведение. Начнем с HTML-формы, которая отчасти нам уже знакома (листинг 7.5).

Листинг 7.5. Форма с событиями проверки достоверности

```
<!DOCTYPE html>
<html>
<head>
  <title>A basic form</title>
  <style>
    input:invalid {
      background-color: yellow
    }
  </style>
</head>
<body>
<h2>A basic form</h2>

<p>Please provide your first and last names.</p>

<form id="nameForm">
  <fieldset>
    <label for="firstName">First Name:</label><br/>
    <input id="firstName" name="firstName" type="text" required/><br/>
    <label for="lastName">Last Name:</label><br/>
    <input type="text" name="lastName" id="lastName" required/><br/>
  </fieldset>
  <div>
    <input type="submit" value="Submit the form"/>
    <input type="reset" value="Reset the form"/>
  </div>
  <div>
    <input id="firstNameBtn" type="button"
           value="Check first name validity."/>
    <input id="formBtn" type="button" value="Check form validity"/>
    <input id="preventBtn" type="button"
           value="Prevent default behavior"/>
    <input id="restoreBtn" type="button"
           value="Restore default behavior"/>
  </div>
</form>

<div id="vResults"></div>

<script src="listing_7_5.js"></script>
```

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

Обратите внимание на то, что недостоверные элементы ввода были дополнены стилевым оформлением. Такое стилевое оформление не связано со стандартным поведением для события в связи с непрохождением проверки достоверности. А теперь рассмотрим код сценария, поддерживающего данную HTML-форму, как показано в листинге 7.6.

Листинг 7.6. Обработка событий проверки достоверности в сценарии JavaScript

```
window.addEventListener( 'DOMContentLoaded', function () {
  var outputDiv = document.getElementById( 'vResults' );
  var firstName = document.getElementById( 'firstName' );

  firstName.addEventListener("focus", function(){
    outputDiv.innerHTML = '';
  });

  function preventDefaultHandler( evt ) {
    evt.preventDefault();
  }

  firstName.addEventListener( 'invalid', function ( event ) {
    outputDiv.innerHTML = 'firstName is invalid';
  } );

  document.getElementById( 'firstNameBtn' ).addEventListener(
          'click', function () {
    firstName.checkValidity();
  } );

  document.getElementById( 'formBtn' ).addEventListener(
          'click', function () {
    document.getElementById( 'nameForm' ).checkValidity();
  } );

  document.getElementById( 'preventBtn' ).addEventListener(
          'click', function () {
    firstName.addEventListener( 'invalid', preventDefaultHandler );
  } );

  document.getElementById( 'restoreBtn' ).addEventListener(
          'click', function () {
    firstName.removeEventListener( 'invalid', preventDefaultHandler );
  } );

} );
```

Как обычно, весь код сценария активизируется после того, как наступит событие `DOMContentLoaded`. В коде этого сценария имеется элементарный обработчик событий в связи с непрохождением проверки достоверности данных в поле `firstName`.

Этот обработчик событий выводит результаты в области `vResults`, размеченной дескриптором `<div>`. Кроме того, в коде этого сценария введены обработчики событий, наступающих в специализированных кнопках. Сначала создаются две служебные кнопки: одна — для проверки достоверности данных в поле `firstName`, а другая — для проверки достоверности данных во всей форме в целом. И наконец, в коде этого сценария реализовано поведение для отмены или восстановления стандартного поведения, связанного с недостоверными элементами. Опробуйте этот сценарий!

Специальная настройка проверки достоверности

Теперь в нашем распоряжении имеются почти все инструментальные средства для полного контроля над проверкой достоверности форм. В частности, мы можем выбрать активизацию конкретной проверки достоверности, контролировать момент ее выполнения, перехватывать недостоверные события и предотвращать стандартное поведение, особенно в отношении стилевого оформления. Но, как обсуждалось ранее, мы не можем, к сожалению, специально настроить конкретные процедуры проверки достоверности. Что же нам тогда остается? Ведь нам хотелось бы проследить за тем, какое сообщение о результатах проверки достоверности всплывает, когда пользователь передает форму на рассмотрение. Но внешний вид всплывающего сообщения не подлежит специальной настройке. Напомним о ряде ограничений, наложенных на прикладной программный интерфейс API для проверки достоверности и его реализацию.

Чтобы изменить сообщение, которое появляется, когда поле оказывается недостоверным, следует вызвать функцию `setCustomValidity()`, связанную с элементом управления формы, передав ей в качестве аргумента символьную строку с текстом всплывающего сообщения. Но такое действие повлечет за собой разные побочные эффекты. Так, в браузере Firefox проверяемое поле будет воспроизведено как недостоверное с эффектом красного свечения во время загрузки страницы. А вызов той же самой функции `setCustomValidity()` в браузере Internet Explorer или Chrome не окажет никакого действия во время загрузки страницы. Как упоминалось ранее, стилевое оформление может быть отключено в браузере Firefox путем переопределения псевдокласса `:-moz-ui-invalid`. Но дело в том, что, когда вызывается функция `setCustomValidity()`, в свойстве `customError` объекта `ValidityState` проверяемого элемента управления формы устанавливается логическое значение `true`. Это означает, что в свойстве `valid` объекта `ValidityState` устанавливается логическое значение `false`, а следовательно, проверяемый элемент формы оказывается недостоверным. И все это, конечно, отражается в содержимом сообщения о проверке достоверности! В итоге функция `setCustomValidity()` оказывается практически бесполезной.

В качестве альтернативы можно было бы воспользоваться полизаполнением. Имеется длинный перечень полизаполнений не только для проверки достоверности форм, но и для других элементов HTML5, возможно, не имеющих поддержки в каждом браузере, с которым приходится работать. Этот перечень можно найти по следующему адресу:

`https://github.com/Modernizr/Modernizr/wiki/HTML5-Cross-Browser-Polyfills`

Предотвращение проверки достоверности форм

Мы не рассмотрели еще один аспект проверки достоверности форм: ее отключение. Большая часть этой главы была посвящена применению нового прикладного программного интерфейса API для проверки достоверности и рассмотрению присущих ему ограничений. Но что, если возникнет некоторое препятствие (программная ошибка или несогласованность) для применения этого прикладного программного интерфейса? В таком случае нам останется одно из двух: отменить автоматическую проверку достоверности или заменить ее своей. Первое нас интересует больше, поскольку второе просто означает повторную реализацию прикладного программного интерфейса API по собственному усмотрению. Чтобы отключить проверку достоверности, достаточно ввести атрибут `novalidate` в элемент разметки формы. А для того чтобы предотвратить проверку достоверности при выборе кнопки передачи формы, следует ввести атрибут `formnovalidate` в разметку этой кнопки, хотя это не отменит проверку достоверности формы в целом. Если же прикладной программный интерфейс API для проверки достоверности потребуется заменить собственным, то для полной отмены проверки достоверности формы (на уровне родительского элемента) вряд стоит употреблять атрибут `novalidate`.

Резюме

В этой главе был рассмотрен новый прикладной программный интерфейс API для проверки достоверности форм в коде JavaScript. Несмотря на присущие этому интерфейсу ограничения, он является эффективным средством для автоматической проверки достоверности данных, вводимых пользователями в формах. Проверка достоверности формы происходит автоматически при ее передаче на рассмотрение, но в то же время ее можно осуществить в любой удобный момент по своему выбору. А при необходимости проверку достоверности можно отключить. Кроме того, имеется возможность специально настроить внешний вид достоверных и недостоверных элементов формы, а также сообщение, выводимое, когда элемент оказывается недостоверным.

Применение прикладного программного интерфейса API для проверки достоверности форм не лишено трудностей. Ему недостает ряда очень важных возможностей для специальной настройки проверки достоверности, в том числе стилевого оформления сообщений об ошибках или специальной настройки процедур проверки достоверности. А после исправления этих недостатков незначительные отличия в реализации среди основных браузеров могут быть устранены впоследствии. Одни официально признанные части прикладного программного интерфейса API (например, свойство `willValidate`) в настоящее время не реализованы, а другим (в частности, функции `setCustomValidity()`) присущи непреодолимые трудности.

В целом внедрение прикладного программного интерфейса API для проверки достоверности форм стало крупным шагом в дальнейшем развития JavaScript, HTML, CSS и браузеров. Остается только надеяться, что он будет усовершенствован в будущем.

ГЛАВА 8

Введение в Ajax

Термин *Ajax* придуман Джессом Джеймсом Гарреттом (Jesse James Garrett) из компании Adaptive Path для описания асинхронного обмена данными между клиентом и сервером, который стал возможным благодаря применению объекта `XMLHttpRequest`, предоставляемого всеми современными браузерами. Сокращение Ajax обозначает *A*synchronous *J*avaScript *a*nd *X*ML — Асинхронный JavaScript и XML, т.е. технологию, которая развилась в ряд методик, необходимых для создания динамических веб-приложений. Кроме того, отдельные компоненты технологии Ajax полностью взаимозаменяемы. Так, вполне допустимо применение формата JSON вместо XML.

С момента выхода в свет первого издания данной книги применение технологии Ajax претерпело существенные изменения. Будучи некогда экзотическим прикладным программным интерфейсом API, технология Ajax стала теперь стандартной и неотъемлемой частью арсенала инструментальных средств всякого, кто профессионально программирует на JavaScript. Консорциум W3C внес существенные коррективы в объект `XMLHttpRequest`, составляющий основу Ajax, дополнив его новыми средствами и уточнив поведение уже существующих средств. Одно из основных правил Ajax гласит: никаких соединений вне доменов. И это правило подкрепляется применением стандартного протокола CORS (Cross Origin Resource Sharing — Совместное использование ресурсов из разных источников).

В этой главе подробно рассматривается весь процесс обработки Ajax-запросов. Мы уделим внимание не только прикладному программному интерфейсу API для объекта `XMLHttpRequest`, но и обсуждению таких дополнительных вопросов, как обработка ответов на Ajax-запросы, реагирование на коды состояния по сетевому протоколу HTTP и т.д. И все это будет сделано для того, чтобы дать ясное представление о том, что происходит в цикле Ajax-запросов и ответов.

В этой главе не рассматривается прикладной программный интерфейс API для взаимодействий по технологии Ajax. Написать код по различным спецификациям, управляющим Ajax-процессом, нетрудно, но совсем другое дело — написать целую библиотеку для практического применения технологии Ajax. Поэтому мы рассмотрим часть библиотеки jQuery, относящуюся к технологии Ajax и устраняющую различные расхождения в Internet Explorer, Firefox, Chrome и прочих браузерах в отношении прикладного программного интерфейса API. А поскольку технология Ajax уже внедрена в jQuery, Dojo, Ext JS и ряде других библиотек, то не имеет смысла

изобретать колесо снова. Вместо этого мы рассмотрим примеры взаимодействий по технологии Ajax, написанные по современным (на момент написания данной книги) спецификациям. Эти примеры предназначены для целей демонстрации, а не окончательного применения. В связи с этим для применения технологии Ajax рекомендуется обратить внимание на такие служебные библиотеки, как jQuery, Zepto, Dojo, Ext JS и MooTools, или ориентированные на технологию Ajax библиотеки вроде Fermata и reqwest.

Итак, в этой главе рассматриваются следующие вопросы.

- Изучение различных типов HTTP-запросов.
- Выбор наилучшего способа для отправки данных на сервер.
- Анализ всего HTTP-ответа и выбор способа обработки не только удачного, но и неудачного (в какой-то степени) ответа.
- Чтение, обход и манипулирование данными из результата, полученного в ответе, присланном с сервера.
- Обработка асинхронных ответов.
- Составление запросов по доменам, разрешенным по протоколу CORS.

Применение технологии Ajax

Для простой реализации технологии Ajax не требуется особенно много кода, важнее выбрать наиболее подходящую реализацию. Например, вместо того чтобы вынуждать пользователя запрашивать совершенно новую веб-страницу после передачи формы на рассмотрение, в прикладном коде можно обработать переданную форму асинхронно, загрузив небольшую часть требующихся результатов по завершении процесса передачи формы на рассмотрение. В действительности, если связать Ajax-запросы с обработчиками событий, то больше нет никакой нужды ожидать передачи формы на рассмотрение. Этим объясняются "волшебные" свойства автозаполнения функции поиска в Google. Как только в Google начинается поиск по заданному критерию, инициируется Ajax-запрос, опирающийся на введенный критерий. По мере уточнения поиска посылаются другие Ajax-запросы. В поисковом механизме Google будут отображены не только предложения, основанные на первом возможном варианте ввода критерия поиска, но даже простая страница результатов. Пример такого процесса приведен на рис. 8.1.

Рис. 8.1. Пример применения сценария Instant Domain Search для мгновенного поиска фраз по мере их ввода

HTTP-запросы

Наиболее важная и, вероятно, согласованная стадия процесса обработки Ajax-запросов относится к формированию HTTP-запросов. Сетевой протокол HTTP (Hypertext Transfer Protocol — Протокол передачи гипертекста) специально предназначен для передачи HTML-документов и связанных с ними файлов. Правда, во всех современных браузерах поддерживаются средства для установления соединений по сетевому протоколу HTTP в динамическом и асинхронном режиме средствами JavaScript. Такая поддержка оказалась невероятно удобной для разработки более адаптивных веб-приложений.

Передача данных на сервер в асинхронном режиме и получение дополнительных данных в ответ является конечной целью технологии Ajax. Но порядок форматирования данных в конечном итоге зависит от конкретных требований к данному процессу.

В последующих разделах сначала будет показано, каким образом форматируются данные, передаваемые на сервер по разным HTTP-запросам. Затем будет рассмотрен порядок установления элементарных соединений с сервером и подробно показано, как это делается в кросс-браузерной среде.

Установление соединения

Все процессы обработки Ajax-запросов начинаются с установления соединения с сервером. Как правило, соединения с сервером организованы с помощью объекта `XMLHttpRequest`. (Единственное исключение заключается в формировании междоменных запросов в прежних версиях браузера Internet Explorer, как поясняется далее в этой главе. А до тех пор мы будем опираться на объект `XMLHttpRequest`.)

Обмен данными с помощью объекта `XMLHttpRequest` происходит в следующем порядке.

1. Создание экземпляра объекта `XMLHttpRequest`.
2. Настройка объекта подходящими установками.
3. Организация запроса с указанием конкретной HTTP-операции и места назначения.
4. Отправка запроса.

В листинге 8.1 демонстрируется, каким образом устанавливается элементарный запрос сервера по методу `GET`.

Листинг 8.1. Кросс-браузерные средства для установления HTTP-запроса сервера по методу `GET`

```
// Создать объект запроса
var xml = new XMLHttpRequest();

// Если требуется выполнить любую специальную настройку,
// это следует сделать именно здесь

// Открыть сетевой сокет
xml.open('GET', '/some/url.cgi', true);

// Установить соединение с сервером и отправить любые дополнительные данные
xml.send();
```

Как видите, код, требующийся для установления соединения с сервером, довольно прост и ничем особенным не отличается. Один ряд трудностей возникает в том случае, если требуются дополнительные средства (например, для проверки блокировок по времени или видоизмененных данных; подробнее об этом речь пойдет далее, в разделе "HTTP-ответ"). Другой ряд трудностей возникает в том случае, если требуется передать данные от клиента (браузера) серверу. Это одно из самых важных средств во всей технологии Ajax. Так, если требуется послать простые данные по указанному URL, то следует ли для этого выбрать метод `POST` или более сложный формат? Имея эти (и, конечно, другие) вопросы в виду, рассмотрим подробнее, что же требуется для упаковки данных и их отправки на сервер.

Сериализация данных

Первая стадия отправки массива данных на сервер состоит в их форматировании, чтобы их было легко прочитать на сервере. Этот процесс называется *сериализацией*. В связи с этим перед сериализацией данных необходимо найти ответы на следующие вопросы.

1. Какие именно данные посылаются? Посылаются ли данные в виде пар "переменная–значение", крупных массивов или файлов?
2. Каким методом следует посылать данные: `GET`, `POST` или с помощью другой HTTP-операции?
3. Какой формат следует выбрать? Таких форматов имеется два: `application/x-www-formurlencoded` и `multipart/form-data`. Первый формат иногда называется *кодировкой строки запроса* и принимает форму `var1=val1&var2=val2...`

С точки зрения JavaScript третий из перечисленных вопросов является самым важным, а первый и второй вопросы относятся к области проектирования. Они будут оказывать заметное влияние на разработку веб-приложения, но не обязательно потребуют написания другого кода. А вот вопрос выбора формата данных оказывает серьезное влияние на разработку веб-приложения.

В современных браузерах совсем не трудно обращаться с данными формата `multipart/form-data`. Благодаря объекту типа `FormData` можно очень просто выполнить сериализацию данных в объект, который браузер автоматически преобразует в формат `multipart/form-data`. К сожалению, пока еще не все браузеры полностью поддерживают форматирование данных, указанное в спецификации.

Объекты `FormData`

Объекты `FormData` являются относительно новым средством, предоставляемым в HTML5. Рабочая группа WHATWG и консорциум W3C намеревались предложить более объектно-ориентированный, преобразовательный подход к представлению данных, посылаемых в виде части Ajax-запроса, а на самом деле — HTTP-запроса. Соответственно объект `FormData` может быть инициализирован как пустой или как связанный с формой. Если он инициализируется в форме, то следует получить ссылку на элемент DOM, содержащий форму (как правило, вызвав метод `getElementById()`), и передать ее конструктору объектов `FormData`. В противном случае объект `FormData` окажется пустым. Так или иначе, новые данные можно ввести с помощью метода `append()`, как показано в листинге 8.2.

Листинг 8.2. Пример применения метода `append()` вместе с объектом `FormData`

```
// Создать объект FormData
var formDataObj= new FormData();

// Присоединить имена и значения для отправки на сервер
formDataObj.append('first', 'Yakko');
formDataObj.append('second', 'Wakko');
formDataObj.append('third', 'Dot');

// Создать объект запроса
var xml = new XMLHttpRequest();

// Установить запрос сервера по методу POST
xml.open('POST', '/some/url.cgi');

// Отправить объект FormData
xml.send(formDataObj);
```

В спецификациях WHATWG и W3C имеются некоторые отличия. Так, в спецификации WHATWG определены методы для удаления, получения и установки значений в объекте `FormData`, но ни один из этих методов не реализован в современных браузерах. Отчасти это объясняется тем, что в спецификации W3C определен только метод `append()`, а все современные браузеры следуют спецификации W3C — по крайней мере, так было на момент написания данной книги. Это означает, что объект `FormData` имеет односторонний характер: данные поступают, но доступны только на другой стороне HTTP-запроса.

Альтернативой объектам `FormData` служит сериализация данных в JavaScript. Это означает, что данные, предназначенные для передачи на сервер, нужно закодировать в формате URL и отправить на сервере как часть запроса. И хотя сделать это совсем не трудно, тем не менее следует иметь в виду ряд оговорок. Рассмотрим в качестве примера несколько типов данных, которые можно послать на сервер, а также вывод полученных результатов в удобном для сервера сериализированном виде (листинг 8.3).

Листинг 8.3. Примеры исходных объектов JavaScript, преобразованных в сериализированную форму

```
// Простой объект содержащий пары "ключ-значение"
{
  name: 'John',
  last: 'Resig',
  city: 'Cambridge',
  zip: 02140
}
// Сериализированная форма этого объекта
name=John&last=Resig&city=Cambridge&zip=02140

// Еще один ряд данных со многими значениями
[
  { name: 'name', value: 'John' },
  { name: 'last', value: 'Resig' },
```

```
  { name: 'lang', value: 'JavaScript' },
  { name: 'lang', value: 'Perl' },
  { name: 'lang', value: 'Java' }
]

// И сериализированная форма этих данных
name=John&last=Resig&lang=JavaScript&lang=Perl&lang=Java

// И наконец, найти ряд элементов ввода данных
[
  document.getElementById( 'name' ),
  document.getElementById( 'last' ),
  document.getElementById( 'username' ),
  document.getElementById( 'password' )
]

// И сериализовать их в строку данных
name=John&last=Resig&username=jeresig&password=test
```

Формат, используемый для сериализации данных, является стандартным для передачи дополнительных параметров в HTTP-запросе. Их вряд ли можно увидеть в стандартном HTTP-запросе по методу `GET` вроде следующего:

```
http://someurl.com/?name=John&last=Resig
```

Эти данные могут быть также переданы в запросе по методу `POST`, причем в намного большем количестве, чем по методу `GET`. Мы рассмотрим эти отличия в последующем разделе, а до тех пор рассмотрим пример построения стандартных средств для сериализации структур данных, приведенных в листинге 8.3. Соответствующая функция для этой цели представлена в листинге 8.4. Эта функция способна выполнять сериализацию большинства элементов ввода, за исключением элементов ввода со множественным выбором.

Листинг 8.4. Стандартная функция для сериализации структур данных по схеме параметров, совместимой с сетевым протоколом HTTP

```
// Сериализовать ряд данных. Эта функция может принимать
// разные типы объектов:
// - Массив элементов ввода.
// - Хеш-код пар "ключ-значение".
// Эта функция возвращает сериализированную строку данных
function serialize(a) {
  // Ряд сериализированных результатов
  var s = [];

  // Если передан массив, то считать его массивом элементов формы
  if ( a.constructor === Array ) {

    // Сериализовать элементы формы
    for ( var i = 0; i < a.length; i++ )
      s.push( a[i].name + '=' + encodeURIComponent( a[i].value ) );
```

```
    // Иначе считать, что это объект, состоящий из пар "ключ-значение"
  } else {
    // Сериализовать пары "ключ-значение"
    for ( var j in a )
      s.push( j + '=' + encodeURIComponent( a[j] ) );
  }

  // Возвратить результат сериализации данных
  return s.join('&');
}
```

Итак, данные получены в сериализированной форме (т.е. в виде простой строки данных). А теперь посмотрим, каким образом можно передать эти данные на сервер по HTTP-запросу методом GET или POST.

Установление HTTP-запроса по методу GET

Вернемся к вопросу установления HTTP-запроса по методу GET с помощью объекта XMLHttpRequest, но на этот раз рассмотрим отправку вместе с этим запросом сериализированных данных. В листинге 8.5 приведен простой пример, показывающий, как это делается.

Листинг 8.5. Кросс-браузерные средства для установления HTTP-запроса на сервер по методу GET (и без чтения любых результирующих данных)

```
// Создать объект запроса
var xml = new XMLHttpRequest();

// Установить асинхронный запрос по методу GET
xml.open('GET', '/some/url.cgi?' + serialize( data ), true);

// Установить соединение с сервером
xml.send();
```

Следует заметить, что сериализированные данные присоединяются к URL сервера, отделяясь знаком ?. Всем веб-серверам и каркасам приложений известно, что данные, включаемые в URL после знака ?, следует интерпретировать как сериализированный ряд пар "ключ–значение".

Установление HTTP-запроса по методу POST

Другой способ установления HTTP-запроса сервера с помощью объекта XMLHttpRequest происходит по методу POST, включая и совершенно другой способ оправки данных на сервер. В частности, запрос по методу POST рассчитан на отправку данных в любом формате и любой длины, т.е. не ограничиваясь только сериализированной строкой данных.

Формат сериализации, применяемый для данных, обычно получает тип содержимого application/x-www-form-urlencoded при передаче данных на сервер. Это означает, что на сервер можно отправить данные только в формате XML с помощью типа содержимого text/xml или application/xml либо даже в виде объекта JavaScript с использованием типа содержимого application/json. Простой пример установле-

ния запроса и отправки дополнительно сериализированных данных приведен в листинге 8.6.

Листинг 8.6. Кросс-браузерные средства для установления HTTP-запроса на сервер по методу POST (и без чтения любых результирующих данных)

```
// Создать объект запроса
var xml = new XMLHttpRequest();

// Установить асинхронный запрос по методу POST
xml.open('POST', '/some/url.cgi', true);

// Установить заголовок content-type, чтобы сообщить серверу,
// как интерпретировать посылаемые ему данные
xml.setRequestHeader(
    'Content-Type', 'application/x-www-form-urlencoded');

// Установить соединение с сервером и отправить на сервер
// сериализированные данные
xml.send( serialize( data ) );
```

Чтобы расширить предыдущий пример, рассмотрим отправку данных на сервер не в сериализированном формате. Характерный тому пример приведен в листинге 8.7.

Листинг 8.7. Пример отправки данных на сервер в формате XML по методу POST

```
// Создать объект запроса
var xml = new XMLHttpRequest();

// Установить асинхронный запрос по методу POST
xml.open('POST', '/some/url.cgi', true);

// Установить заголовок content-type, чтобы сообщить серверу,
// как интерпретировать посылаемые ему данные в формате XML
xml.setRequestHeader( 'Content-Type', 'text/xml');

// Установить соединение с сервером и отправить на сервер
// сериализированные данные
xml.send( '<items><item id='one'/><item id='two'/></items>' );
```

Возможность посылать данные в массовом количестве имеет большое значение, поскольку на количество передаваемых данных не накладывается никаких ограничений. С другой стороны, количество данных, передаваемых по методу GET, ограничивается парой килобайт в зависимости от конкретного браузера. Это позволяет создавать реализации различных протоколов передачи данных, например XML-RPC или SOAP. Но ради простоты обсуждения ограничимся наиболее общими и полезными форматами, которые могут быть доступны в виде HTTP-ответа.

HTTP-ответ

Класс `XMLHttpRequest` второго уровня теперь обеспечивает лучший контроль над извещением браузера по поводу того, как отправлять данные обратно. С этой целью устанавливается свойство `responseType`, а данные принимаются с помощью свойства `response`. Рассмотрим сначала очень простой пример обработки данных ответа, полученного от сервера, как демонстрируется в листинге 8.8.

Листинг 8.8. Установление соединения с сервером и чтение результирующих данных

```
// Создать объект запроса
var request = new XMLHttpRequest();

// Установить асинхронный запрос по методу POST
request.open('GET', '/some/image.png', true);

// Большой двоичный объект
request.responseType = 'blob';

request.addEventListener('load', downloadFinished, false);

function downloadFinished(evt){
  if(this.status == 200){
    var blob = new Blob([this.response], {type: 'img/png'});
  }
}
```

В данном примере показано, как получать двоичные данные и преобразовывать их в файл формата PNG. В свойстве `responseType` может быть установлено одно из следующих значений.

- **`Text`**. Результаты возвращаются в виде текстовой строки.
- **`ArrayBuffer`**. Результаты возвращаются в виде двоичных данных.
- **`Document`**. Результаты предполагаются в виде XML-документа, хотя это может быть и HTML-документ.
- **`Blob`**. Результаты возвращаются в виде файла вроде объекта исходных данных.
- **`JSON`**. Результаты возвращаются в виде документа формата JSON.

Итак, показав, как устанавливать свойство `responseType`, можно перейти к рассмотрению способов контроля над ходом обработки запроса.

Контроль над ходом обработки запроса

Как было показано ранее, функция `addEventListener()` делает прикладной код более простым и удобочитаемым. И в данном случае к объекту запроса применяется тот же самый прием. Независимо от того, загружаются ли данные на сервер или выгружаются из него, события можно принимать, как показано в листинге 8.9.

Листинг 8.9. Прием событий о ходе обработки запроса на сервере с помощью функции `addEventListener()`

```
var myRequest = new XMLHttpRequest();

myRequest.addEventListener('loadstart', onLoadStart, false);
myRequest.addEventListener('progress', onProgress, false);
myRequest.addEventListener('load', onLoad, false);
myRequest.addEventListener('error', onError, false);
myRequest.addEventListener('abort', onAbort, false);

// Непременно ввести приемники событий, прежде чем выполнять
// метод send() или open()

myRequest.open('GET', '/fileOnServer');

function onLoadStart(evt){
  console.log('starting the request');
}

function onProgress(evt){
  var currentPercent = (evt.loaded / evt.total) * 100;
  console.log(currentPercent);
}

function onLoad(evt){
  console.log('transfer is complete');
}

function onError(evt){
  console.log('error during transfer');
}

function onAbort(evt){
  console.log('the user aborted the transfer');
}
```

Теперь вы можете узнать намного больше, чем прежде, о том, что происходит с вашим файлом. В частности, используя свойства `loaded` и `total`, можно выяснить, до какой степени в процентах загружен файл. Если по какой-нибудь причине пользователь решит остановить загрузку, будет получено событие преждевременного завершения данного процесса (`abort`). А если что-нибудь произойдет с файлом или его загрузка завершится, будет получено событие, связанное с ошибкой (`error`) или окончанием загрузки (`load`) соответственно. И наконец, если запрос сервера делается первый раз, то будет получено событие, связанное с началом загрузки (`loadstart`). А теперь рассмотрим вкратце блокировки по времени.

Проверка блокировок по времени и совместное использование ресурсов из разных источников

Проще говоря, блокировки по времени позволяют установить время ожидания приложением ответа от сервера. Установить блокировку по времени и отслеживать ее совсем не трудно. В примере из листинга 8.10 демонстрируется, каким образом происходит проверка блокировки по времени в собственном веб-приложении.

Листинг 8.10. Пример проверки блокировки запроса по времени

```
// Создать объект запроса
var xml = new XMLHttpRequest();

// Ожидать обработки запроса в течение 5 секунд,
// прежде чем отказаться от нее
xml.timeout = 5000;

// Принимать событие блокировки по времени
xml.addEventListener('timeout', onTimeOut, false);

// Установить асинхронный запрос по методу POST
xml.open('GET', '/some/url.cgi', true);

// Установить соединение с сервером
xml.send();
```

По умолчанию браузеры не позволят веб-приложениям делать запросы серверов, кроме того сайта, с которого они исходят. Этим пользователи защищаются от атак типа межсайтового выполнения сценариев. Сервер должен разрешить запросы, а иначе возникнет ошибка недостоверного доступа (`INVALID_ACCESS`). Заголовок, передаваемый серверу в запросе, будет выглядеть следующим образом:

```
Access-Control-Allow-Origin:*
  // Использовать метасимвол (*), чтобы разрешить доступ кому угодно
Access-Control-Allow_origin:http://spaceappleyoshi.com
  // Разрешить доступ из определенного домена
```

Резюме

В этой главе было показано, как обращаться с данными на сервере. В частности, у сервера можно запросить конкретные результаты, которые предполагается получить в ответ. Кроме того, можно принимать события, сообщающие о ходе передачи файла или ошибке, возникающей в течение данного процесса. И наконец, в этой главе были рассмотрены блокировки по времени и совместное использование ресурсов из разных источников (или протокол CORS). А в следующей главе мы рассмотрим ряд инструментальных средств для веб-производства.

ГЛАВА 9

Инструментальные средства для веб-производства

Инструментальные средства для разработки веб-сайтов совершенствовались с годами. Они прошли путь от простых редакторов вроде Notepad до полноценных сред разработки типа WebStorm. В распоряжении разработчиков появились также библиотеки, подобные JQuery. Они могут воспользоваться Handlebars в качестве механизма построения по шаблону или AngularJS в качестве полноценного каркаса по шаблону MVC. Имеются также среды модульного тестирования и системы контроля версий, помогающие разработчикам веб-приложений лучше и быстрее выполнять свою работу. Итак, имея в своем распоряжении все эти инструментальные средства, как же правильно поддерживать их в организованном порядке?

Чтобы ответить на этот вопрос, мы разделим данную главу на две части. Сначала мы рассмотрим инструментальные средства для создания веб-сайтов, а затем инструментальные средства для отслеживания изменений на действующих сайтах. Для создания веб-сайтов мы исследуем такие инструментальные средства, как Yeoman, Grunt, Bower и Node Package Manager (NPM), а для отслеживания изменений — Git.

Прежде чем обсуждать совместную работу всех этих инструментальных средств, рассмотрим вкратце назначение каждого из них в отдельности.

- Bower — это система управления пакетами. Ее назначение — обеспечить загрузку и установку клиентского кода, от которого зависит разрабатываемый проект. Сайт: `http://bower.io/`.
- Grunt — это инструментальное средство построения, позволяющее автоматизировать решение многих видов задач, включая модульное тестирование, статический анализ (контроль ошибок в коде JavaScript) и ввод прикладного кода в систему контроля версий. Может быть также использовано при развертывании прикладного кода на сервере. Сайт: `http://gruntjs.com/`.
- Yeoman — это инструментальное средство для построения каркаса проектов. Создает файлы и папки, составляющие каркасный вариант проекта, затем использует Bower для сборки всего кода, от которого зависит проект, и наконец, применяет инструментальное средство построения (вроде Grunt) для

автоматизации задач. Все это делается с помощью так называемых генераторов. Сайт: `http://yeoman.io/`.

- Node Package Manager (NPM) — это инструментальное средство служит для управления пакетами, которые выполняются на платформе Node.js. По мере распространения платформы Node.js некоторые из подобных пакетов были специально созданы для разработки кода на стороне клиента, а не только на стороне сервера, где применяется эта платформа. Сайт: `https://nodejs.org/`.
- Git — это система контроля версий. Если вам знакомы такие инструментальные средства, как Subversion или Perforce, то система Git похожа на них. Она служит для отслеживания всех рабочих файлов проекта и способна указывать на их отличия. Сайт: `http://git-scm.com/`.

Построение каркаса проектов

Компьютеры отлично справляются с теми задачами, которые людям не хочется решать самим, причем они могут делать это многократно и без устали. В частности, никому не хочется создавать папки для файлов изображений, таблиц CSS и сценариев JavaScript всякий раз, когда начинается новый проект. Имеется немало мелких задач, которые вполне можно автоматизировать. Поэтому было бы совсем не плохо начать проект, в котором уже имеются все нужные папки, созданные по одной команде.

Именно по такому принципу и действует построение каркаса проектов. Большинство веб-сайтов организовано одинаково, и поэтому нет нужды разрабатывать их структуру вручную. Инструментальное средство Yeoman позволяет построить каркас проекта практически любого типа. Для быстрой и простой установки проекта в Yeoman применяются генераторы, опираясь на нормы передовой практики программирования.

Генераторы представляют собой шаблоны, которые может создать кто угодно. Имеются отдельные команды разработчиков, занимающиеся проектами по созданию "официальных" генераторов, но если они не подходят для конкретных целей, то ничто не мешает создать свой подходящий генератор. Кроме того, генераторы имеют открытый исходный код, чтобы всякий желающий мог просмотреть его и разобраться во внутреннем механизме работы генераторов. Но для того чтобы работать с Yeoman, нужно сначала установить платформу Node.js.

NPM — основа всего

Инструментальное средство Node Package Manager (NPM) дает возможность управлять зависимостями в разрабатываемом веб-приложении. Это означает, что если для разработки проекта требуется внешний код (например, из библиотеки JQuery), то NPM упрощает его внедрение в проект, а также установку большинства требующихся инструментальных средств. NPM является составной частью платформы Node.js, которая является средой с открытым исходным кодом для разработки серверных приложений на JavaScript. И хотя мы не собираемся рассматривать здесь построение проекта на платформе Node.js, тем не менее, нам понадобится установить узел. Это можно сделать несколькими способами, и в качестве примера мы воспользуемся самым простым из них.

Когда вы обращаетесь на веб-сайт по адресу `https://nodejs.org/`, то он автоматически определяет операционную систему, которой вы пользуетесь. Щелкните на кнопке Install (Установить), чтобы загрузить и запустить на выполнение программу установки узла. По завершении установки перейдите в режим терминала (в Mac OS и Linux) или режим командной строки (в Windows) и введите команду **`node -version`**. В окне терминала или командной строки появится текущая версия установленного узла.

Итак, установив узел, мы можем получить все остальное из того, что нам потребуется в дальнейшем. В частности, для установки Yeoman достаточно ввести команду **`npm install -g yo`**, для установки Grunt — команду **`npm install -g grunt-cli`**, а для установки Bower — команду **`npm install -g bower`**. Указание параметра **`-g`** в командной строке означает, что установка будет выполнена глобально. Все эти инструментальные средства могут быть запущены на выполнение из любой папки при создании новых проектов. А параметр **`-cli`** означает, что будет использоваться интерфейс командной строки. Ради упражнения мы будем в дальнейшем пользоваться режимом командной строки, к которому полезно и стоит привыкнуть. А теперь можно установить генератор и приступить к рассмотрению остальных инструментальных средств.

Генераторы

Как обсуждалось ранее, генераторы на самом деле являются шаблонами, описывающими структуру веб-сайта. Эти шаблоны можно настроить, передав Yeoman различные параметры. На веб-сайте `http://yeoman.io/` можно найти перечень генераторов и обратные ссылки на информационные хранилища GitHub. В этих хранилищах хранятся все инструкции по применению генераторов. Так, если требуется создать веб-сайт средствами AngularJS, достаточно ввести следующую команду:

```
npm install -g generator-angular
```

В итоге будет установлен генератор AngularJS. А если потребуется создать AngularJS-сайт, дополнив его Karma — исполнителем тестов для модульного тестирования кода JavaScript, то команда установки будет выглядеть следующим образом:

```
npm install -g generator-angular generator-karma
```

А теперь, когда установлен генератор, введя команду **`yo`** в режиме командной строки, вы можете получить список установленных генераторов и обновить его. К этому моменту создайте папку для своего проекта, перейдите к этой папке и введите команду

```
yo angular
```

Yeoman задаст вам ряд вопросов о разрабатываемом вами приложении. Например, у вас спросят, желаете ли вы воспользоваться инструментальным средством Sass, как показано на рис. 9.1.

Далее вам зададут еще несколько вопросов по поводу установки вашего проекта. По завершении вопросов Bower извлечет из хранилища GitHub последние версии всех библиотек, которые могут потребоваться, а затем построит автоматически каркас проекта. Как только все необходимое будет установлено, введите следующую команду, чтобы посмотреть разрабатываемый веб-сайт в действии:

```
grunt serve
```

```
Russs-MacBook-Pro:yeomanAngular asciibn$ git commit -m "My First Commit"
[master (root-commit) 9e29ab1] My First Commit
 21 files changed, 1689 insertions(+)
 create mode 100644 Gruntfile.js
 create mode 100644 README.md
 create mode 100644 app/.buildignore
 create mode 100644 app/.htaccess
 create mode 100644 app/404.html
 create mode 100644 app/favicon.ico
 create mode 100644 app/images/yeoman.png
 create mode 100644 app/index.html
 create mode 100644 app/robots.txt
 create mode 100644 app/scripts/app.js
 create mode 100644 app/scripts/controllers/about.js
 create mode 100644 app/scripts/controllers/main.js
 create mode 100644 app/styles/main.scss
 create mode 100644 app/views/about.html
 create mode 100644 app/views/main.html
 create mode 100644 bower.json
 create mode 100644 package.json
 create mode 100644 test/.jshintrc
 create mode 100644 test/karma.conf.js
 create mode 100644 test/spec/controllers/about.js
 create mode 100644 test/spec/controllers/main.js
Russs-MacBook-Pro:yeomanAngular asciibn$ git log
commit 9e29ab14dc88d176c9578d8546d950a8387b8ee8
Author: Russ Ferguson <rferguson@technologycoach.com>
Date:   Sat May 16 18:25:07 2015 -0400

    My First Commit
Russs-MacBook-Pro:yeomanAngular asciibn$
```

Рис. 9.1. Установка AngularJS-сайта средствами Yeoman

На этот раз для создания локального веб-сервера в текущей папке и обслуживания начальной страницы веб-сайта используется Grunt (рис. 9.2).

Вот, собственно, и все, что нужно сделать. В итоге вы получите готовый к работе веб-сайт. А теперь самое время ввести полученный в итоге код в систему контроля версий.

Контроль версий

Изменения происходят постоянно. Рабочие файлы обновляются снова и снова, но в работе над проектом иногда возникают перерывы. В одних случаях оказывается достаточно простой отмены изменений, а в других — вернуться к предыдущему состоянию проекта, особенно если он разрабатывается коллективно с внесением многих изменений. Одно изменение способно нарушить нормальную работу всего веб-сайта, и в этом случае оказывается нелегко отследить причину возникшей неполадки. Именно здесь и приходит на помощь система контроля версий.

Такой системой контроля версий и является Git. Мы выбрали ее потому, что она весьма распространена среди разработчиков и предоставляет удобный графический пользовательский интерфейс (ГПИ) для своих клиентов. Аналогично рассмотренному выше примеру установки узла Node.js, мы пойдем по кратчайшему пути, чтобы установить систему Git.

Итак, перейдите на веб-сайт Git по адресу `http://git-scm.com/`, чтобы загрузить и установить эту систему контроля версий. Установив систему Git, можете настроить ее в режиме командной строки. В частности, чтобы идентифицировать себя, введите команды **`git config -global user.name "`*`ваше имя`*`"`** и **`git config -global user.email "`*`ваш адрес электронной почты`*`"`**.

```
1. bash
  (use "git rm --cached <file>..." to unstage)

        new file:   README.md
        new file:   app/.buildignore
        new file:   app/.htaccess
        new file:   app/404.html
        new file:   app/favicon.ico
        new file:   app/images/yeoman.png
        new file:   app/index.html
        new file:   app/robots.txt
        new file:   app/scripts/app.js
        new file:   app/scripts/controllers/about.js
        new file:   app/scripts/controllers/main.js
        new file:   app/styles/main.scss
        new file:   app/views/about.html
        new file:   app/views/main.html

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   README.md

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        .bowerrc
        .editorconfig
        .gitattributes
        .gitignore
        .jshintrc
        .travis.yml
        .yo-rc.json
        Gruntfile.js
        bower.json
        package.json
        test/

Russs-MBP:yeomanAngular asciibn$
```

Рис. 9.2. Запуск AngularJS-сайта на локальном сервере средствами Yeoman

После установки и настройки системы Git мы собираемся быстро ввести файлы и зафиксировать их локально. С этой целью перейдите к папке проекта и, находясь в ней, введите команду **`git init`**. В итоге будет создана папка `.Git`, которая будет содержать всю информацию о проекте. Как правило, эта папка невидима, и поэтому у вас может возникнуть потребность внести изменения в некоторые настройки операционной системы, чтобы сделать ее видимой. Далее проверим состояние фиксации (т.е. текущего варианта проекта). Введите команду **`git status`**, чтобы убедиться, что в данный момент в систему контроля версий не было введено ни одного файла из данного проекта (рис. 9.3).

Ввод файлов, обновления и первая фиксация изменений

Далее введем файлы, чтобы отслеживать в них изменения средствами Git. Чтобы ввести файлы в информационное хранилище Git, достаточно набрать команду **`git add`** ***имя файла/папка*** в режиме командной строки. В итоге система Git начнет автоматически отслеживать изменения в файлах. Как показано на рис. 9.4, в систему

Git по команде **`git add app/`** введена папка `app` текущего проекта. После этого можно еще раз проверить состояние проекта и посмотреть результаты контроля версий.

```
1. bash
Russs-MBP:yeomanAngular asciibn$ git status
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

        new file:   app/.buildignore
        new file:   app/.htaccess
        new file:   app/404.html
        new file:   app/favicon.ico
        new file:   app/images/yeoman.png
        new file:   app/index.html
        new file:   app/robots.txt
        new file:   app/scripts/app.js
        new file:   app/scripts/controllers/about.js
        new file:   app/scripts/controllers/main.js
        new file:   app/styles/main.scss
        new file:   app/views/about.html
        new file:   app/views/main.html

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        .bowerrc
        .editorconfig
        .gitattributes
        .gitignore
        .jshintrc
        .travis.yml
        .yo-rc.json
        Gruntfile.js
        README.md
        bower.json
        package.json
        test/

Russs-MBP:yeomanAngular asciibn$
```

Рис. 9.3. Файлы проекта еще не введены в систему Git

Продолжим вводить файлы и, в частности, файлы `bower.json` и `package.json`. Эти файлы служат для отслеживания зависимых модулей и их версий в проекте. А в файле `Gruntfile.js` будут находиться все задачи, которые можно выполнить. Ранее мы уже запускали задачу на выполнение, введя команду **`grunt serve`**. По этой команде автоматически запускается локальный сервер.

В качестве нормы передовой практики папки `node_modules` и `bower_components` не вводятся в систему контроля версий. Их можно переустановить в дальнейшем по командам **`npm install`** и **`bower install`**. А те файлы, которые не требуется вводить в систему Git, определены в файле с расширением **`.gitignore`**. Этот файл можно видоизменить, чтобы включить в него типы файлов и все, что должно быть проигнорировано в Git.

Как упоминалось выше, в рабочих файлах проекта со временем происходят изменения, и поэтому к системе Git можно обратиться, чтобы выяснить, когда конкретный файл был изменен. Но сначала этот файл нужно ввести в информационное хранилище. Как было показано ранее, это делается по команде **`git add`**. А когда в

файле произойдут изменения, его состояние можно запросить снова по команде **`git status`**. В итоге будут перечислены любые файлы, изменившиеся с момента их ввода в информационное хранилище. Не следует лишь забывать, что системе нужно Git дать знать о файле, прежде чем вносить в него изменения, чтобы они автоматически отслеживались во времени. Пример отслеживания изменений в файле `README.md` приведен на рис. 9.5.

```
Russs-MBP:yeomanAngular asciibn$ git status
On branch master

Initial commit

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        .bowerrc
        .editorconfig
        .gitattributes
        .gitignore
        .jshintrc
        .travis.yml
        .yo-rc.json
        Gruntfile.js
        README.md
        app/
        bower.json
        package.json
        test/

nothing added to commit but untracked files present (use "git add" to track)
Russs-MBP:yeomanAngular asciibn$
```

Рис. 9.4. Папка `app`, введенная в систему Git

Измененный однажды файл можно отредактировать снова, как это делалось прежде. И при последующей проверке состояния система контроля версий вернется к списку отслеживаемых элементов. А теперь, когда все файлы отслеживаются в Git, их можно зафиксировать. Такую фиксацию можно сравнить с моментальным снимком проекта в его текущем состоянии (рис. 9.6). Если же в проекте произойдут какие-нибудь изменения, можно всегда вернуться к его прежнему состоянию.

Чтобы зафиксировать изменения, достаточно ввести команду **`commit -m "примечания к фиксации"`**. Параметр **`-m`** обозначает сообщение. С другой стороны, для составления нужного сообщения можно воспользоваться текстовым редактором. Если же требуется просмотреть предысторию сообщений, достаточно ввести команду **`git log`**.

Итак, мы рассмотрели порядок отслеживания изменений в файлах проекта на локальной машине, что вполне подходит для индивидуальной разработки. Но если потребуется обмениваться разрабатываемым кодом или вести проект коллективно, то придется добавить серверный компонент. Двумя наиболее распространенными его вариантами являются GitHub (`https://github.com/`) и BitBucket (`https://bitbucket.org/`). С помощью любого из этих компонентов можно организовать удаленное хранилище в дополнение к файлам на локальной машине.

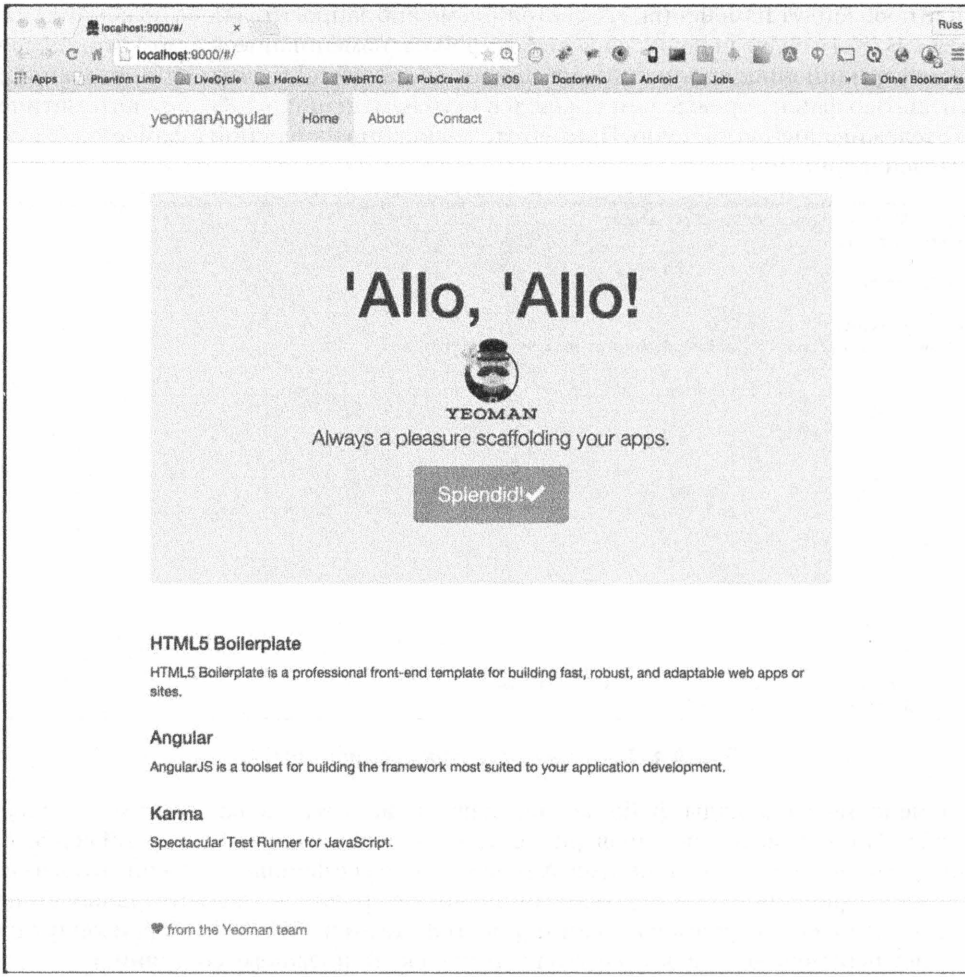


Рис. 9.5. Система Git способна автоматически отслеживать изменения в файле

```
Last login: Tue Apr 21 07:10:46 on ttys001
Russs-MBP:yeomanAngular asciibn$ yo angular

     _-----_
    |       |    .--------------------------.
    |--(o)--|    |    Welcome to Yeoman,    |
   `---------´   |   ladies and gentlemen!  |
    ( _´U`_ )    '--------------------------'
    /___A___\
     |  ~  |
   __'.___.'__
 ´   `  |°  ´ Y `

Out of the box I include Bootstrap and some AngularJS recommended modules.

? Would you like to use Sass (with Compass)? (Y/n)
```

Рис. 9.6. Состояние файла, зафиксированное и показанное в системном журнале

Резюме

Надеемся, что по прочтении этой главы вы обнаружите немало полезных для себя ресурсов. Возможность быстро создать веб-сайт с помощью генераторов Yeoman позволит вам сэкономить немало времени и труда в работе над новым проектом. Yeoman можно также использовать в качестве обучающего средства, чтобы лучше понять принцип действия различных каркасов.

В этой главе был сделан лишь беглый обзор системы контроля версий Git, описанию возможностей которой можно посвятить целую книгу. Правда, такая книга уже написана Скоттом Чаконом (Scott Chacon) и Беном Штраубом (Ben Straub). Ее второе издание вышло под названием *Pro Git* в издательстве Apress в 2014 г. Ссылку на нее можно найти на начальной странице веб-сайта Git (`http://git-scm.com/`). Она доступна для чтения в оперативном режиме или в виде загружаемого цифрового файла. А теперь, рассмотрев быстрый способ создания веб-сайта, в следующей главе мы перейдем к обсуждению весьма распространенной библиотеки AngularJS.

ГЛАВА 10

AngularJS и тестирование

В предыдущей главе было показано, как пользоваться современными инструментальными средствами для быстрого создания веб-сайта, а также системой контроля версий для отслеживания отличий во всех рабочих файлах проекта. А в этой главе дается общее представление о принципе действия библиотеки AngularJS.

Если говорить кратко, то назначение этой библиотеки — оказать помощь в написании крупных веб-приложений более организованным и простым способом. К числу других преимуществ этой библиотеки относится более краткий период обучения для коллективной разработки веб-приложений. Как только новый член команды разработчиков освоит эту библиотеку, ему станет понятнее, каким образом действует разрабатываемый веб-сайт. В этой главе мы рассмотрим применение библиотеки AngularJS на конкретном примере.

На момент написания данной книги текущей была версия AngularJS 1.4.1. Дополнительные сведения об этой библиотеке можно найти по адресу `https://angularjs.org/`, а сведения о версии Angular 2 — по адресу `https://angular.io/`.

Одна из задач, которую пытается решить AngularJS, состоит в том, чтобы упростить разработку динамических веб-приложений. Ведь сам язык HTML не предназначен для написания одностраничных веб-приложений. А в AngularJS предоставляется возможность разрабатывать современные веб-приложения, быстро обучаясь этому процессу. С этой целью разработка каждой части ведется отдельно, чтобы она обновлялась независимо от остальных частей приложения, как и предписывает архитектурный шаблон MVC (Model-View-Controller — модель–контроллер–представление). Аналогичным образом действуют и другие библиотеки, в том числе Backbone и Ember.

В главе 9 были представлены некоторые инструментальные средства для веб-производства, помогающие разработчикам работать более продуктивно. Так, в Yeoman (`http://yeomanio/`) применяются встроенные генераторы для быстрого создания файлов и папок, необходимых для того, чтобы сделать веб-сайт работоспособным. А Grunt (`http://gruntjs.com/`) служит для автоматизации задач вроде модульного тестирования и оптимизации файлов перед выпуском готового к эксплуатации веб-сайта. В этой главе предполагается, что оба эти инструментальные средства уже установлены. Подробнее об их установке см. в главе 9.

Чтобы создать новый проект в AngularJS, введите команду **`your angular`** в режиме командной строки. В ответ Yeoman предложит вам ответить на следующие вопросы, касающиеся установки нового проекта.

- **Желаете ли вы пользоваться Sass (вместе с Compass)?** Сокращение Sass (`http://sass-lang.com/`) обозначает Syntactically Awesome Style Sheets — синтаксически превосходные таблицы стилей. Метаязык Sass предоставляет такие возможности, как вложение селекторов и применение переменных, для разработки таблиц стилей. А Compass — это инструментальное средство, написанное на языке Ruby и применяющее файлы Sass для внедрения таких возможностей, как формирование спрайтовых листов из последовательности изображений.

  Ответив на этот вопрос положительно, вы получите SCSS-файл, в котором стили Twitter Bootstrap применяются по умолчанию. А если вы ответите отрицательно, то Yeoman предоставит вам обычный SCSS-файл с теми же самыми CSS-селекторами.

- **Желаете ли вы включить Bootstrap в установку?** Инструментальное средство Twitter Bootstrap (`http://getbootstrap.com/`) поможет вам в разработке пользовательского интерфейса для вашего веб-сайта. С его помощью вы сумеете сделать свой веб-сайт адаптивным, чтобы он мог нормально работать на самых разных устройствах. Для этой цели Twitter Bootstrap предоставляет все необходимые элементы пользовательского интерфейса вроде кнопок и карусели изображений. Если вы решите воспользоваться этим инструментальным средством, Yeoman спросит вас, следует ли выбрать его версию, поддерживающую метаязык Sass.

- **Какие модули вы хотели бы включить в установку?** Модули расширяют возможности AngularJS. Например, модуль `angular-aria` предоставляет специальные возможности, тогда как модуль `angular-route` — возможность вводить средства для создания глубинных ссылок. У вас имеется также возможность ввести или удалить любой из модулей. А в дальнейшем нужные модули могут быть введены вручную.

Как только вы ответите на перечисленные выше вопросы, все нужные файлы будут загружены автоматически. После этого Grunt запустит локальный сервер по стандартному для браузеров адресу загрузки `http://localhost:9000`, как показано на рис. 10.1.

С этого момента вы можете просматривать папки, образующие данный проект. Так, в папке `app` содержится главное приложение, и поэтому мы начнем именно с нее. Папка является обычным явлением для любого HTML-сайта. Но самое интересное начинается с папки `scripts`.

В самой папке `scripts` находится файл `app.js` — это файл главного приложения для AngularJS. Откройте его, чтобы посмотреть, каким образом организована начальная загрузка AngularJS. Как показано на рис. 10.2, главное приложение называется `chapter10app`, поскольку именно так была названа папка, в которой было создано это приложение. Это согласуется с директивой `ng-app` в файле `index.html`. Эта директива предоставляет элементам DOM дополнительные возможности. В данном случае она сообщает библиотеке AngularJS, что приложение начинается именно здесь и называется `chapter10app`.

localhost:9000/#/

Russ

localhost:9000/#/

Apps Phantom Limb LiveCycle Heroku WebRTC PubCrawls iOS DoctorWho » Other Bookmarks

chapter10 Home About Contact

'Allo, 'Allo!

YEOMAN

Always a pleasure scaffolding your apps.

Splendid!

HTML5 Boilerplate

HTML5 Boilerplate is a professional front-end template for building fast, robust, and adaptable web apps or sites.

Angular

AngularJS is a toolset for building the framework most suited to your application development.

Karma

Spectacular Test Runner for JavaScript.

♥ from the Yeoman team

Рис. 10.1. Библиотека AngularJS, действующая через порт **9000**

```
  <link rel="stylesheet" href="styles/main.css">
  <!-- endbuild -->
</head>
<body ng-app="chapter10App">
  <!--[if lt IE 7]>
    <p class="browsehappy">You are using an <stror
  <![endif]-->
```

```
angular
  .module('chapter10App', [
    'ngAnimate',
    'ngAria',
    'ngCookies',
    'ngMessages',
    'ngResource',
```

Рис. 10.2. Директива `ng-app` сообщает библиотеке AngularJS, где находится корневой элемент приложения

Анализируя содержимое файла `app.js`, после имени приложения можно обнаружить целый ряд загружаемых модулей, расширяющих возможности AngularJS. Один из этих модулей называется `ngRoute`. Он позволяет обрабатывать URL для приложения. Так, в методе `.config()` объект `$routeProvider` служит для распознавания URL и загрузки нужного шаблона вместе с контроллером, используя последовательность операторов `when`.

Оператор `when` позволяет специально настраивать URL для приложения. Так, если ввести **`/about`**, то AngularJS будет знать, что следует загрузить шаблон из файла `about.html` и воспользоваться контроллером `AboutCtrl`. Поясним подробнее, что это означает.

Представления и контроллеры

Как обсуждалось ранее, в AngularJS, как и в других библиотеках, применятся шаблон проектирования MVC. Это означает, что разрабатываемое приложение разделяется на следующие три части.

- **Модель** — служит для хранения данных, предназначаемых для приложения.
- **Представление** — служит для воспроизведения данных из модели, например, для построения диаграммы, представляющей эти данные.
- **Контроллер** — служит для отправки команд модели с целью обновить данные, а также для отправки команд представлению с целью обновить воспроизведение данных из модели.

В папке `views` содержатся HTML-шаблоны, которые могут быть обновлены данными, поступающими из модели, а в папке `controllers` — файлы JavaScript, предназначенные для обмена данными с файлами модели и представления. Рассмотрим файл `about.html`, в котором предстоит ввести кнопку для работы с ней контроллера. С этой целью откройте файл `about.html`, находящийся в папке `views`. Введите в этом файле дескриптор кнопки. В этом дескрипторе мы собираемся употребить еще одну директиву, чтобы дать AngularJS знать, когда произведен щелчок на кнопке.

Нам нужно ввести директиву `ng-click` в разметку кнопки. С этой целью введите **`ng-click='onButtonClick()'`**, как показано в листинге 10.1. Эта директива будет разрешена контроллером, чтобы визуально отделить представляемые части приложения от бизнес-логики.

Листинг 10.1. Определение метода с помощью директивы `ng-click`

```
<button ng-click="onButtonClick()">Button</button>
```

Откройте файл `about.js` в папке `controllers`. Контроллеры дают возможность ввести всю бизнес-логику, которая требуется для нормальной работы данной части приложения. Обратите внимание на имя `AboutCtrl` в методе `controller()`, которое совпадает с тем, что мы наблюдали в файле `app.js`, а также на свойство `$scope`.

Свойство `$scope` позволяет вводить методы или свойства в рабочее представление. В данном случае мы имеем дело с функцией, объявленной в HTML-файле. Директива `ng-click="onButtonClick()"` была определена в коде HTML, и поэтому она входит в область действия данного контроллера. В листинге 10.2 показано, каким образом организуется совместная работа контроллера и HTML кода.

Листинг 10.2. Применение контроллера для определения функции, объявленной в HTML-файле

```
angular.module('chapter10App')
.controller('AboutCtrl', function ($scope, $http) {
  $scope.awesomeThings = [
    'HTML5 Boilerplate',
    'AngularJS',
    'Karma'
  ];
  $scope.onButtonClick = function(){
    console.log('hello world');
  };
});
```

Если приложение выполняется в браузере, то в нем должно быть замечено, что файл JavaScript был обновлен, а следовательно, нужно все перекомпилировать. Как только это будет сделано, щелкните на кнопке и посмотрите сообщение, появляющееся на консоли. В противном случае перейдите к главной папке своего приложения и введите команду **`grunt serve`** в режиме командной строки.

Это лишь начало того, что можно сделать, чтобы отделить представление приложения от его бизнес-логики. Что, если, например, требуется отобразить список элементов? Вернувшись к контроллеру, мы можем ввести в область его действия объект формата JSON, как показано в листинге 10.3.

Листинг 10.3. Данные, назначенные для области действия контроллера `about`

```
$scope.bands = [
  {'name':"Information Society", 'album':"_hello world"},
  {'name':"The Cure", 'album':"Wish"},
  {'name':"Depeche Mode", 'album':"Delta Machine"}];
```

А теперь, когда имеются нужные нам данные, их необходимо передать HTML-шаблону. Вернитесь к файлу `about.html`, чтобы воспользоваться директивой `ng-repeat`, как демонстрируется в листинге 10.4.

Листинг 10.4. Директива `ng-repeat` для перебора данных, предоставляемых контроллером для отображения правильного количества элементов в списке

```
<ul>
  <li ng-repeat="band in bands">{{band.name}}<p>{{band.album}}</p></li>
</ul>
```

На данном этапе веб-страница должна выглядеть так, как показано на рис. 10.3.

До сих пор нам удалось связать контроллер с элементами, определенными в HTML-представлении с помощью директив. Данные были также определены в контроллере. Но допустим, что требуется получить данные из внешнего источника. Как в таком случае вызвать удаленный сервер и отобразить результаты? Попробуем ответить на этот вопрос в следующем разделе.

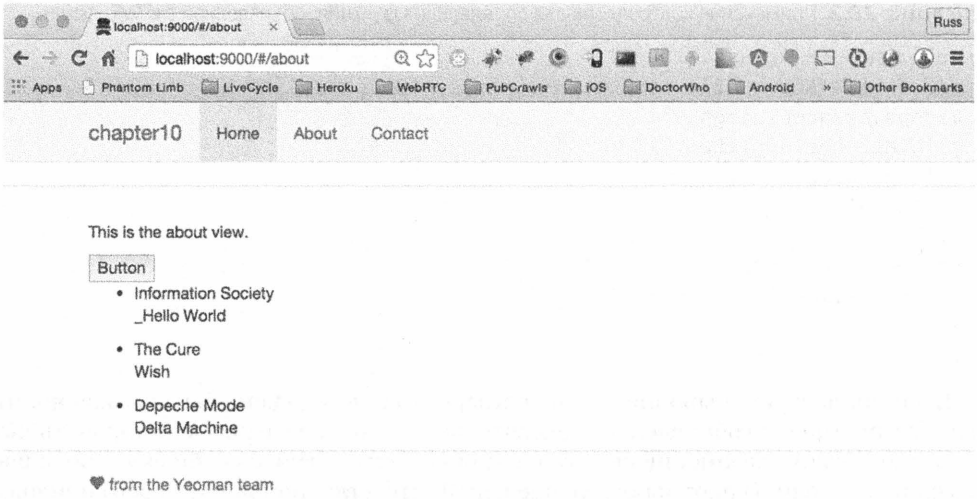


Рис. 10.3. Данные, полученные из удаленного контроллера и отображаемые на странице

Удаленные источники данных

Рассмотрим упомянутый ранее метод обращения с кнопкой для получения удаленных данных. В данном случае мы собираемся воспользоваться службой `$http`, предназначенной для автоматической обработки удаленных вызовов. Это аналогично AJAX-методу из библиотеки JQuery. Чтобы выгодно воспользоваться данной службой, нужно сначала ввести ее в метод `controller()`, как показано в листинге 10.5.

Листинг 10.5. Ввод службы `$http` в контроллер `AboutCtrl`

```
.controller('AboutCtrl', function($scope, $http){
  $scope.awesomeThings = [
    'HTML5 Boilerplate',
    'AngularJS',
    'Karma'
  ];
});
```

А теперь, когда служба доступна для контроллера, мы можем воспользоваться ею в методе `onButtonClick()`. Мы собираемся также воспользоваться методом `get()`, соответствующим методу `GET` из сетевого протокола HTTP. Подробнее о методах сетевого протокола HTTP можно узнать из статьи Википедии по следующему адресу:

`https://ru.wikipedia.org/wiki/HTTP#Request_methods`

Мы воспользуемся также REST-службой `JSONPlaceholder`, которая позволяет тестировать HTTP-запросы и возвращать фиктивные данные. С этой целью удалите код, присутствующий в теле метода `onButtonClick()`, и введите вместо него код, приведенный в листинге 10.6.

Листинг 10.6. Вызов по HTTP-запросу и присваивание результатов свойству `results`

```
$http.get('http://jsonplaceholder.typicode.com/photos').
          success(function(data){
  $scope.results = data;
});
```

Как видите, в этом коде применяется метод `get()`, а кроме того, принимается событие `success`. Если вызов по HTTP-запросу окажется успешным, то для присваивания полученного результата свойству `results` вызывается анонимная функция.

Получив результаты запроса, мы можем обновить шаблон, чтобы отобразить не только текст, но и миниатюрные изображения, возвращаемые из службы. И для этого мы воспользуемся еще одной директивой для дескриптора `<image>`. С этой целью откройте файл `about.html` и обновите существующий в нем код, удалив предыдущий список и введя в шаблон функциональный код, приведенный в листинге 10.7.

Листинг 10.7. Создание неупорядоченного списка на основании результатов, возвращенных из REST-службы

```
<ul>
  <li ng-repeat="result in results">
    <p>ID: {{result.id}}</p>
    <p>{{result.title}}</p>
    <p><img ng-src="{{result.thumbnailUrl}}"/></p>
  </li>
</ul>
```

В данном случае директива `ng-src` используется для того, чтобы указать в дескрипторе `<image>`, где следует искать каждое изображение из списка. Как и в предыдущем примере, это потребует перебора всего списка результатов и их вывода на экран. Примечательная особенность применения данной директивы состоит в том, что в дескрипторе `<image>` не предпринимается попытка воспроизвести изображение до тех пор, пока данные не будут получены. Поэтому нужно лишь возвратить результаты из REST-службы. На данном этапе веб-страница должна выглядеть так, как показано на рис. 10.4.

Итак, предприняв лишь несколько шагов, мы получили веб-сайт, способный извлекать данные из удаленного источника и воспроизводить их в окне браузера по мере надобности. В библиотеке AngularJS имеется немало других любопытных средств, и мы рассмотрим далее одно из них. В следующем разделе речь пойдет о маршрутах.

Маршруты

Маршруты позволяют составлять специальные URL для разрабатываемого веб-приложения и дают возможность сделать закладку на страницу, чтобы непосредственно перейти на нее в дальнейшем. Пример тому мы уже рассматривали в файле `app.js`. Так, в методе `.config()` применяется прикладной программный интерфейс API объекта `$routeProvider`. А в ряде операторов `when` можно определить загружаемый HTML-шаблон и применяемый вместе с ним контроллер.

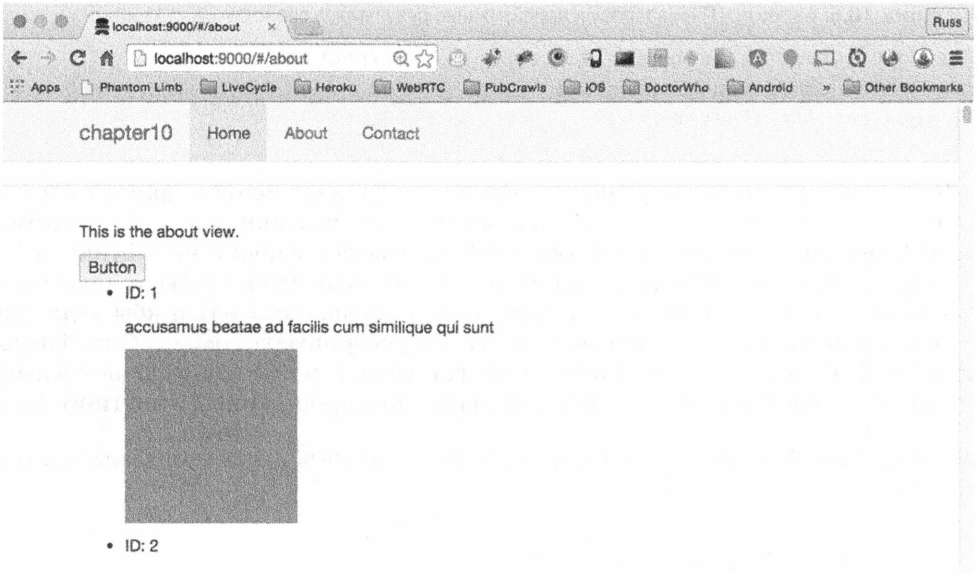


Рис. 10.4. Отображение результатов, возвращенных из REST-службы

Рассмотрев этот механизм, выясним, как передать параметры приложению. Допустим, что требуется показать лишь одну рассылку из предыдущего примера. С этой целью создадим маршрут. Если вы разрабатывали свое приложение средствами Yeoman, как пояснялось в предыдущих разделах, то перейдите в режим командной строки и введите следующую команду:

```
yo angular:route post
```

По этой команде Yeoman обновит файл `app.js`, введя в него новый маршрут. Кроме того, будет создан новый файл JavaScript для контроллера и HTML-файл для представления, что совсем не плохо для одной команды. Если приложение выполняется, перейдите в окно браузеров и введите следующий адрес:

```
http://localhost:9000/#/post
```

В итоге появится представление рассылки, готовое к применению. В данном случае преследуется цель загрузить рассылку по номеру, введенному в URL. Так, если URL выглядит следующим образом:

```
http://localhost:9000/#/post/4
```

то на экране должна появиться четвертая по списку рассылка, полученная с сервера, использовавшегося в последнем примере.

Параметры маршрута

Чтобы привести все это в действие, нужно сделать функцию маршрутизации более гибкой. С этой целью откройте файл `app.js`. Нам нужно обновить маршрут рассылки таким образом, чтобы он принимал переменные в URL. Поэтому замените в нем **`/post`** на **`/post/:postId`**.

Введя в маршрут параметр `:postId`, мы создаем переменную, которой можно воспользоваться в контроллере. Эта переменная будет представлять номер рассылки в URL. Чтобы продемонстрировать это, обновим контроллер.

Откройте файл `post.js`, где в теле метода `.controller()` имеется анонимная функция, использующая свойство `$scope`. Как было показано в предыдущих примерах, свойство `$scope` дает возможность управлять HTML-шаблоном. Мы введем в эту функцию дополнительный параметр `$routeParams`, чтобы получить доступ к переменной в URL.

Теперь мы можем извлечь переменную из URL и присвоить ее содержимое свойству `$scope`. Это даст нам возможность отобразить его при обновлении шаблона. С этой целью введите в теле метода `.controller()` следующую строку кода:

```
$scope.postId = $routeParams.postId;
```

Чтобы увидеть номер рассылки на экране, мы можем быстро обновить шаблон. Для этого откройте файл `post.html` в папке `views`. Удалите сначала копию между дескрипторами `<p>`, чтобы она выглядела следующим образом:

```
<p>{{postId}}</p>
```

Сделав это, введите номер рассылки в URL, чтобы увидеть его на экране. На данном этапе окно браузера должно выглядеть так, как показано на рис. 10.5.

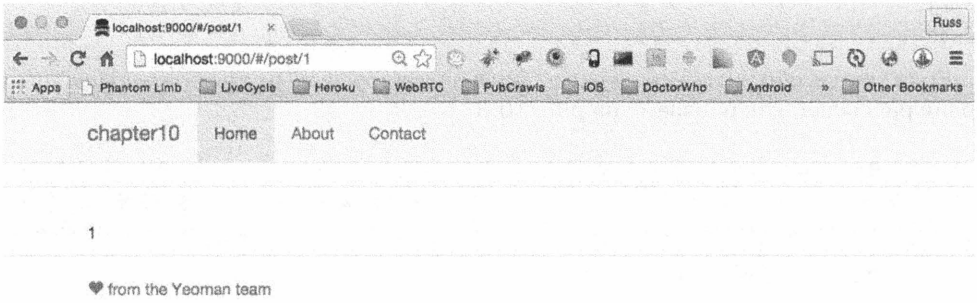


Рис. 10.5. Вывод на экран содержимого переменной из URL

Совсем не плохо. А теперь нам нужно лишь связать параметры маршрута с запросом по методу `GET` и отобразить результаты. В данном случае код будет похож на тот, что был написан прежде. В управление рассылкой нам нужно ввести службу `$HTTP`, чтобы сделать вызов по REST-запросу. Соответствующий код приведен в листинге 10.8.

Листинг 10.8. Полный контроллер `PostCtrl` с введенными службами `$routeParams` и `$http Services`

```
.controller('PostCtrl', function($scope, $routeParams, $http){
  $scope.awesomeThings = [
    'HTML5 Boilerplate',
    'AngularJS',
    'Karma'
  ];
});
```

Сразу после этого мы сделаем тот же самый вызов по REST-запросу, что и прежде. Но на этот раз добавим переменную `postId`, как показано в листинге 10.9.

Листинг 10.9. Получение единого результата с помощью служб `$http` и параметров маршрута

```
$http.get(
          'http://jsonplaceholder.typicode.com/photos/'+
          $routeParams.postId).success(function(data){
  $scope.results = data;
});
```

Что же касается HTML-шаблона, то воспользуемся тем же самым кодом, что и в примере с шаблоном `about`. Но на этот раз удалим директиву `ng-repeat` и употребим свойство `results` (листинг 10.10).

Листинг 10.10. HTML-шаблон для отображения рассылки

```
<ul>
  <li>
    <p>ID: {{results.id}}</p>
    <p>{{results.title}}</p>
    <p><img ng-src='{{results.thumbnailUrl}}'/></p>
  </li>
</ul>
```

Если теперь обновить номер рассылки в поле адреса, то в окне браузера появится новая рассылка, как показано на рис. 10.6

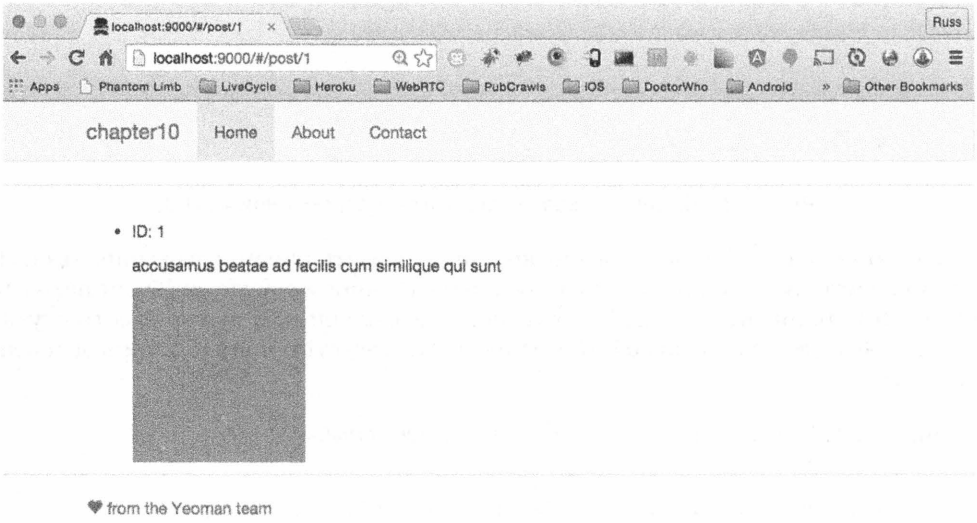


Рис. 10.6. Отображение единственной рассылки, исходя из переменной в URL

Тестирование приложения

Из документации на AngularJS можно узнать, как писать тесты, покрывающие разные части разрабатываемого веб-приложения. Тестирование помогает убедиться в том, что написанный код работает все время устойчиво. Так, если требуется внести изменение в функцию, она должна по-прежнему возвращать тот же самый результат, несмотря на это изменение. Если же подобное изменение приводит к неожиданному результату в какой-нибудь другой части приложения, то тест должен это выявить.

Принимая все это во внимание, возникает вопрос: как же протестировать веб-приложение, разработанное средствами AngularJS? Для того чтобы ответить на этот вопрос, нужно рассмотреть следующие два вида тестирования: *модульное* и *сквозное*.

Модульное тестирование

При написании функции во внимание принимаются возвращаемые ею параметры, обрабатываемые ею данные и возвращаемый результат. Имея модульный тест для проверки этого кода, можно убедиться в том, что он ведет себя именно так, как и предполагалось.

Если вы пользовались до сих пор версией AngularJS, сформированной средствами Yeoman, вам нужно установить ряд дополнительных компонентов, чтобы выполнять тестирование. С этой целью перейдите в режим командной строки и введите следующую команду:

```
npm install grunt-karma --save-dev
```

В итоге будет установлен компонент Karma. Этот компонент в документации на AngularJS описывается как "инструментальное средство на JavaScript, работающее в режиме командной строки и предназначенное для порождения веб-сервера, загружающего исходный код приложения и выполняющего тесты". Короче говоря, Karma запускает браузеры, выполняет код по написанным тестам и выдает результаты. Если у вас нет браузера для тестирования (например, браузера Internet Explorer под Mac OS), воспользуйтесь такой службой, как BrowserStack (`https://www.browserstack.com/`) или Sauce Labs (`https://saucelabs.com/`). Самые последние сведения о Karma можно найти по адресу `http://karma-runner.github.io/`.

Далее нужно установить компонент PhantomJS. С этой целью введите в режиме командной строки следующую команду:

```
npm install karma-phantomjs-launcher --save-dev
```

PhantomJS — это "обезглавленный" браузер (т.е. браузер без пользовательского интерфейса). Установив его, можно выполнять приложение в окне браузера, а все команды — в режиме командной строки. Самые последние сведения о PhantomJS можно найти по адресу `http://phantomjs.org/`. И наконец, введите следующую команду:

```
npm install karma-jasmine --save-dev
```

В итоге будет установлен компонент Jasmine — среда для тестирования рассматриваемого здесь приложения. Дополнительные сведения о Jasmine можно найти по адресу `http://jasmine.github.io/`.

А теперь убедитесь, что все работает исправно, введя команду **grunt test**. В итоге должны быть выполнены тесты, находящиеся в папке test.

Ввод новых тестов

Одно из преимуществ применения Yeoman заключается в том, что вместе с новыми контроллерами создаются соответствующие файлы для модульного тестирования. Если заглянуть в главную папку, то в ней можно обнаружить папку spec, содержащую папку controllers, где находятся файлы для модульного тестирования каждого созданного контроллера. Откройте файл about.js, чтобы выяснить, как тестируется контроллер.

Прежде чем приступить к написанию тестов, рассмотрим сначала существующий код, чтобы стало понятнее, о чем идет речь. В самом начале этого кода находится метод describe(), который служит для описания тестов, которые предстоит написать. С помощью этого метода можно описать на высоком уровне все, что требуется протестировать.

Далее следует метод beforeEach(), который должен выполняться перед каждым тестом. Это позволяет получить доступ ко всему приложению, загрузив его как отдельный модуль.

После этого создаются две переменные, AboutCtrl и scope, и еще один метод beforeEach(), где этим переменным присваиваются значения контроллера, а также область его действия, как если бы он применялся непосредственно. И наконец, мы можем приступить к написанию тестов, описываемых в последовательном ряде методов it(). Это позволяет упростить документирование тестов, где описывается, каким образом должна работать тестируемая функция.

Тест по умолчанию выводит сообщение "should attach a list of awesomeThings to the scope" (список из массива awesomeThings должен быть присоединен к области действия), затем он выполняет проверяемую функцию с помощью метода expect(). Этот метод важен, поскольку он дает возможность протестировать предполагаемый результат. В данном случае мы проверяем длину массива awesomeThings, предполагая, что она окажется равной 3. В противном случае тест не пройдет.

А теперь мы можем протестировать длину созданного ранее массива bands. С этой целью введите новый метод it(), как показано в листинге 10.11.

Листинг 10.11. Модульный тест количества музыкальных ансамблей, которые должны быть в области действия контроллера about

```
it('should have at least 3 bands', function(){
  expect(scope.bands.length).toBe(3);
});
```

Если перейти в режим командной строки и ввести команду **grunt test**, этот тест должен пройти. Если же в методе expect() окажется другое количество элементов массива, например 2, этот тест не пройдет (рис. 10.7).

Мы можем также проверить значение элемента в массиве, как показано в листинге 10.12.

```
Running "karma:unit" (karma) task
WARN [watcher]: Pattern "/Users/asciibn/Documents/Projects/apress/proJavaScript/chapter12/angularSite/test/mock/**/*.js" does not match any file.
INFO [karma]: Karma v0.12.36 server started at http://localhost:8080/
INFO [launcher]: Starting browser PhantomJS
INFO [PhantomJS 1.9.8 (Mac OS X 0.0.0)]: Connected on socket pU5zBpH8QaUpv-xK1ndf with id 47251687
PhantomJS 1.9.8 (Mac OS X 0.0.0) Controller: AboutCtrl should have at least 3 bands FAILED
        Expected 3 to be 2.
            at /Users/asciibn/Documents/Projects/apress/proJavaScript/chapter12/angularSite/test/spec/controllers/about.js:24
PhantomJS 1.9.8 (Mac OS X 0.0.0): Executed 4 of 4 (1 FAILED) (0.002 secs / 0.017 secs)
Warning: Task "karma:unit" failed. Use --force to continue.

Aborted due to warnings.


Execution Time (2015-06-14 02:11:50 UTC)
wiredep:app          168ms  [=]  2%
concurrent:test        7.8s  [==========================================================]  92%
autoprefixer:server  307ms  [=]  3%
karma:unit             1.1s  [=========]  12%
Total 9.6s

Russs-MBP:angularSite asciibn$
```

Рис. 10.7. В тесте ожидались три элемента массива, а фактически получено два

Листинг 10.12. Модульный тест для проверки значения в массиве

```
it('should have the second album be Wish', function(){
  expect(scope.bands[1].album).toEqual('Wish');
});
```

Это был очень беглый обзор процесса тестирования контроллера. Тем не менее он дает ясное представление о том, как писать и выполнять тестовые методы для оценки получаемых результатов. Jasmine дает немало возможностей проверить надежность написанного кода. А теперь рассмотрим тестирование под другим углом зрения, открыв файл `post.js` и протестировав HTTP-запрос.

Тестирование HTTP-запросов с помощью службы `$httpBackend`

В предыдущем примере мы проверяли данные, которые были связаны с контроллером. И в том случае источник данных находился в самом контроллере. Но в большинстве веб-приложений данные получаются из удаленного источника. Так как же написать тест, не имея контроля над источником данных? С этой целью можно воспользоваться службой `$httpBckend`, чтобы убедиться в том, что составляемые запросы действуют независимо от службы. В таком тесте должно быть воссоздано все, что мы раньше делали со службой `$routeParams`. Этот тест должен быть автономным и фактически не обращаться к серверу.

Прежде всего введем переменные `httpBackend` и `routeParams`, помимо переменных `PostCtrl` и `scope`. В данном случае мы не ссылаемся на директивы, и поэтому в именах переменных отсутствует знак `$`.

В настоящий момент контроллер инициализируется во втором методе `beforeEach()`. Именно здесь мы и введем директивы, как в настоящем контроллере, для ссылки на службы `$httpBackend` и `$routeParams`.

А теперь присвоим значения созданным ранее переменным. В браузере мы могли бы получить единственную рассылку, присвоив значение переменной `postId` из URL. Эта операция имитируется, как показано в листинге 10.13.

Листинг 10.13. Присваивание значений переменным для имитации получения значения из браузера (часть 1)

```
beforeEach(inject(function(
          $controller,$rootScope,$httpBackend,$routeParams){
  scope = $routeScope.$new();
  routeParams = $routeParams;
  routeParams.postId = 1;
  httpBackend = $httpBackend;
  httpBackend.expectGet(
    'http://jsonplaceholder.typicode.com/photos/'+routeParams.postId).
    respond({id:'1', title:'title 1', thumbnailUrl:'myImage.png'});
  PostCtrl = $controller('PostCtrl', {
  $scope: scope
});
httpBackend.flush();
});
```

Мы не загружаем этот код в браузер, чтобы проверить его работоспособность, и поэтому значение 1 жестко закодировано в переменной `postId`. С помощью переменной `httpBackend` мы имитируем вызов, который делается в контроллере. Но в данном случае мы пользуемся методом `expectGet()`, чтобы сымитировать HTTP-запрос по методу `GET`. Если потребуется сделать HTTP-запрос по методу `POST`, то для его имитации следует воспользоваться методом `expectPost()`.

Метод `respond()` предоставляет полезную информацию для тестирования. В данном случае возвращается простой объект, аналогичный предоставляемому прикладным программным интерфейсом API. После назначения области действия объект `httpBackend` снова используется для вызова метода `flush()`. Благодаря этому ответ будет доступен для тестирования, как и при настоящем вызове по HTTP-запросу.

А теперь перейдем к тестам. Как и в другом примере, последовательный ряд методов `it()` служит для описания предполагаемого результата. Убедимся сначала, что с сервера получается лишь один результат, как показано в листинге 10.14.

Листинг 10.14. Присваивание значений переменным для имитации получения значения из браузера (часть 2)

```
it('should be a single post', function(){
  expect(scope.results).not.toBeGreaterThan(1);
});
```

Среда Jasmine упрощает чтение результатов тестирования. Мы получили результаты и хотим убедиться в наличии лишь одного объекта. Если бы нам требовалось проверить, что в данном объекте имеется свойство `ID`, мы воспользовались бы кодом, приведенным в листинге 10.15.

Листинг 10.15. Проверка свойства `ID`

```
it('should have an id', function(){
  expect(scope.results.id).toBeDefined();
}
```

Как и прежде, мы можем ввести тесты, которые позволят нам лучше понять контроллер по мере ввода в него дополнительных функциональных возможностей. Методика написания тестов перед созданием прикладного кода называется *разработкой посредством тестирования*. По такой методике сначала пишется тест, который заведомо не пройдет, а затем пишется минимальный объем кода для того, чтобы тест прошел. После этого может потребоваться реорганизация кода. Среда Jasmine служит не только для модульного тестирования кода, но и для интеграции тестов в браузер. Так как же сымитировать щелчки на кнопках в разных браузерах? Ведь из истории развития веб-разработки известно, что даже то, что поначалу казалось простым, в конечном итоге не действовало в браузерах. И здесь на помощь приходит среда тестирования Protractor.

Сквозное тестирование в среде Protractor

Среда Protractor (`http://angular.github.io/protractor`) позволяет выполнять тесты в настоящих браузерах. Например, в ней можно проверить, что кнопка действительно была нажата при передаче формы на рассмотрение. В отличие от модульного тестирования, где проверяются небольшие блоки кода, сквозное тестирование позволяет проверять целые части приложения относительно HTTP-сервера. Это все равно, что открыть веб-сайт в окне браузера и убедиться в его работоспособности. Но самое главное, что процесс сквозного тестирования автоматизирован.

Подобные задачи должны быть автоматизированы. И среда Protractor позволяет это сделать, поскольку она построена на платформе WebDriver, предназначенной для автоматизации тестирования в браузере. В среде Protractor поддерживается также библиотека AngularJS, что потребует незначительной ее настройки. Итак, установите среду Protractor для выполнения ряда тестов, введя следующую команду:

```
npm install -g protractor --save-dev
```

По этой команде среда Protractor устанавливается глобально, чтобы ею можно было пользоваться в разных проектах. Для приведения этой среды тестирования в рабочее состояние потребуется незначительная конфигурация. С этой целью создайте новый файл конфигурации под именем `protractor.conf.js` и сохраните его в папке `test` рядом с файлом `karma.config.js`. В этом файле предоставляются сведения о том, где среда Protractor должна искать сервер Selenium, какие браузеры нужно запускать на выполнение и где находятся тестовые файлы. Код для этого файла конфигурации приведен в листинге 10.16.

Листинг 10.16. Элементарное содержимое файла конфигурации Protractor

```
export.config = {
  seleniumAddress: 'http://localhost:4444/wd/hub',
  multiCapabilities: [{browserName: 'firefox'},{browserName: 'chrome'}],
  baseUrl: 'http://localhost.9000',
  framework: 'jasmine',
  specs: ['protractor/*.js']
};
```

Приведенный выше код требует некоторых пояснений. В частности, свойство `seleniumAddress` указывает среде Protractor место, где выполняется сервер Selenium, а свойство `multiCapabilities` — типы браузеров для выполнения тестов. Как видите, это свойство содержит массив объектов, где перечислено имя каждого браузера.

Мы собираемся выполнять тестирование локально, поэтому можем делать это только в тех браузерах, которые установлены на компьютере. Следовательно, если вы работаете на компьютере под Mac OS, то не сможете выполнять тестирование в браузере Internet Explorer. Если же вам потребуется выполнить тестирование в браузерах, подобных Internet Explorer, или в мобильных браузерах, введите в файл конфигурации дополнительные свойства, которые позволяют связываться со службой SauceLabs или BrowserStack.

Далее следует свойство `baseUrl`, которое указывает среде Protractor сервер для размещения проверяемого приложения. Очень важно, чтобы веб-приложение находилось на локальном сервере при выполнении тестов. В свойстве `framework` устанавливается среда Jasmine, поскольку именно она применяется в рассматриваемых здесь тестах. Свойство `specs` играет важную роль потому, что в нем указывается папка, где хранятся тесты. В данном случае это папка `protractor`. А метасимвол указывает на то, что в этой папке следует искать любой файл JavaScript.

Итак, мы настроили среду Protractor, и теперь можно приступить к написанию тестов. С этой целью создайте сначала папку `Protractor`, а затем файл `app-spec.js`. Его содержимое будет очень похоже на то, что мы уже делали в предыдущих примерах.

Начнем с метода `describe()` для описания набора выполняемых тестов. После этого метода сразу же следует ряд методов `it()`, как показано в листинге 10.17. Ради простоты воспользуемся примерами, взятыми непосредственно с веб-сайта Protractor.

Листинг 10.17. Метод `it()` для проверки наличия заголовка у веб-сайта

```
it('should have a title', function(){
  browser.get('http://juliemr.github.io/protractor-demo');
  expect(browser.getTitle()).toEqual('Super Calculator');
});
```

Теперь у нас есть все, что требуется для тестирования, которое мы будем выполнять в режиме командной строки. Если вы еще не запустили задачу `serve` на выполнение и не просматриваете проверяемый сайт локально, введите следующую команду:

```
grunt serve
```

Это даст возможность запустить проверяемый сайт на локальном сервере через порт **9000**. Именно там мы предписали среде Protractor искать тесты для выполнения. А теперь введите следующую команду:

```
protractor test/protractor.conf.js
```

По этой команде будет найдена папка `test`, выполнен файл конфигурации и запущен на выполнение браузер для проведения тестов. В итоге тесты должны пройти, как показано на рис. 10.8.

В итоге браузер будет направлен на проверяемый сайт по указанному URL и произведена проверка его заглавия. Как видите, все довольно просто. А теперь напишем несложный тест, где складываются два значения и проверяется результат. Код этого теста приведен в листинге 10.18.

```
Russs-MacBook-Pro:angularSite asciibn$ protractor test/protractor.conf.js
[launcher] Running 2 instances of WebDriver
.
------------------------------------
[chrome #2] PID: 16221
[chrome #2] Specs: /Users/asciibn/Documents/Projects/apress/proJavaScript/chapter12/angularSite/test/protractor/app-spec.js
[chrome #2]
[chrome #2] Using the selenium server at http://localhost:4444/wd/hub
[chrome #2] .
[chrome #2]
[chrome #2] Finished in 1.301 seconds
[chrome #2] 1 test, 1 assertion, 0 failures
[chrome #2]

[launcher] 1 instance(s) of WebDriver still running
.
------------------------------------
[firefox #1] PID: 16220
[firefox #1] Specs: /Users/asciibn/Documents/Projects/apress/proJavaScript/chapter12/angularSite/test/protractor/app-spec.js
[firefox #1]
[firefox #1] Using the selenium server at http://localhost:4444/wd/hub
[firefox #1] .
[firefox #1]
[firefox #1] Finished in 1.535 seconds
[firefox #1] 1 test, 1 assertion, 0 failures
[firefox #1]

[launcher] 0 instance(s) of WebDriver still running
[launcher] chrome #2 passed
[launcher] firefox #1 passed
Russs-MacBook-Pro:angularSite asciibn$
```

Рис. 10.8. Тесты, выполняемые в среде Protractor, проходят в обоих браузерах, Firefox и Chrome

Листинг 10.18. Ввод значений в двух полях и проверка результата их сложения

```
describe('Protractor Demo App', function() {
  it('should add one and two', function() {
    browser.get('http://juliemr.github.io/protractor-demo/');
    element(by.model('first')).sendKeys(1);
    element(by.model('second')).sendKeys(2);

    element(by.id('gobutton')).click();

    expect(element(by.binding('latest')).getText()).
    toEqual('3');
  });
});
```

В данном тесте мы можем обратиться к директиве `ng-model` из каркаса AngularJS, чтобы получить доступ к текстовым полям и ввести в них значения. Затем мы можем найти кнопку по ее идентификатору и щелкнуть на ней. В результате этого щелчка вызывается метод `doAddition()`. И наконец, мы можем проверить значение, обновляемое по результату, возвращаемому данным методом.

Резюме

В этой главе был сделан довольно беглый обзор библиотеки AngularJS и процесса написания модульных и сквозных тестов. Обе эти темы заслуживают отдельных

книг. По мере того как ваши проекты будут становиться все крупнее, а вы будете все больше вовлекаться в их разработку, наличие среды тестирования поможет вам вести проекты организованно. Помимо возможности тестировать свой код, вы приобретете большую уверенность в его надежности. Модульное тестирование позволяет выяснить, работает ли клиентский код именно так, как и предполагалось, а интеграция тестов — согласованность работы проверяемого кода в разных браузерах.

На веб-сайте Year of Moo (`http://www.yearofmoo.com/tags/AngularJS.html`) имеется превосходная подборка статей по тестированию и отладке кода в AngularJS. В этих статьях рассматриваются следующие вопросы: когда следует писать тесты, как проводить тестирование в прежних браузерах, что следует и чего не следует тестировать.

По результатам тестирования можно с уверенностью реорганизовать код, зная, что это не повредит приложению. А если и повредит, то об этом быстро станет известно.

ГЛАВА 11

Перспективы развития JavaScript

Итак, мы совершили подробный экскурс в JavaScript. Очевидно, что развитие языка JavaScript переживает переходный период. От скромного начала в какой-то мере игрушечного языка JavaScript дорос до языка уровня крупных проектов. В ходе этого процесса начали проявляться недостатки, что отчасти привело к потерям. Когда же разработчики, имевшие опыт программирования на более зрелых языках, перешли на JavaScript, их нередко удивляло, чего удалось добиться, несмотря на ограничения, присущие JavaScript. Им было также любопытно, как этот язык будет совершенствоваться дальше.

Окончив "выпускной класс" в 1995 году, JavaScript достиг уровня самых передовых до сих пор языков программирования, в том числе Java, Ruby, PHP и ColdFusion. Многие разработчики скажут, что "одноклассники" языка JavaScript продвинулись намного вперед по сравнению с ним. Тем не менее многие из них заимствовали свои свойства у JavaScript, глядя на применение в нем прототипов, реализацию функций как основных языковых средств, гибкость стиля программирования, что вдохновило на внедрение в них собственных новых средств.

Какое же будущее ожидает язык JavaScript? В каком направлении пойдет его дальнейшее развитие? Правда, в отличие от неясности и смутности нашего личного будущего, будущее языка JavaScript и перспективы развития его спецификаций просматриваются ясно. Стандарт ECMAScript 6, скорее всего, будет принят полностью, хотя он еще не был реализован на момент написания данной книги, тогда как стандарт ECMAScript 7 уже находится на стадии разработки и обсуждения. Таким образом, перспективы развития JavaScript выглядят блестяще.

В этой главе мы рассмотрим сначала, что ожидает язык JavaScript. Мы вкратце обсудим текущее состояние этого языка и перспективы его дальнейшего развития, рассмотрев процесс разработки и принятия стандартов. Затем мы выясним, что же требуется сделать, чтобы пользоваться JavaScript вместе с текущим набором инструментальных средств. А остальную часть этой главы мы посвятим подробному обсуждению стандарта ECMAScript 6, где предусмотрены языковые средства, которыми разработчикам, скорее всего, придется пользоваться в ближайшие несколько лет. Мы попытаемся даже заглянуть в отдаленное будущее, которое может настать, а может быть, и нет...

Разумеется, в одной главе невозможно рассмотреть спецификацию ECMAScript 6 во всех подробностях. Поэтому мы решили остановиться на наиболее полезных, лучше определенных и самых интересных языковых средствах.

Прошлое, настоящее и будущее JavaScript

Начнем с того, что нам уже известно. Европейская ассоциация производителей компьютеров, называемая также Ecma International, является органом, надзирающим за разработкой стандартов, которых придерживается язык JavaScript. О том, как до этого дошло, можно было, конечно, написать отдельную книгу, но вряд ли данная тема заинтересует программирующих на JavaScript. Важнее другое: в организации Ecma International образовался технический комитет TC39, который подхватил знамя стандартизации JavaScript и предает гласности их регулярные обновления. Но еще важнее, что различные производители вычислительной техники, программного обеспечения и прочие заинтересованные стороны возложили большие надежды на стандарты. А сообщество программирующих на JavaScript обладает жизнеспособной системой, чтобы направлять дальнейшее развитие этого языка в нужное русло. Это должно помочь в устранении враждебных различий, с которыми нам приходилось бороться в прошлом, а также рационализировать процесс становления JavaScript как эффективного языка уровня крупных проектов.

Процесс приведения JavaScript к эффективно управляемому стандарту начался давно. Многие языковые средства, которые теперь считаются стандартными, появились в версии 3 стандарта ECMAScript. Эта версия возникла в те времена, когда организация Ecma International все еще шла вдогонку производителей браузеров. И хотя предпринимались усилия создать четвертую версию этого стандарта, они так ничем и не завершились. Через десять лет после выпуска версии 3 в 2009 году была предложена версия 5 данного стандарта. Она была нацелена на установление нового положения дел, учет изменений, происходивших в области программирования, а также уточнение многих неясностей, существовавших в версии 3. Эта версия стандарта была широко принята и помогла прояснить дальнейший путь возобновления обязательств, взятых организацией Ecma International по управлению стандартами на JavaScript.

Тем не менее широкое принятие версии 5 стандарта ECMAScript так и не принесло ожидаемых результатов. Так, в браузере Internet Explorer он был реализован лишь в версии 9, но значительная часть мира до сих пользуется версией 8 и даже более ранними версиями. Для организации Ecma International и технического комитета TC39 труднее было не столько установить стандарт, сколько убедить целевую аудиторию переходить на новые стандарты, как только они предлагаются.

Было не совсем ясно, каким путем следовать после версии 5. В конечном итоге были выбраны два пути. Во-первых, появилась версия 5.1 стандарта ECMAScript, приведенная в соответствие со спецификацией Международной организации по стандартизации (ISO) на этот стандарт, что само по себе имело долгую и скучную историю. И во-вторых, ожидается новая версия 6 стандарта ECMAScript, нередко называемая ECMAScript Harmony (это название возникло в связи с разнообразными предложениями, которые со временем были гармонизированы в различных стандартах JavaScript, ECMAScript, JScript и т.д., а также происходит от исходного кодового названия ECMAScript в версии 4).

В новом стандарте программирующих на JavaScript больше всего интересует то, что он позволяет им делать. Предложенный вариант ES6/Harmony должен был быть подготовлен к средине 2015 года. Впрочем, оставим область стандартов и лучше обсудим, как нам работать с новым стандартом.

Применение стандарта ECMAScript Harmony

В отличие от версии ECMAScript 5, которая стала стандартной *фактически* задолго того, как стать таковой *юридически*, версия Harmony является скорее направляющей, чем следуемой. Это означает, что текущая (на момент написания данной книги) реализация Harmony неоднородна. Поэтому для отслеживания различных состояний реализации Harmony нам потребуется ряд инструментальных средств. Во-первых, потребуются ресурсы, из которых мы могли бы узнать, в каких именно браузерах реализованы отдельные аспекты Harmony. Во-вторых, нужно выяснить, каким образом следует внедрять Harmony в этих браузерах. И в-третьих, мы должны рассмотреть программные средства, которые позволяют преобразовывать исходный код по стандарту ES6 в код, совместимый со стандартом ES5. После этого мы должны проанализировать те языковые средства данного стандарта, которые широко применяются или предполагают широкое употребление, чтобы войти в окончательный стандарт.

Ресурсы проекта Harmony

Технический комитет TC39 ведет страницы в Википедии по адресу `http://wiki.ecmascript.org/`, где можно отслеживать состояние предложенного стандарта ECMAScript Harmony. Особый интерес вызывают две страницы, где представлены требования, цели, средства и предложения. На странице требований приведена методология, по которой ведется разработка стандарта ES6. И хотя она не так важна для понимания языка, тем не менее разъясняет причины, по которым были приняты некоторые решения, исходя из целей и средств, определенных для проекта Harmony.

Например, в Harmony предлагается надлежащая область действия блока, но она реализуется посредством внедрения ключевого слова `let` вместо простого переопределения режима работы интерпретаторов JavaScript. Само это предложение сделано в соответствии с первыми двумя подпунктами первой цели: усовершенствование языка для написания сложных приложений и библиотек. Но реализация следует четвертой цели (сохранение как можно более простого контроля версий) и первым средствам (минимизации дополнительного семантического состояния, требующегося свыше стандарта ES5). Требования, цели и средства для проекта Harmony можно выяснить по следующему адресу:

```
http://wiki.ecmascript.org/doku.php?id=harmony:harmony
```

Другим важным ресурсом для изучения процесса работы над проектом Harmony является страница предложений. На этой странице отслеживаются различные предложения, внесенные в стандарт ES6, а также состояние их рассмотрения. Подача предложений была завершена в 2011 году, и поэтому на данной странице не видно никаких дополнений. На ней можно обнаружить в основном существующие предложения, а иногда и те предложения, которые были исключены из спецификации. Страницу предложений можно найти по следующему адресу:

http://wiki.ecmascript.org/doku.php?id=harmony:proposals

Спецификации замечательны как справочные документы, но иногда в них не хватает подробностей реализации. Нам бы хотелось иметь под рукой справочную информацию о состоянии, в котором находится реализация стандарта ES6 в браузерах и прочих механизмах JavaScript. К счастью, в нашем распоряжении имеются для этой цели две страницы. Первую страницу ведет известный разработчик приложений на JavaScript Юрий Зайцев, который называет себя kangax. Эта страница содержит таблицу совместимости с версией стандарта ECMAScript 6 и доступна по адресу `http://kangax.github.io/compat-table/es6/`. Таблица совместимости разделяет стандарт ES6 по языковым средствам и проверяет его внедрение в большинстве современных браузеров (для настольных и мобильных систем) и других реализациях JavaScript вроде Node.js. Эти проверки в какой-то мере упрощены и обычно сосредоточены на существовании предложения, а не на предлагаемых в нем функциональных возможностях и соответствии реализации данному предложению. Тем не менее данная страница служит отличной отправной точкой. Юрий Зайцев ведет также таблицы совместимости со стандартом ES5, грядущей спецификацией ECMAScript 7, а также с такими нестандартными языковыми средствами, как метод `__defineGetter__()` или свойство `caller` для функций.

На другой странице Томас Лан (Thomas Lahn) ведет таблицу ECMAScript, где отслеживается реализация стандартов ECMAScript в текущих версиях механизмов JavaScript. Эту страницу можно найти по адресу `http://pointedears.de/es-matrix/`. У Томаса Лана несколько иной подход, чем у Юрия Зайцева. В частности, Томаса Лана интересует текущее состояние механизмов JavaScript, и поэтому он отслеживает только механизмы Mozilla JavaScript, V8 (в браузере Chrome), Opera ECMAScript и некоторые другие, а Юрий Зайцев уделяет больше внимания реализации стандартов в браузерах и программном обеспечении. Кроме того, в таблице Томаса Лана отслеживаются все механизмы ECMAScript (вплоть до версии 6), что дает возможность оценить реализацию ключевого слова `let` наряду с массивами и циклами `for`. Его подход более основательный, но и таблица получается более крупной, что иногда замедляет ее загрузку. Тем не менее такая таблица совместимости, как и аналогичная таблица Юрия Зайцева, служит незаменимым ресурсом для профессиональных разработчиков приложений на JavaScript.

Работа со стандартом Harmony

Для работы с языковыми средствами по стандарту Harmony браузеры могут находиться в следующих состояниях.

- Браузеры, особенно возобновляемые Chrome и Firefox, могут иметь уже готовую реализацию языкового средства по стандарту Harmony, не требуя специальных усилий со стороны программиста. Но лишь немногие языковые средства были реализованы подобным образом на момент написания данной книги.
- Для применения языковых средств по стандарту Harmony в большинстве браузеров требуется добровольное согласие самого разработчика. Подробнее об этом речь пойдет несколько позже.
- Если языковое средство еще не реализовано в браузере (или же не реализовано надлежащим образом), возможно, придется воспользоваться транспилятором, позволяющим писать на уровне стандарта ES6, а затем приводить его в

соответствие со стандартом ES5, чтобы он мог выполняться в избранном механизме.

- С другой стороны, можно прибегнуть к полизаполнению для отдельных языковых средств из стандарта ECMAScript 6, которыми требуется воспользоваться.

Очевидно, что первое состояние браузеров не требует особых пояснений, поэтому перейдем сразу к обсуждению второго состояния. У обоих браузеров, Chrome и Firefox, имеются свои особенности в работе со стандартом ECMAScript 6.

В браузере Chrome придется перейти по следующему URL: `chrome://flags`. Это дает возможность активизировать экспериментальные языковые средства. В частности, придется активизировать состояние `chrome://flags/#enable-javascript-harmony`. Следует, однако, иметь в виду, что при некоторых обстоятельствах это может изменить режим работы браузера Chrome и привести к тому, что он будет не совсем верно воспроизводить некоторые страницы. А кроме того, изменение в состоянии `enable-javascript-harmony` остается постоянным. Если же вместо этого потребуются изменения только в начале конкретного сеанса, то браузер Chrome следует запустить на выполнение из командной строки с параметром **`--javascript-harmony`**. В некоторых случаях потребуется также выполнить сценарий JavaScript в строгом режиме, управлять которым можно на уровне прикладного кода.

При запуске браузера Firefox на выполнение изменять какие-либо настройки не требуется, но не исключено, что придется внести изменения в прикладной код. В общем, браузер Firefox потребует от вас пометить свой код как отличающийся от стандартного кода JavaScript. С этой целью введите атрибут `type` в дескрипторы `<script>`, установив в нем значение `"application/javascript;version=1.7"`. Этим будет активизировано большинство языковых средств по стандарту Harmony. Если же в прикладной код, написанный по стандарту Harmony, потребуется внести какие-нибудь дополнительные изменения, они должны быть отмечены конкретным языковым средством, которое они активизируют.

Любопытно, что браузер Internet Explorer требует минимальной (а по существу, никакой) настройки для выполнения кода по стандарту ECMAScript 6. С другой стороны, в версии Internet Explorer 10 реализованы лишь четыре языковых средства из спецификации по новому стандарту, тогда как в версии Internet Explorer 11 — в целом 12 таких средств, хотя этот браузер сильно отстает во внедрении нового стандарта от Firefox и Chrome. Можно сказать, что в версии Internet Explorer 11 мало что реализовано из нового стандарта, а то, что реализовано, сделано слишком просто.

Транспиляторы

Третью возможность для работы со стандартом ECMAScript 6 предоставляет *транспилятор*. В частности, транспилятор берет код, написанный по стандарту ECMAScript 6, и выполняет его кросс-компиляцию в код, совместимый со стандартом ECMAScript 5. Для транспиляции имеются самые разные инструментальные средства. Актуальный список транспиляторов и полизаполнений в хранилище GitHub ведет Эдди Османи (Addy Osmani). Этот список можно просмотреть по адресу `https://github.com/addyosmani/es6-tools`. Как следует из этого списка, имеется немало транспиляторов и полизаполнений. Мы продемонстрируем применение транспилятора Traceur для преобразования некоторого кода по стандарту ECMAScript 6 и его выполнения в браузере, поддерживающем стандарт ECMAScript 5.

Traceur относится к числу наиболее распространенных и часто обновляемых транспиляторов.

Чтобы воспользоваться транспилятором Traceur, его проще всего загрузить через платформу Node.js. Аналогично описанному в главе 9, посвященной инструментальным средствам веб-производства, воспользуемся платформой Node.js для загрузки дополнительного кода средствами NPM. В частности, для загрузки транспилятора Traceur введите следующую команду:

```
npm install traceur
```

По этой команде устанавливается текущая версия 0.0.72 транспилятора Traceur. Напомним, что в этой команде можно указать параметр **-g**, если требуется сделать Traceur глобально доступным. Но в любом случае вам, вероятно, придется обновить переменную окружения `PATH` в своей операционной системе, чтобы включить в нее путь к Traceur. Если вы работаете под Windows, то обнаружите в папке `node_modules` командный файл `.bin\traceur.cmd` для выполнения Traceur на платформе Node.js. Но Traceur можно запустить на выполнение и непосредственно, если ввести каталог `node_modules\.bin` в переменную окружения `PATH`. Чтобы проверить, находится ли Traceur по текущему пути, выполните команду **traceur --version**, по которой должен быть возвращен номер версии или сообщение об ошибке, если транспилятор Traceur не найден.

Транспилятор Traceur можно запустить на выполнение относительно существующего кода, написанного по стандарту ES6. С этой целью перейдите в режим командной строки и введите команду **traceur** вместе с файлом JavaScript, содержащим код по стандарту ES6. В итоге Traceur выполнит ваш код и выведет результат на консоль.

Рассмотрим пример нового синтаксиса классов в JavaScript. Стандарт ES6 позволяет создавать классы, хотя синтаксически получаемый код служит лишь оболочкой вокруг функционального стиля объявляемого типа данных. Этот синтаксис прост и понятен, поэтому воспользуемся им в примере кода, преобразуемого с помощью Traceur и демонстрируемого в листинге 11.1.

Листинг 11.1. Классы по стандарту ECMAScript 6, преобразуемые с помощью Traceur

```
class Car {

  constructor( make, model ) {
    this.make = make;
    this.model = model;
    this.speed = 0;
  }

  drive( newSpeed ) {
    console.log( 'DEBUG: Speed was previously %d', this.speed );
    this.speed = newSpeed;
    console.log( 'DEBUG: Speed is now %d', this.speed );
  }

  brake() {
    this.speed = 0;
    console.log( 'DEBUG: Setting speed to 0' );
```

```
  }

  getSpeed() {
    return this.speed;
  }

  toString() {
    return this.make + ' ' + this.model;
  }
}

var honda = new Car( 'Honda', 'Civic' );
console.log( 'honda.toString(): %s', honda.toString() );
honda.drive( 55 );
console.log( 'The Honda is going %d mph', honda.getSpeed() );
```

В приведенном выше примере кода создается класс типа `Car`, определяются три свойства (`make`, `model` и `speed`) и несколько методов, которые служат в качестве оболочек вокруг свойств (методы `brake()`, `drive()`, `getSpeed()`) или служебных целей (метод `toString()`). Как видите, в этом классе нет ничего особенного.

Но этот код не будет выполняться ни в одном из современных браузеров. Можете убедиться в этом сами. А если вы обратитесь к таблице совместимости, составленной Юрием Зайцевым, то обнаружите, что классы еще не реализованы ни в одном из основных браузеров — по крайней мере, так было на момент написания данной книги. Следовательно, этот код вполне подходит для экспериментирования с Traceur. Если вы сохраните его в файле (например, под именем `classes.js` в текущей папке), то можете выполнить его с помощью Traceur, введя следующую команду:

```
traceur classes.js
```

Результат выполнения этой команды должен выглядеть так, как показано ниже.

```
C:Projects\PJT\FutureOfJS
>traceur classes.js
honda.toString(): Honda Civic
DRBUG: Speed was previously 0
DRBUG: Speed is now 55
The Honda is going 55 mph
```

Как видите, транспилятор Traceur вполне справляется с рассматриваемым здесь кодом, написанным по новому стандарту. Подспудно транспилятор Traceur скомпилировал код по стандарту ES5, а затем выполнил его на самой платформе Node.js, т.е. он не сделал ничего особенного.

А как выполнить этот код в браузерах? Для этого имеются разные возможности. С одной стороны, можно воспользоваться транспилятором Traceur, чтобы сформировать выходные файлы, которые будут выполняться в браузерах, поддерживающих стандарт ES5. А с другой стороны, можно воспользоваться кодом, написанным по стандарту ES6, непосредственно в браузере, выполнив его транспиляцию с помощью Traceur в динамическом режиме. Чтобы вывести результат транспиляции в избранный файл, следует вызвать Traceur с параметром **`-out`**. Получаемый в итоге выходной файл не является автономным. Очевидно, что его можно выполнить вместе с Traceur, но его нельзя включить в состав HTML-страницы. Сначала придется

загрузить в динамическом режиме Traceur, а затем сценарий, который требуется выполнить. Пример HTML-оболочки для Traceur приведен в листинге 11.2.

Листинг 11.2. HTML-оболочка для Traceur

```
<!DOCTYPE html>
<html>
<head>
  <title>Traceur and classes</title>
</head>
<body>
<h2>Running Traceur output in the browser</h2>

<script src="../node_modules/traceur/bin/traceur-runtime.js">
</script>
<script src="classes-es5.js"></script>
</body>
</html>
```

Следует иметь в виду, что файл `traceur-runtime.js` загружается из папки `bin` транспилятора Traceur. В этом файле находится только тот код, который требуется для выполнения файлов, прошедших транспиляцию в Traceur. (А другой, намного больший файл в папке `bin` содержит код самого транспилятора Traceur.) Если загрузить HTML-файл из листинга 11.2, то на консоль будут выведены предполагавшиеся результаты. Но важнее другое: такой способ выполнения кода пригоден для текущих версий браузеров Firefox, Chrome и Internet Explorer.

Если требуется работать непосредственно с кодом, написанным по стандарту ECMAScript Harmony, то его можно всегда транспилировать с помощью Traceur в динамическом режиме. Например, чтобы воспользоваться исходным кодом из листинга 11.1, достаточно видоизменить HTML-оболочку из листинга 11.2 следующим образом:

```
<script src="https://google.github.io/traceur-compiler/bin/traceur.js">
</script>
<script src="https://google.github.io/traceur-compiler/src/bootstrap.js">
</script>
<script src="classes.js" type="module"></script>
```

В данном случае переход к хранилищу GitHub для транспилятора Traceur делается потому, что подобным образом проще всего получить доступ ко второму файлу под названием `bootstrap.js`. Этот файл не включается в установку транспилятора Traceur средствами NPM, как впрочем, и в его установку средствами Bower, и поэтому к нему приходится обращаться непосредственно. Начальная загрузка позволяет запустить Traceur на выполнение в контексте JavaScript. Кроме того, обращение к файлу `classes.js` происходит так, как будто он относится к типу `module`. Это условие, принятое в файле `bootstrap.js`, который загружает файл `classes.js` средствами Ajax явным образом как исходный код, написанный по стандарту ES6. Помимо этого, атрибут `type` оказывает побочный эффект, не загружая в браузер код, способный вызвать ошибку. В качестве альтернативы достаточно встроить код, написанный по стандарту ES6, непосредственно в блок дескрипторов `<script>`, хотя в атрибуте `type` все равно придется устанавливать значение `"module"`.

Встроенная транспиляция интересна для экспериментирования, но она все же не рекомендуется как практический прием для разработки и развертывания веб-приложений. Ведь она требует дополнительных затрат ресурсов всякий раз, когда загружается страница, что можно было бы сделать один раз, транспилировав исходный код в выходной файл. Не говоря уже о том, что загрузка файлов `traceur.js`, `bootstrap.js` и исходного кода происходит намного медленнее, чем загрузка только файла `traceur-runtime.js` и исходного кода.

Полизаполнения

И наконец, для реализации некоторых аспектов стандарта ECMAScript 6 можно загрузить на страницу полизаполнение. Прецедент использования такого решения несколько *уже*, чем область его применения. Вместо того чтобы реализовывать весь ряд языковых средств по стандарту ECMAScript 6, для чего может понадобиться нечто вроде транспилятора Traceur, достаточно выбрать только те средства, которые требуются, используя полизаполнения. Полизаполнения просто и логично предоставляют новые методы для прототипов объектов `String`, `Number` и `Array` или реализуют объекты `WeakMap` и `Set`. С другой стороны, полизаполнения по своему характеру не могут заменить такие языковые средства, как ключевые слова `let`, `const` или стрелочные функции. Таким образом, набор языковых средств, доступных по стандарту Harmony с помощью полизаполнений, ограничен.

Для полизаполнений имеется немало высококачественных реализаций. В каталоге инструментальных средств, составленном Эдди Османи по стандарту ES6, имеется раздел, посвященный полизаполнениям. Особого внимания заслуживает библиотека ES6-Shim, разработанная Полом Миллером (Paul Miller) и содержащая полизаполнения для большинства языковых средств по стандарту Harmony (`https://github.com/paulmillr/es6-shim/`). Когда мы будем рассматривать перечень языковых средств по стандарту ECMAScript 6 далее в этой главе, то упомянем те из них, которые предоставляются в библиотеке ES6-Shim.

Языковые средства по стандарту ECMAScript Harmony

В стандарте ECMAScript 6 внедрено немало новых языковых средств. Эти средства устраняют вопиющие недостатки в JavaScript (область действия блоков), упорядочивают синтаксис, делая акцент на функциональных возможностях (стрелочные функции), а также расширяют возможности JavaScript в обращении со сложными образцами кода (классами, модулями и обещаниями).

Рассмотрим сначала область действия блоков. Принятый в JavaScript необычный подход к области действия и поднятию переменных и определений функций до верхнего уровня их локальной области действия многие годы был камнем преткновения для начинающих программировать на JavaScript. Этот недостаток имел и культурный характер, поскольку те, у кого был опыт программирования на "настоящих" языках, с насмешкой относились к JavaScript из-за отсутствия в нем области действия блоков (классов или других привычных для них языковых средств). И хотя для подобной критики имелись все основания, это, тем не менее, мешало некоторым программистам оценить на практике достоинства JavaScript. Поэтому рассмотрим этот вопрос, прежде чем двигаться дальше.

Ключевое слово `let` позволяет ограничить область действия переменных произвольным блоком. Переменные с областью действия, ограниченной ключевым словом `let`, не подвержены поднятию. У ключевых слов `let` и `var` имеются два существенных отличия. Можно сказать, что переменные с областью действия, ограниченной ключевым словом `let`, действуют аналогично большинству локальных переменных, тогда как переменные с областью действия, ограниченной ключевым словом `var`, обладают рядом интересных возможностей. Если попытаться получить доступ к переменной с областью действия, ограниченной ключевым словом `let`, за пределами ее блока, то возникнет ошибка `ReferenceError`, как будто эта попытка сделана к недоступной переменной с областью действия, ограниченной ключевым словом `var`. И первое оказывается не более сложным, чем второе. Поэтому ключевое слово `let` рекомендуется употреблять вместо ключевого слова `var`, если требуется ограничить область действия блоков.

Ключевому слову `let` сопутствует также ключевое слово `const`, которое позволяет объявить переменную с постоянным значением. Переменные, объявленные с помощью ключевого слова `const`, должны быть инициализированы при их объявлении, иначе в этом нет никакого смысла, поскольку в дальнейшем изменить значение такой переменной нельзя. Такие переменные, как и те, что объявлены с помощью ключевого слова `let`, не подлежат поднятию, а область их действия ограничивается тем блоком, в котором они объявлены. Семантика попыток доступа к переменным `const` несколько отличается в разных браузерах. Если попытаться модифицировать переменную, объявленную с помощью ключевого слова `const`, браузер негласно завершит свою работу аварийным сбоем или выдаст ошибку соответствующего типа. Константы позволяют скомпилировать некоторые фрагменты в более быстродействующий код, поскольку механизму JavaScript известно, что они вообще не изменятся. А в сочетании с некоторыми коллекциями, рассматриваемыми далее в этой главе, константы можно употреблять для хранения закрытых данных класса. Подробнее об этом можно узнать по адресу `http://fitzgeraldnick.com/weblog/53/`.

Стрелочные функции

К еще одной категории усовершенствований в стандарте ECMAScript 6 относится внедрение синтаксических изменений, упрощающих некоторые объявления. Одним из таких изменений являются стрелочные функции, которые предоставляют более краткий синтаксис для определения функций, особенно встраиваемых. Рассмотрим в качестве примера следующий фрагмент кода:

```
var numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9];
numbers.forEach(function(num) {
  // сделать что-нибудь с числом, переданным функции
});
```

Этот код нельзя назвать неуклюжим, тем не менее, он несколько многословен. Попытка минимизировать длину литерального кода, не компилируя его, представляет в JavaScript особую трудность. Если определять функции с помощью ключевого слова `function`, то на быстродействии кода это никак не скажется. И чем больше функций применяется, тем больше придется употреблять ключевое слово `function`. Поэтому было бы неплохо иметь в своем распоряжении более краткий синтаксис функций.

Под влиянием языка CoffeeScript технический комитет TC39 внес предложение внедрить стрелочные функции, а точнее, функции, определяемые стрелками. Таким образом, код из предыдущего примера можно написать следующим образом:

```
var numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9];
numbers.forEach(num => {
  // сделать что-нибудь с числом, переданным функции
});
```

Что же в этом коде более изящного? Ниже приведены самые главные составляющие синтаксиса стрелочных функций.

```
аргументы => { код }
()        => { код } // Функция без аргументов
i         => { код } // Функция с одним аргументом
(i, j)    => { код } // Функция с несколькими аргументами
```

Для обозначения тела функции можно заключить несколько строк кода в фигурные скобки. А если это лишь одна строка кода, то ее можно оставить без подобного обрамления. Таким образом, следующая строка кода:

```
x => x * 2
```

равнозначна приведенному ниже определению функции.

```
function(x) {
  return x * 2;
}
```

Стрелочные функции имеют ряд отличий от обычных или стандартных функций. Прежде всего, стрелочные функции всегда привязаны к тому контексту, в котором они определены. Это может несколько усложнить дело. Рассмотрим следующий пример кода:

```
var courseAssignments = {
  teacher : 'Stephen Duffy',
  canTeach: function(courses) {
    courses.forEach( function ( course ) {
      console.log(
              'Ask %s if he can teach %s', this.teacher, course );
    });
  }
};

courseAssignments.canTeach(['Greek', 'Latin', 'Theology', 'History']);
```

В результате выполнения этого фрагмента кода появится следующая строка:

```
Ask undefined if he can teach Greek
(Спросить не определено, может ли он обучить греческому языку)
```

Очевидно, что это совсем не тот результат, который хотелось бы получить. Контекст и значение ссылки `this` в функции `forEach()` отсылают обратно к функции `canTeach()`, а не к объекту `courseAssignments`. Как правило, этот недостаток устраняется следующим образом:

```
var courseAssignments = {
  teacher : 'Stephen Duffy',
  canTeach: function(courses) {
    var that = this; // Сохранить контекст свойства canTeach
    courses.forEach( function ( course ) {
      console.log(
              'Ask %s if he can teach %s', that.teacher, course );
    });
  }
};

courseAssignments.canTeach(['Greek', 'Latin', 'Theology', 'History']);
```

В приведенном выше фрагменте кода полужирным выделено сохранение контекста ссылки `this` на уровне функции `canTeach()` вместо функции `forEach()`. Это положение можно значительно упростить с помощью стрелочных функций, которые автоматически привязывают к надлежащему контексту. Таким образом, приведенный выше фрагмент кода можно переписать следующим образом:

```
var courseAssignments = {
  teacher : 'Stephen Duffy',
  canTeach: function(courses) {
    courses.forEach( course => console.log(
            'Ask %s if he can teach %s', this.teacher, course ) );
  }
};

courseAssignments.canTeach(['Greek', 'Latin', 'Theology', 'History']);
```

Благодаря стрелочной функции ссылка `this.teacher` автоматически привязывается к контексту ссылки `this` в свойстве `canTeach`, что, собственно, и требуется. И это просто замечательно!

У стрелочных функций имеются и другие отличия. В частности, стрелочные функции нельзя употреблять в качестве конструкторов. Для этого им недостает внутреннего кода. Кроме того, стрелочные функции не поддерживают объект `arguments`. Дело в том, что в стандарте ECMAScript 6 определяются и допускаются значения параметров по умолчанию и остальных параметров, и поэтому потребность в объекте `arguments` в функциях вообще отпадает.

Классы

Одним из самых крупных синтаксических изменений в стандарте ECMAScript 6 является внедрение классов. Ключевое слово `class` было зарезервировано в Java Script с самого начала, но не имело с тех пор никакой реализации. С появлением объектно-ориентированных средств в языке JavaScript технический комитет TC39 признал, что в JavaScript потребовался более синтаксически ясный способ реализации классов и механизма наследования. В то же время у членов этого комитета не было желания внедрять еще один способ реализации типов, классов или чего-то вроде классов. Они преследовали цель устранить или хотя бы смягчить недоразумение, а не усугублять его.

Классы в спецификации ECMAScript 6: The ECMAScript Harmony устанавливаются с помощью ключевого слова `class`, внедренного в качестве синтаксического удобства для реализации функционально ориентированных типов данных. Это происходит следующим образом.

1. Сначала пишется код класса JavaScript на основании нового синтаксиса по стандарту ES6.
2. Затем механизм JavaScript компилирует этот класс в функцию, определяющую тип данных.
3. Далее обращение с этим классом происходит таким же образом, как и с типом данных.

Таким образом, семантика взаимодействия с классом (или типом данных) и его экземплярами не изменилась. А большая часть семантики определения класса (или типа данных) остается той же самой. Вернемся к предыдущему примеру, чтобы рассмотреть его более подробно. Напомним, что класс в JavaScript определяется следующим образом:

```
class [имя_класса] {
  constructor(аргумент 1, аргумент 2) { ... }
  [другие методы]
}
```

Класс определяется с помощью ключевого слова `class`. В его определение включается функция `constructor()`, которая будет вызываться в дальнейшем при обращении по ссылке `new [имя_класса]`. Остальные методы будут скопированы в прототип функции, определяемой в качестве конструктора. Получается довольно изящно!

В новой спецификации классы будут также иметь два важных средства, которые уже давно были желательны в объектно-ориентированной части JavaScript: простое наследование и метод доступа к суперклассу. Наследование доступно посредством ключевого слова `extends` следующим образом:

```
class Car extends Vehicle
```

В пределах подкласса можно воспользоваться ключевым словом `super` в качестве средства доступа к методам и свойствам из суперкласса. В отличие от некоторых других языков программирования (например, Java), в стандарте ECMAScript 6 не предусмотрен вызов `super` в качестве метода. Вместо этого в подклассе хранится ссылка на суперкласс аналогично ссылке на текущий экземпляр суперкласса.

К сожалению, классы вообще не реализованы ни в одном из основных современных браузеров. Спецификация класса станет частью конечной спецификации по стандарту ECMAScript 6, которую намечено выпустить в 2015 году. И производители браузеров, по-видимому, ожидают появления окончательной спецификации, чтобы убедиться, что их реализации не нанесут никакого вреда. Между тем транспилятор Traceur вполне справляется с классами, включая их наследование с помощью ключевых слов `extends` и `super`.

Обещания

Ранее в данной книге при рассмотрении технологии Ajax мы коснулись немного вопроса управления кодом в Ajax-ориентированных системах. Когда дело доходит

до создания функции, возвращающей результаты асинхронно, то написать зависимый код не так-то просто. Долгое время отсутствовало конкретное и простое решение этого вопроса. Большая часть программистов писали очень длинные функции обратного вызова, которые, в свою очередь, могли вызывать свои асинхронные функции, еще больше усложняя дело! А другие программисты пользовались именованными функциями, но придерживались стека вызовов. (Имеется немало именованных функций, но как управлять ими: в модуле или пространстве имен, не говоря уже о том, что такой способ вносит дополнительные сложности?) Асинхронное выполнение кода давно относилось к числу важных проблем, которые нужно было разрешить в JavaScript. Поэтому требовалось каким-то образом управлять асинхронными взаимодействиями, и для этого были внедрены обещания.

Обещание — это шаблон, помогающий управлять кодом, асинхронно возвращающим данные. Оно инкапсулирует этот код и вводит уровень управления, что очень похоже на обработку событий. Код можно зарегистрировать с помощью обещания, которое должно быть выполнено независимо от того, возвращаются ли данные успешно или не успешно. По завершении обещания выполняется соответствующий код. Аналогично обработке событий, зарегистрировать можно любое количество функций для выполнения в зависимости от удачного или неудачного исхода. А в любой момент можно зарегистрировать обработчик событий независимо от того, завершилось ли обещание или нет, что совсем *не* похоже на обработку событий.

Рассмотрим принцип действия обещаний для большей ясности на конкретном примере, демонстрируемом в листинге 11.3. В целом обещание может находиться в одном из следующих двух состояний: ожидания и установки. Обещание ожидает до тех пор, пока асинхронный вызов, который оно в себе заключает, не завершится возвратом данных, окончанием времени ожидания или иным образом. В этот момент обещание устанавливается. В состоянии установки общение разделяется на две разновидности: разрешенное или отклоненное. Разрешенное обещание устанавливается успешно, а отклоненное — безуспешно. А поскольку обещания произвольны (формально им даже не требуется асинхронный код), то и определение разрешения или отклонения может быть произвольным и отдается на откуп программисту.

Листинг 11.3. Обещания

```
var p = new Promise( function(resolve, reject) {
  // Сделать здесь что-нибудь, например, обработать Ajax-запрос
  setTimeout(function() {
    var result = 10 * 5;
    if (result === 50) {
      resolve(50);
    } else {
      reject( new Error( 'Bad math, mate' ) );
    }
  }, 1000);
});

p.then(function(result) {
  console.log( 'Resolved with a value of %d', result );
});
```

```
p.catch(function(err) {
  console.error( 'Something went horribly wrong.' );
});
```

Возможно, данный пример немного надуман, но он наглядно показывает порядок выполнения асинхронного кода. В основу данного примера положен вызов функции `setTimeout()`, а функция, планируемая для последующего выполнения, лишь решает математическое уравнение. Важная роль в данном примере отведена вызовам функций `resolve()` и `reject()`. В частности, функция `resolve()` сообщает потребителю обещания, что оно разрешено (т.е. успешно завершено). А функция `reject()` делает то же самое в отношении обещаний, устанавливаемых безуспешно. Но в любом случае вызываемой функции передается определенное значение. Именно это значение и получает потребитель обещания в функции, обрабатывающей разрешение или отклонение обещания.

В данном примере имеется также код, потребляющий обещание. Обратите, в частности, внимание на вызовы функций `then()` и `catch()`. Они соответствуют вызовам обработчиков событий, наступающих при удачном и неудачном исходе соответственно. Точнее говоря, вызвать функцию `then()` можно было бы следующим образом: `Promise.then(`*`onSuccess, onFailure`*`)`. Это дало бы возможность обратиться к функции `then()` непосредственно. Но ради соблюдения принятого стиля программирования функцию `catch()` лучше вызвать отдельно для обработки любых возникающих ошибок.

Вполне благоразумным было и отделение обработки обращения от его состояния. Это означает, что вызов функции `p.then()` или `p.catch()` можно сделать сколько угодно раз независимо от состояния обещания. Если на установку обещания потребуется 1000 миллисекунд, то код, регистрируемый при вызове функции `p.then()`, будет введен в стек кода для последующего вызова, когда установится обещание. А по истечении периода ожидания в течение 1000 миллисекунд, вполне вероятно, что код будет выполнен немедленно. Нетрудно представить наличие в реализации обещания следующего псевдокода:

```
function then(onSuccess, onFailure) {
  if (this.state === "pending") {
    addSuccessStack(onSuccess);
    addFailureStack(onFailure);
  } else if (this.state === "settled") {
    if (this.settledState === "success") {
      onSuccess();
    } else {
      onFailure();
    }
  }
}
```

Конкретная реализация требует несколько большей детализации, но рассмотрение данного вопроса выходит за рамки этой главы. Тем не менее рассмотренный здесь пример наглядно, хотя и приближенно, демонстрирует, что же происходит в самом обещании, когда выполняемый код регистрируется при установке обещания.

Обещания являются стандартным в версии ECMAScript 6 способом управления асинхронным кодом. На самом деле обещания уже применяются во многих библио-

теках JavaScript для управления Ajax-запросами, анимацией и другими, как правило, асинхронными взаимодействиями. Как будет показано в следующем разделе, обещания потребуются для реализации модулей по стандарту Harmony.

Между тем для работы с обещаниями имеются самые разные варианты. Транспилятор Traceur понимает обещания и применяет их внутренним образом для реализации модулей (и некоторых других средств). В таких библиотеках, как jQuery, AngularJS и прочих, имеются собственные реализации обещаний, хотя они немного отличаются от официальной спецификации. Это особенно касается библиотеки jQuery, поэтому не следует терять бдительность. Своя реализация обещаний имеется и в библиотеке ES6-Shim, тогда как в отличной библиотеке Q, разработанной Крисом Коуэлом (Kris Kowal), имеется особый тип `Q.Promise`, реализующий прикладной программный интерфейс ES6 API.

Обещания были лишь бегло рассмотрены в этом разделе. За рамками этого обсуждения остались прикладной программный интерфейс API и соответствующий тип данных. Впрочем, на данную тему в Интернете имеется немало превосходных статей, в том числе статья Джейка Арчибальда (Jake Archibald), доступная по адресу `http://www.html5rocks.com/en/tutorials/es6/promises/`.

Модули

До сих пор модули не рассматривались, но если говорить, например, о загрузчике модулей, то следует упомянуть и модулях AMD (Asynchronous Module Definition — асинхронное определение модулей), а на платформе Node.js — о модулях CommonJS. В стандарте ECMAScript 6 предпринимается попытка соединить эти два разных типа модулей в единый синтаксис. И в какой-то мере это сделано удачно, хотя и не идеально. Тем не менее технический комитет TC39 попытался найти компромиссное решение, наметив для программистов путь дальнейшего следования. Если придерживаться модулей типа AMD или предпочесть модули типа CommonJS, то в конечном счете кто-нибудь все равно напишет для них транспилятор — возможно, даже вы сами.

Итак, рассмотрим реализацию модулей по стандарту ES6. Мы не станем сравнивать модули по стандарту ES6 с модулями типа AMD или CommonJS, поскольку для этого потребовалась бы отдельная глава книги (превосходная публикация на эту тему имеется в блоге Алекса Раушмейера (Alex Rauschmeyer) по адресу `http://www.2ality.com/2014/09/es6-modules-final.html`). Вместо этого мы обсудим, что полезного могут нам принести модули.

В основу модулей положен следующий простой замысел: требуется надежное инкапсулированное пространство имен, в котором можно определить данные и функциональные возможности. И тогда по собственному усмотрению можно сделать доступными некоторые или все эти данные и функциональные возможности. Главное назначение модуля состоит в его повторном использовании. Это позволяет определить функциональные возможности один раз и пользоваться ими сколько и где угодно. В листинге 11.4 демонстрируется, каким образом эти требования реализуются в модулях.

Листинг 11.4. Определение модуля

```
export const schoolName = 'Mickey Kullen Memorial High School';

export const firstSport = 'Basketball';
```

```
export function getPrincipal() {
  // Возможно, обратиться к серверу в реальном коде
  return 'Jason Franzke';
}

export function fgPct(shots, baskets) {
  return baskets / shots;
}
```

Как видите, с помощью ключевого `export` слова экспортируется прикладной программный интерфейс API для модуля. Если эта операция выполняется по отдельным компонентам, можно экспортировать константы, обычные переменные и функции. При желании все, что требуется экспортировать, можно перечислить отдельным списком в конце исходного файла, как показано ниже, даже если в процессе изменяются имена экспортируемых компонентов!

```
export {schoolName, firstSport, getPrincipal, fgPct};
export {schoolName as name, firstSport as sport ... };
```

Имеется также возможность задать один компонент модуля как экспортируемый по умолчанию. Неудивительно, что такой компонент обозначается ключевым словом `default`, как показано ниже. Стандартные и именованные компоненты можно указывать вместе, хотя это лишь усложняет дело при последующем импорте таких компонентов.

```
export default function () { ... };
export default 'foo';
```

Какой же из этих вариантов выбрать? Тот, который работоспособен. Но, как показывает опыт Дэйва Хермана (Dave Herman), активного участника процесса разработки стандарта ES6, предпочтение следует отдавать экспорту одиночных модулей с помощью ключевого слова `default`. На эту тему велась острая полемика, в ходе которой многие высказывали совершенно противоположные мнения или, по крайней мере, альтернативные предпочтения. Но те, кто программирует на JavaScript, всегда выбирают именно тот способ, который больше всего удовлетворяет их потребностям.

Применяя модуль, можно взаимодействовать с ним двумя способами: декларативно и программно. Декларативный синтаксис обращения с модулями прост и понятен, как показано в листинге 11.5.

Листинг 11.5. Декларативный способ обращения с модулями

```
import * as school from 'test-module';

console.log('The principal of %s is %s.', school.schoolName,
            school.getPrincipal());
```

Ключевое слово `import` позволяет определить те части модуля, которые импортируются в указанное пространство имен. Имя модуля совпадает с именем файла, за исключением расширения **`.js`**. Допускается также указывать пути к файлу. Так, по пути `foo/bar` будет найден файл `bar.js`, находящийся в каталоге `foo` относительно местоположения текущего файла. Команда `import` довольно гибкая, поскольку она

позволяет импортировать из модуля столько компонентов, сколько требуется, как показано ниже.

```
// Импортировать только компонент 'schoolName'
import schoolName from 'test-module';

// Импортировать оба указанных именованных компонента
import {schoolName, getPrincipal} from 'test-module;

// Импортировать компоненты, экспортируемые по умолчанию
import someDefault from 'test-module';

// Импортировать стандартный и именованный компоненты.
// В TRACEUR НЕ ДЕЙСТВУЕТ
import someDef {schoolName} from 'test-module';

// Импортировать весь модуль, включая функцию, переменную
// и все, что доступно по умолчанию
import * as school from 'test-module';
console.log(s.default()); // Выполнить функцию,
                          // экспортируемую по умолчанию
```

Модули загружаются асинхронно. Браузер будет ожидать до тех пор, пока не будут загружены все модули перед выполнением любого кода. Если требуется более полный контроль над этим процессом (или же предпочтение отдается другому синтаксису), то можно выбрать программный способ импорта модулей (листинг 11.6).

Листинг 11.6. Программный способ импорта модулей

```
System.import( 'test-module' )
  .then( school => {
    console.log('The principal of %s is %s.', school.schoolName,
                school.getPrincipal());
  })
  .catch( error => {
    console.error( 'Something has gone horribly wrong.' );
  });
```

Синтаксис базового модуля не изменяется. Вместо этого применяется синтаксис, опирающийся на обещания, для загрузки модулей, если требуется связать конкретный код с загрузкой или выполнением конкретных модулей. (В данном случае ради интереса был употреблен синтаксис стрелочных функций!) Здесь, к сожалению, недостаточно места, чтобы подробно рассматривать возможности такого синтаксиса. Достаточно сказать, что обширный контроль над загрузкой и выполнением модулей всегда можно получить по мере надобности в нем.

Для модулей имеется ряд полизаполнений. Транспилятор вполне приемлемо обращается с модулями, за одним или двумя исключениями. Остальные инструментальные средства полизаполнений, включая пакет ES6 Module Loader Polyfill (`https://github.com/ModuleLoader/es6-module-loader`), перечислены в упомянутом ранее списке Эдди Османи на веб-странице ES6-Tools. Следует, однако, иметь в виду, что полизаполнение модулей в библиотеке ES6-Shim не предусмотрено.

Расширения типов данных

К последней категории изменений в спецификации ECMAScript относятся усовершенствования существующих типов данных. Некоторые из подобных изменений формализуют языковые средства, уже давно имеющиеся в JavaScript, вроде HTML-функций для строковых типов `String` и т.д. Одни из этих изменений не нуждаются в особых пояснениях, а другие требуют уточнения, что и будет сделано ниже.

Символьные строки

Рассматриваемые здесь строковые функции вызываются для экземпляра типа `String`. Иными словами, они доступны по ссылке `String.prototype`. Прежде всего, для обработки символьных строк предусмотрены следующие HTML-функции: `anchor()`, `big()`, `bold()`, `fixed()`, `fontcolor()`, `fontsize()`, `italics()`, `link()`, `small()`, `strike()`, `sub()` и `sup()`. Каждая из них принимает экземпляр типа `String` и возвращает копию, заключаемую в оболочку соответствующего дескриптора. Различные браузеры могут дополнять или исключать функции из приведенного выше списка. Так, в браузере Chrome поддерживается строковая функция `blink()`.

В приведенной ниже таблице перечислены некоторые служебные строковые функции и их назначение.

String.prototype.startsWith(*str*)	Определяет, начинается ли символьная строка с указанной подстроки ***str***
String.prototype.endsWith(*str*)	Определяет, оканчивается ли символьная строка указанной подстрокой ***str***
String.prototype.contains(*str*, [*startPos*])	Определяет, содержит ли символьная строка указанную подстроку ***str***
String.prototype.repeat(*count*)	Формирует новую символьную строку, которая является повторением строки типа **String** заданное количество раз ***count***

Числа

Тип `Number` дополнен несколькими статическими методами, большинство из которых служат для определения характеристик передаваемого аргумента. Эти методы перечислены в приведенной ниже таблице вместе с кратким описанием их назначения.

Number.isNaN(*num*)	Определяет, относится ли аргумент ***num*** к типу **Number**. Заменяет глобальную функцию **isNaN()**, применение которой вызывало некоторые трудности
Number.isFinite(*num*)	Определяет, является ли аргумент ***num*** конечным числом. Плюс и минус бесконечность (±µ) и не число (**NaN**) не относятся к конечным числам
Number.isInteger(*num*)	Определяет, является ли аргумент ***num*** целым числом. Не число (**NaN**) не относится ни к целым, ни к каким-либо нечисловым значениям
Number.isSafeInteger(*num*)	Определяет, может ли аргумент ***num*** быть надежно представлен числом с двойной точностью по стандарту IEEE-754, и не округляется ли до этого числа другое число с двойной точностью по стандарту IEEE-754

Number.parseInt(*string*, [*radix*])	Заменяет глобальную функцию **parseInt()**. Рекомендуется указывать основание системы счисления в качестве аргумента ***radix***, поскольку это позволяет нивелировать отличия в реализации
Number.parseFloat(*string*)	Заменяет глобальную функцию **parseFloat()**

Библиотека Math

Служебная библиотека Math дополнена рядом полезных функций. Большинство из них имеют более эзотерическое назначение, и поэтому здесь нет места для их подробного рассмотрения. Эти функции перечислены в приведенной ниже таблице вместе с кратким описанием их назначения.

Math.imul(*x*, *y*)	Возвращает результат умножения двух параметров с 32-разрядной точностью, как и в языке C
Math.clz32(*num*)	Возвращает количество начальных нулевых двоичных разрядов в 32-разрядном двоичном представлении числа
Math.fround(*num*)	Возвращает ближайшее представление числа с плавающей точкой и одинарной точностью
Math.log10(*num*)	Возвращает логарифм числа по основанию **10**
Math.log2(*num*)	Возвращает логарифм числа по основанию **2**
Math.log1p(*num*)	Возвращает натуральный логарифм (по основанию **e**) числа **+ 1**
Math.expm1(*x*)	Возвращает **ex - 1**, где ***x*** — аргумент, а **e** — основание натурального логарифма
Math.cosh(*num*)	Возвращает гиперболический косинус числа
Math.sinh(*num*)	Возвращает гиперболический синус числа
Math.tanh(*num*)	Возвращает гиперболический тангенс числа
Math.acosh(*num*)	Возвращает гиперболический арккосинус числа
Math.asinh(*num*)	Возвращает гиперболический арксинус числа
Math.atanh(*num*)	Возвращает гиперболический арктангенс числа
Math.hypot([*num*, *num2*, *num3*, ...])	Возвращает квадратный корень суммы квадратов своих аргументов
Math.trunc(*num*)	Возвращает целую часть числа, отбрасывая дробную часть числа, но не округляя его
Math.sign(*num*)	Возвращает знак числа, которое может быть положительным, отрицательными или нулевым
Math.cbrt(*num*)	Возвращает кубический корень числа

Массивы

Массивы претерпели в новом стандарте немало изменений. Рассмотрим сначала их функции и первой из них — простую служебную функцию `Array.from()`, статически доступную в типе `Array`. Эта функция принимает объекты, подобные массивам, или итерируемые объекты, преобразуя их в массивы, для которых доступны все остальные функции и свойства массивов. Параметр `arguments` в теле функции или значение, возвращаемое функцией `document.querySelectorAll()`, подобны массивам, хотя им и не достает многих свойств массивов. Теперь итерируемые объекты можно оперативно преобразовывать в массивы.

Массивы дополнены также тремя новыми свойствами для обхода их содержимого: `keys`, `values` и `entries`. Нетрудно догадаться, что свойство `keys` предоставляет

индексы массива, values — индексы значений, а entries — элементы массива, каждый из которых состоит из ключа и значения. Но обход массива осуществляется с помощью итератора. Это означает, что вместо получения всех значений просматриваются отдельные элементы массива по мере их обхода. Это может быть удобно для обработки динамических массивов, улучшения управления памятью, поиска в массивах и т.д.

Для поиска в массивах уже имеется функция Array.prototype.indexOf(). Но если требуется проверка не только на простое равенство, то можно воспользоваться функциями Array.prototype.find() и Array.prototype.findIndex(). Обе эти функции принимают в качестве аргументов предикатную функцию, а также дополнительный контекст, в котором она должна выполняться. А эта предикатная функция, подобно другим предикатным функциям вроде forEach(), map() и прочих, принимает в качестве аргументов элемент, индекс и обратную ссылку на первоначальный код. Характерный пример применения функций обработки массивов приведен в листинге 11.7.

Листинг 11.7. Применение функции Array.prototype.find()

```
var names = ['John', 'Jon', 'Josй', 'Joseph', 'Mike',
             'Andre', 'Melanie', 'Jaymi', 'Kathy', 'Jennifer'];

names.find( function ( element, index ) {
  if ( element.startsWith( 'J' ) ) {
    console.log('The name %s at position %d starts with "J"!',
                element, index);
  }
} );
```

Имеется также функция Array.prototype.fill(), позволяющая заполнять массив значениями. В качестве дополнительных аргументов она принимает начальную и конечную позиции элементов в массиве. Еще одним важным усовершенствованием массивов является внедрение оператора spread. Программирующим на JavaScript уже давно требовалась возможность развернуть массив, передав его функции в качестве аргумента. Вплоть до стандарта ECMAScript 5 для этого не было ничего предусмотрено, хотя можно было воспользоваться функцией Function.prototype.apply(), которая принимала массив, развертывая его в виде аргументов вызываемой функции. Но это было очень неудобно и во всяком случае не совсем ясно. А теперь для той же самой цели можно воспользоваться оператором spread следующим образом:

```
 [1, 2, 3].push(...[4, 5, 6])
[1, 2, 3, 4, 5, 6]
```

Изящно, не так ли? Благодаря этому функции push(), pop(), shift() и unshift() становятся во многом более эффективными, тогда как функции splice() и concat() — намного более специализированными.

Полизаполнения

Эти расширения стандартных типов данных не поддерживаются в транспиляторе Traceur. В общем, акцент в транспиляторе Traceur делается на новых изменениях в синтаксисе, а не на расширениях уже имеющихся типов данных.

С другой стороны, все эти средства поддерживаются в библиотеке ES6-Shim. Полизаполнения большинства этих средств можно найти в упоминавшемся ранее списке Эдди Османи на веб-странице ES6-Tools, где они разделены по функциям или типам.

Новые типы коллекций

В языке JavaScript явно не хватало реализации коллекций. До появления стандарта ECMAScript Harmony для создания платформенно-ориентированных структур данных имелись только типы `Array` и `Object`, что было далеко не идеальным выбором. В стандарте ECMAScript 6 для дополнительной поддержки новых структур данных были внедрены типы `Set`, `WeakSet`, `Map` и `WeakMap`, предназначенные в основном служить в качестве прикладного программного интерфейса Collections API для JavaScript. Дальнейшую судьбу коллекций в JavaScript следует связывать с типами `Set`, `WeakSet`, `Map` и `WeakMap`, а типом `Object` пользоваться для работы только с объектами, но не для выполнения двойной функции отображений и ассоциативных массивов.

Рассмотрим сначала неслабые типы коллекций. В частности, тип `Map` определяет множество ключей и значений. Ключами и значениями могут быть любые значения примитивных типов или объекты. Эти типы коллекций явно предназначены для замены структур данных типа `Object`, но у них имеются следующие важные отличия.

- Ключи для коллекций типа `Object` могут быть символьными строками, тогда как ключи для коллекций типа `Map` — любого типа данных.
- Для определения размера у коллекций типа `Map` имеется специальное свойство, а у коллекций типа `Object` такое свойство отсутствует. Иными словами, размер коллекций типа `Object` придется отслеживать вручную, тогда как в коллекциях типа `Map` это делается автоматически.
- У коллекций типа `Object` имеются свои прототипы. Строго говоря, прототипы имеются и у коллекций типа `Map`, но прототипы экземпляров типа `Map` получаются иначе, чем прототипы экземпляров типа `Object`. Очень важно подчеркнуть, что у коллекций типа `Object` имеются стандартные ключи, тогда как у коллекций типа `Map` они отсутствуют.

Коллекции типа `Set` являются массивами, гарантирующими их однозначность. Множество элементов может быть однозначным в соответствии с проверкой на равенство в операции `===`. Данные можно извлекать из коллекции типа `Set` в том порядке, в каком они были в нее введены. Коллекцию типа `Set` можно создать из массива, передав объект типа `Array` в качестве аргумента конструктору объектов типа `Set`. И хотя преобразовать коллекцию типа `Set` в массив нелегко кросс-браузерными средствами, предусмотренными в стандарте ECMAScript, всегда можно перебрать элементы коллекции типа `Set` в цикле `for...of` или с помощью функции `Set.prototype.forEach()` и ввести отдельные элементы в массив по мере надобности.

Новые коллекции поддерживаются в последних версиях браузеров Firefox и Chrome. А в браузере Internet Explorer предусмотрена лишь элементарная поддержка коллекций типа `Map`, `WeakMap` и `Set`, но не `WeakSet`! Это означает, что браузеру Internet Explorer известен тип коллекции и имеется некоторая ее реализация, но не разрешается, например, создавать экземпляр коллекции типа `Map` с помощью конструктора `new Map(iterable)`. Тем не менее имеется возможность создавать пустые коллекции типа `Map`, `Set`, `WeakMap` и вводить в них элементы, обращаясь к соответ-

ствующим функциям из прикладного программного интерфейса API. Впрочем, предполагается полная поддержка коллекций в последующих версиях Internet Explorer по сведениям о состоянии развития этого браузера, доступным по адресу `https://status.modern.ie/`.

Слабые типы коллекций

В чем же особенности слабых коллекций типа `WeakSet` и `WeakMap`? Чтобы уяснить их, нужно хотя бы немного разбираться в механизме сборки "мусора", действующем в JavaScript. В общем, объект доступен для сборки "мусора", когда подсчет ссылок на него достигнет нуля. Это означает, что наличные ссылки на рассматриваемый объект отсутствуют, включая переменные, ключи, значения, элементы и пр. Элементы коллекций типа `Map` и `Set` считаются ссылками на объекты.

Допустим, что ссылка на сложный объект создана и сохранена сначала в переменной, а затем в коллекции типа `Set`. Прежде чем произойдет возврат из функции или фрагмента кода, переменная высвобождается из оперативной памяти, а ее значение становится равным пустому. Казалось бы, очистка памяти происходит должным образом, но не следует забывать, что это делается небыстро. Элемент хранится в коллекции типа `Set`, а объект недоступен для сборки "мусора" до тех пор, пока он не будет удален из данной коллекции (или вся коллекция освобождена из оперативной памяти).

Именно здесь и приходят на помощь коллекции типа `WeakSet` и `WeakMap`. Так, в коллекции типа `WeakSet` хранятся слабые ссылки на объекты. Это означает, что экземпляр, хранившийся в коллекции типа `Set`, не учитывается при общем подсчете ссылок, поскольку ссылка на него является слабой. Если из оперативной памяти высвобождены все остальные ссылки на объект, он становится доступным для сборки "мусора". Это может оказаться очень удобным для управления памятью, хотя несколько усложняет дело, если полагаться на сохранение слабой ссылки.

Аналогичным образом действует и коллекция типа `WeakMap`, где хранятся слабые ключи. К тому же эти ключи могут быть *только* объектами типа `Object`, но не примитивными типами данных. В отличие от обычных коллекций типа `Map`, ключи в коллекциях типа `WeakMap` нельзя перебрать, поскольку их состояние не определено вследствие потенциальной сборки "мусора". Поэтому список ключей придется вести вручную — возможно, в виде массива, если потребуется доступ к нему.

Прикладной программный интерфейс Collections API

Разные коллекции реализуются с помощью общего прикладного программного интерфейса Collections API. Функции этого интерфейса поддерживаются не во всех коллекциях, но большинство из них являются общими для двух и более классов коллекций. Эти функции и свойства перечислены в приведенной ниже таблице вместе с их кратким описанием.

Функция/Свойства	Set	WeakSet	Map	WeakMap	Описание
size	Да	Да	Да	Нет	Количество элементов в коллекции
add(e)	Да	Да	Нет	Нет	Вводит элемент во множество
clear()	Да	Да	Да	Да	Удаляет все элементы из коллекции

Функция/Свойства	Set	WeakSet	Map	WeakMap	Описание
delete(*k*)	Да	Да	Да	Да	Удаляет элемент из отображения по заданному ключу или из множества по указанному значению
has(*k*)	Да	Да	Да	Да	Возвращает логическое значение **true** или **false** в зависимости от наличия заданного ключа в отображении или значения во множестве
get(*k*)	Нет	Нет	Да	Да	Получает значение, связанное с заданным ключом
set(*k*, *v*)	Нет	Нет	Да	Да	Устанавливает заданный ключ ***k*** по указанному значению ***v***
entries()	Да	Нет	Да	Нет	Возвращает итератор, который возвратит отдельные массивы для пар "ключ–значение" в отображении или двумерные массивы для каждого значения во множестве
forEach(*fn*, [*scope*])	Да	Нет	Да	Нет	Перебирает элементы коллекции, возвращая заданную функцию ***fn*** для каждого элемента (значения из множества или массив пар "ключ–значение" из отображения)
keys()	Да	Нет	Да	Нет	Возвращает ключи для коллекции. Во множествах ключи и значения означают одно и то же
values()	Да	Нет	Да	Нет	Возвращает значения для коллекции

Полизаполнения

Аналогично расширениям типов данных в JavaScript, в Traceur не реализованы транспиляции новых типов коллекций по стандарту ES5, поскольку это сфера действия полизаполнений. И, как упоминалось ранее, у полизаполнения из библиотеки ES6-Shim имеются свои реализации типов `Map`, `Set`, `WeakMap` и `WeakSet` по стандарту ES5. Имеется также полизаполнение коллекций по стандарту ES6 (`https://github.com/Benvie/harmony-collections`), реализующее только типы `Map`, `Set`, `WeakMap` и `WeakSet`.

Следует, однако, иметь в виду, что если полизаполнение и может подражать прикладному программному интерфейсу API, оно неспособно дублировать очень важную особенность коллекций типа `WeakMap` и `WeakSet`: хранение слабых ссылок на объекты, чтобы не вести общий подсчет ссылок для целей сборки "мусора". Это можно реализовать только путем изменений в самом механизме JavaScript, чтобы эмулировать, например, поведение коллекции типа `WeakMap`, не связываясь со сборщиком "мусора", и благодаря этому будут освобождаться ссылки, хранящиеся в данной коллекции.

Резюме

В этой главе предпринята попытка сделать краткий обзор новых языковых средств, которые предусмотрены в грядущем стандарте ECMAScript 6. С этой целью была рассмотрена большая часть спецификации JavaScript по новому стандарту.

В 2015 году она должна приобрести окончательную форму, и на эту тему еще появится немало литературы.

Между тем мы обсудили в этой главе языковые средства (например, стрелочные функции), устраняющие некоторые из самых неприятных недостатков JavaScript. Мы также рассмотрели ряд средств, которыми многие программирующие на JavaScript пользуются уже теперь, в том числе обещания, классы и модули. И мы с большим оптимизмом смотрим в будущее JavaScript. К моменту выхода данной книги в свет стандарт ECMAScript 6 будет определен окончательно, а за ним последует стандарт ECMAScript 7, к работе над которым уже приступает технический комитет TC39!

ПРИЛОЖЕНИЕ А

Справочник по модели DOM

В этом приложении приведен справочный материал по функциональным возможностям, которые предоставляет объектная модель документа (Document Object Model — DOM), обсуждавшаяся в главе 5.

Ресурсы

Функциональные возможности модели DOM появились в самых разных вариантах, начиная с предварительной спецификации модели DOM нулевого уровня и вплоть до уровня 3. В отношении модели DOM следует уяснить, что она считается фактически действующим стандартом. На каждом уровне этой модели описываются внедряемые средства и виды поведения. Сама модель DOM представляет документ в виде узлов и свойств, с которыми связаны определенные события.

Если требуется разобраться в подробностях реализации модели DOM, превосходным источником для ее изучения служат следующие веб-сайты консорциума W3C и рабочей группы WHATWG:

- HTML DOM Level 3: `http://www.w3.org/TR/DOM-Level-3-Core/`.
- WHATWG DOM: `https://dom.spec.whatwg.org/`.

Кроме того, для изучения функциональных возможностей модели DOM имеется немало другого превосходного справочного материала. Но среди всего этого материала нет ничего лучше того, что можно найти на веб-сайте, организованном Питером-Полом Кохом (Peter-Paul Koch) по адресу `http://www.quirksmode.org/`, где основательно рассмотрен каждый метод DOM в сравнении с результатами его применения во всех современных (и некоторых других) браузерах. Этот неоценимый ресурс позволяет выяснить, что можно и чего нельзя сделать в браузерах, для которых разрабатываются веб-приложения. Еще один полезный ресурс по модели DOM был создан Алексисом Девера (Alexis Devera) по адресу `http://caniuse.com/`, где можно найти нужное средство и просмотреть таблицу совместимости, чтобы выяснить, в каких именно браузерах поддерживается данное средство.

Терминология

В главе 5 и в этом приложении употребляется терминология, общепринятая в XML и DOM для описания различных свойств представления XML-документа в модели DOM. Перечисленные ниже термины связаны с моделью DOM в частности и XML-документами вообще. Употребление всех этих терминов связано с примером HTML-документа, приведенным в листинге А.1.

Листинг А.1. Образец HTML-документа для обсуждения терминологии DOM и XML

```
<!doctype html>
<html>
<head>
  <title>Introduction to the DOM</title>
</head>
<body>
  <h1>Introduction to the DOM</h1>
  <p class="test">There are a number of reasons why the DOM is awesome,
     here are some:</p>
  <ul>
    <li id="everywhere">It can be found everywhere.</li>
    <li class="test">It's easy to use.</li>
    <li class="test">It can help you to find what you want,
        really quickly.</li>
  </ul>
</body>
</html>
```

Предок

Термин, взятый из генеалогии для обозначения родителя текущего элемента и всех его предшественников. В примере HTML-документа из листинга А.1 *предками* элемента разметки `<ul>` являются элементы разметки `<body>` и `<html>`.

Атрибут

Атрибутами являются свойства элементов, содержащие дополнительные сведения о них. В примере HTML-документа из листинга А.1 у элемента разметки `<p>` имеется атрибут `class`, содержащий значение `test`.

Потомок

У любого элемента может быть любое количество узлов, каждый из которых считается *потомком* родительского элемента. В примере HTML-документа из листинга А.1 элемент разметки `<ul>` состоит из семи порожденных узлов, тремя из которых являются элементы разметки `<li>`, а четырьмя другими — знаки окончания строки, расположенные между этими элементами и находящиеся в текстовых узлах.

Документ

XML-документ состоит из одного элемента разметки, называемого *корневым узлом* или *элементом документа*. В примере HTML-документа из листинга А.1 эле-

мент разметки <html> является элементом документа, содержащим остальную часть документа.

Узел-потомок

К *узлам-потомкам* элемента разметки относятся порожденные им узлы и все порожденные ими узлы. В примере HTML-документа из листинга А.1 к узлам-потомкам элемента разметки <body> относятся элементы разметки <h1>, <p>, <ul>, все элементы разметки <li> и все содержащиеся в них текстовые узлы.

Элемент

Элемент представляет собой контейнер, содержащий атрибуты и остальные узлы. Элементы являются самыми главными и наиболее примечательными составляющими любого HTML-документа. В примере HTML-документа из листинга А.1 имеется немало элементов, причем все дескрипторы <html>, <head>, <title>, <body>, <h1>, <p>, <ul> и <li> обозначают элементы этого документа.

Узел

Узел является общим блоком в DOM-представлении документа. Элементы, атрибуты, комментарии, документы и текстовые узлы — все они являются узлами, и поэтому у них имеются типичные свойства узлов. Например, свойства nodeType, nodeName и nodeValue имеются у каждого узла.

Родитель

Термином *родитель* обозначается элемент, содержащий текущий узел. У каждого узла, кроме корневого, имеется свой родительский узел. В примере HTML-документа из листинга А.1 элемент разметки <body> является родителем элемента разметки <p>.

Родственник

Родственный узел является потомком того же самого родительского узла. Обычно этим термином обозначается контекст двух атрибутов previousSibling и nextSibling, имеющихся у всех узлов в модели DOM. В примере HTML-документа из листинга А.1 родственными для элемента разметки <p> являются элементы разметки <ul>, а также пара текстовых узлов, заполненных пробелами.

Текстовый узел

Текстовым является специальный узел, содержащий только текст, в том числе видимый текст и все формы пробелов. Так, если в документе имеется элемент разметки (например, hello world!), то в нем на самом деле содержится текст "hello world!" в отдельном текстовом узле. В примере HTML-документа из листинга А.1 текст "It's easy to use" содержится в текстовом узле второго элемента разметки <li>.

Глобальные переменные

Глобальные переменные существуют в глобальной области действия прикладного кода. Но своим существованием они помогают выполнять общие для DOM операции.

Переменная document

Эта переменная содержит активный HTML-документ, построенный по модели DOM и просматриваемый в окне браузера. Но если переменная document существует и имеет значение, то это совсем не означает, что ее содержимое полностью загружено и проанализировано синтаксически. Подробнее об ожидании загрузки HTML-документов, построенных по модели DOM, см. в главе 5. В листинге А.2 демонстрируются некоторые примеры применения переменной document, хранящей представление HTML-документа, построенного по модели DOM, для доступа к его элементам.

Листинг А.2. Применение переменной document для доступа к элементам документа

```
// Найти элемент с идентификатором 'body'
document.getElementById("body")

// Найти все элементы с дескриптором <div>
document.getElementsByTagName("div")
```

Переменная HTMLElement

Эта переменная содержит объект суперкласса для всех HTML-документов, построенных по модели DOM. Расширяя прототип этого объекта, можно расширить все элементы такого документа. Этот суперкласс доступен по умолчанию в браузерах, совместимых с Mozilla, а также в браузере Opera. Его можно также ввести в браузеры Internet Explorer и Safari. В листинге А.3 демонстрируется пример привязки новых функций к глобальной переменной HTMLElement. Так, привязка функции hasClass() предоставляет возможность выяснить, имеет ли элемент конкретный класс.

Листинг А.3. Привязка новых функций к глобальной переменной HTMLElement

```
// Ввести новый метод во все элементы HTML-документа,
// построенного по модели DOM, чтобы выяснить с его помощью,
// имеет ли элемент конкретный класс
HTMLElement.prototype.hasClass = function( class ) {
  return new RegExp("(^|\\s)" + class +
         "(\\s|$)").test( this.className );
};
```

Перемещение по модели DOM

Перечисленные ниже свойства и функции являются частью всех элементов модели DOM. Они могут быть использованы для обхода документов, построенных по модели DOM.

Свойство body

Это свойство HTML-документа, построенного по модели DOM и хранящегося в глобальной переменной document. Оно указывает непосредственно на элемент <body> разметки HTML-документа, где он должен быть единственным. Это конкрет-

ное свойство существует еще со времен нулевого уровня модели DOM Level 0. В листинге А.4 демонстрируются некоторые примеры доступа к элементу разметки <body> из HTML-документа, построенного по модели DOM.

Листинг А.4. Доступ к элементу разметки <body> из HTML-документа, построенного по модели DOM

```
// Изменить поля документа в элементе разметки <body>
document.body.style.margin = "0px";

// Ссылка document.body равнозначна следующему вызову:
document.getElementsByTagName("body")[0]
```

Свойство childNodes

Это свойство всех документов, построенных по модели DOM. Оно содержит массив всех порожденных узлов, включая элементы, текстовые узлы, комментарии и пр. Это свойство доступно только для чтения. В листинге А.5 демонстрируется, каким образом следует пользоваться свойством childNodes для ввода стиля оформления во все элементы, порожденные родительским элементом.

Листинг А.5. Добавление красного обрамления вокруг элементов, порожденных элементом разметки <body>, с помощью свойства childNodes

```
// Добавить обрамление вокруг всех элементов,
// порожденных элементом разметки <body>
var c = document.body.childNodes;
for ( var i = 0; i < c.length; i++ ) {
  // Убедиться в том, что узел является элементом
  if ( c[i].nodeType == 1 )
    c[i].style.border = "1px solid red";
}
```

Свойство documentElement

Это свойство всех узлов DOM. Оно действует в качестве ссылки на корневой элемент документа. Если это HTML-документ, то свойство documentElement всегда указывает на элемент разметки <html>. В листинге А.6 приведен пример применения свойства documentElement для поиска элемента в модели DOM.

Листинг А.6. Пример обнаружения корневого элемента из любого узла DOM

```
// Найти элемент по идентификатору, используя свойство documentElement
someRandomNode.documentElement.getElementById("body")
```

Свойство firstChild

Это свойство всех элементов DOM, которое указывает на первый порожденный узел данного элемента. Если же у элемента отсутствуют порожденные узлы, свойство firstChild принимает пустое значение. В листинге А.7 приведен пример применения свойства firstChild для удаления всех порожденных узлов из указанного элемента.

Листинг А.7. Удаление всех порожденных узлов из указанного элемента

```
// Удалить все порожденные узлы из указанного элемента
var e = document.getElementById("body");
while ( e.firstChild )
  e.removeChild( e.firstChild );
```

Функция getElementById(*elemID*)

Эта функция позволяет эффективно обнаруживать в документе один элемент по указанному идентификатору. Она доступна только для элемента документа. Кроме того, данная функция может действовать не так, как предполагалось, в документах, построенных по модели DOM, но не относящихся к типу HTML. Как правило, в определении типа документа (DTD) или схеме XML-документов, построенных по модели DOM, приходится явным образом указывать атрибут идентификатора. Эта функция принимает в качестве единственного аргумента наименование искомого идентификатора, как показано в листинге А.8.

Листинг А.8. Два примера обнаружения элементов HTML-разметки по их атрибутам

```
// Найти элемент по идентификатору "body"
document.getElementById("body")

// Скрыть элемент по идентификатору "notice"
document.getElementById("notice").style.display = 'none';
```

Функция getElementsByTagName(*tagName*)

Эта функция обнаруживает все порожденные элементы, начиная с текущего элемента, по указанному имени дескриптора. Она действует одинаково в XML- и HTML-документах, построенных по модели DOM. В современных браузерах можно указать метасимвол * в имени дескриптора и найти все порожденные элементы, что намного ускоряет процесс по сравнению с рекурсивной функцией, написанной только на JavaScript.

Эта функция принимает в качестве единственного аргумента имя дескриптора искомых элементов. В листинге А.9 демонстрируются примеры применения функции `getElementsByTagName()`. В первом блоке кода данного примера во все элементы разметки `<div>` вводится класс выделения текста в документе. Во втором блоке кода сначала обнаруживаются все элементы, присутствующие в элементе разметки с идентификатором *body*, а затем скрываются любые элементы, имеющие класс выделения текста.

Листинг А.9. Два блока кода, демонстрирующие применение функции `getElementsByTagName()`

```
// Найти все элементы разметки <div> в текущем HTML-документе
// и установить их класс равным 'hilite'
var d = document.getElementsByTagName("div");
for ( var i = 0; i < d.length; i++ ) {
  d[i].className = 'hilite';
}
```

```
// Обойти все элементы, порожденные элементом,
// имеющим идентификатор "body".
// Затем найти все элементы с классом, равным 'hilite'.
// И далее скрыть все совпавшие элементы
var all = document.getElementById("body").getElementsByTagName("*");
for ( var i = 0; i < all.length; i++ ) {
  if ( all[i].className == 'hilite' )
    all[i].style.display = 'none';
}
```

Свойство lastChild

Это свойство содержит ссылку, доступную для всех элементов DOM. Оно указывает на последний узел-потомок данного элемента. Если узлы-потомки отсутствуют, свойство `lastChild` будет содержать пустое значение. В листинге А.10 демонстрируется пример применения свойства `lastChild` для ввода элемента в документ.

Листинг А.10. Создание нового элемента разметки `<div>` и его ввод перед последним элементом в дескрипторе `<body>`

```
// Ввести новый элемент перед последним элементом в дескрипторе <body>
var n = document.createElement("div");
n.innerHTML = "Thanks for visiting!";

document.body.insertBefore( n, document.body.lastChild );
```

Свойство nextSibling

Это свойство содержит ссылку, доступную для всех элементов DOM. Оно указывает на следующий родственный узел. Если это последний родственный узел, свойство `nextSibling` будет содержать пустое значение. В листинге А.11 приведен пример применения свойства `nextSibling` для создания интерактивного списка определений.

Листинг А.11. Расширение всех элементов разметки `<dt>` до родственных элементов разметки `<dd>` после щелчка на них

```
// Найти все элементы разметки <dt> (Определение Термин)
var dt = document.getElementsByTagName("dt");
for ( var i = 0; i < dt.length; i++ ) {
  // Следить за моментом, когда на термине будет произведен щелчок
  dt[i].onclick = function() {
    // У каждого термина имеется смежный элемент разметки
    // <dd> (Определение), поэтому его можно отобразить после щелчка.
    // ПРИМЕЧАНИЕ: действует только в том случае, если между элементами
    // разметки <dd> отсутствуют пробелы
    this.nextSibling.style.display = 'block';
  };
}
```

Свойство parentNode

Это свойство всех узлов DOM. В каждом узле DOM оно указывает на элемент, который его содержит, за исключением элемента документа с пустой ссылкой, поскольку это корневой элемент, который ничего не содержит. В листинге А.12 приведен пример применения свойства parentNode для организации специального взаимодействия. Если щелкнуть на кнопке Cancel (Отмена), то родительский узел скроется.

Листинг А.12. Применение свойства parentNode для организации специального взаимодействия

```
// Следить за моментом, когда на ссылке Cancel будет произведен
// щелчок, а затем скрыть родительский узел
document.getElementById("cancel").onclick = function(){
  this.parentNode.style.display = 'none';
};
```

Свойство previousSibling

Это свойство содержит ссылку, доступную для всех узлов DOM. Оно указывает на предыдущий родственный узел. Если это первый родственный узел, свойство previousSibling будет содержать пустое значение. Не следует, однако, забывать, что свойство previousSibling может указывать на элемент DOM, комментарий или даже текстовый узел. Оно не служит в качестве исключительного средства для перемещения по элементам DOM. В листинге А.13 приведен пример применения свойства previousSibling для сокрытия элементов.

Листинг А.13. Сокрытие всех элементов, находящихся перед текущим элементом

```
// Найти все узлы, находящиеся перед данным узлом, и скрыть их
var cur = this.previousSibling;
while ( cur != null ) {
  cur.style.display = 'none';
  cur = this.previousSibling;
}
```

Сведения об узлах

Перечисленные ниже свойства существуют в большинстве элементов DOM для того, чтобы упростить доступ к общим сведениям об элементах.

Свойство innerText

Это свойство всех элементов DOM. Оно поддерживается только в браузерах, несовместимых с Mozilla, поскольку не является частью стандарта, предложенного консорциумом W3C. Это свойство возвращает символьную строку, содержащую весь текст в текущем элементе. А поскольку свойство innerText не поддерживается в браузерах, совместимых с Mozilla, то можно прибегнуть к следующему обходному приему: воспользоваться функцией для сбора значений из порожденных текстовых

узлов. В листинге А.14 приведен пример применения свойства `innerText` и функции `text()`, упоминавшейся в главе 5.

Листинг А.14. Применение свойства `innerText` для извлечения текстовой информации из элемента

```
// Допустим, что имеется следующий элемент разметки <li>,
// хранящийся в переменной 'li':
// <li>Please visit <a href="http://mysite.com/">my web site</a>.</li>

// Воспользоваться свойством innerText
li.innerText

// или функцией text(), описанной в главе 5
text( li )

// В любом случае получится следующий результат:
"Please visit my web site."
```

Свойство nodeName

Это свойство доступно для всех элементов DOM. Оно содержит имя элемента, набранное прописными буквами. Так, если имеется элемент разметки `<li>`, то при обращении к свойству `nodeName` возвращается имя `LI` данного элемента. В листинге А.15 приведен пример применения свойства `nodeName` для видоизменения имен классов родительских элементов.

Листинг А.15. Обнаружение всех элементов разметки `<li>` и установка в них класса `current`

```
// Найти всех родителей данного узла, которые являются
// элементами разметки <li>
var cur = this.parentNode;
while ( cur != null ) {
  // Как только элемент найден и проверено его имя,
  // ввести в него класс current
  if ( cur.nodeName == 'LI' )
    cur.className += " current";
  cur = this.parentNode;
}
```

Свойство nodeType

Это общее свойство для всех узлов DOM. Оно содержит номер, соответствующий типу его узла. Ниже перечислены три наиболее распространенных типа узлов, применяемых в HTML-документах.

- Узел элемента (имеет номер **1** или имя **document.ELEMENT_NODE**).
- Текстовый узел (имеет номер **3** или имя **document.TEXT_NODE**).
- Узел документа (имеет номер **9** или имя **document.DOCUMENT_NODE**).

Обращение к свойству nodeType — надежный способ проверить, что узел, к которому требуется получить доступ, обладает всеми предполагаемыми свойствами. Например, свойство nodeType полезно только для элемента DOM, и поэтому можно проверить, равно ли его значение 1, прежде чем получать доступ к данному элементу. В листинге А.16 демонстрируется пример применения свойства nodeType для ввода класса в ряд элементов разметки.

Листинг А.16. Обнаружение первого элемента в дескрипторе <body> разметки HTML-документа и установка класса header в этом элементе

```
// Найти первый элемент в дескрипторе <body>
var cur = document.body.firstChild;
while ( cur != null ) {
  // Если элемент найден, ввести в него класс header
  if ( cur.nodeType == 1 ) {
    cur.className += " header";
    cur = null;

  // Иначе продолжить перемещение по узлам-потомкам
  } else {
    cur = cur.nextSibling;
  }
}
```

Свойство nodeValue

Это полезное свойство текстовых узлов может использоваться для доступа и манипулирования текстом, который в них содержится. Лучшим тому примером служит функция text(), представленная в главе 5 и предназначенная для извлечения всего текстового содержимого из элемента. В листинге А.17 приведен пример применения свойства nodeValue для построения простой функции, извлекающей текст из элементов разметки документа.

Листинг А.17. Функция, принимающая элемент и возвращающая текстовое содержимое этого элемента и всех порожденных им элементов

```
function text(e) {
  var t = " ";
  // Если элемент передан данной функции, получить
  // его узлы-потомки, а иначе допустить, что это массив
  e = e.childNodes || e;

  // Просмотреть все узлы-потомки
  for ( var j = 0; j < e.length; j++ ) {
    // Если же это не элемент, присоединить его текстовое содержимое,
    // а иначе обойти все узлы-потомки данного элемента
    t += e[j].nodeType != 1 ?
      e[j].nodeValue : text(e[j].childNodes);
  }
```

```
  // Возвратить совпавший текст
  return t;
}
```

Атрибуты

Большинство атрибутов доступны в виде свойств элемента, который их содержит. Например, атрибут идентификатора доступен по простой ссылке `element.id`. Атрибуты существуют еще со времен нулевого уровня модели DOM, но их дальнейшее развитие маловероятно, поскольку они и так довольно просты и распространены.

Свойство `className`

Это свойство существует во всех элементах DOM, позволяя вводить и удалять классы из этих элементов. Оно упоминается здесь потому, что его имя `className` совершенно отличается от предполагаемого имени *класса*. Такое нелогичное наименование объясняется тем, что слово ***class*** зарезервировано в большинстве языков ООП. Во избежание трудностей программирования браузеров рекомендуется избегать употребления этого слова. В листинге А.18 приведен пример применения свойства `className` для сокрытия ряда элементов.

Листинг А.18. Поиск и сокрытие всех элементов разметки `<div>`, имеющих класс `special`

```
// Найти все элементы разметки <div> в документе
var div = document.getElementsByTagName("div");
for ( var i = 0; i < div.length; i++ ) {
  // Найти все элементы разметки <div>, имеющие
  // единственный класс 'special'.
  if ( div[i].className == "special" ) {
    // И скрыть их
    div[i].style.display = 'none';
  }
}
```

Функция `getAttribute(attrName)`

Эта функция служит для доступа надлежащим образом к значению атрибута, содержащегося в элементе DOM. Атрибуты инициализируются значениями, предоставленными пользователем в простом HTML-документе.

Функция `getAttribute()` принимает в качестве единственного аргумента имя атрибута, значение которого требуется извлечь. В листинге А.19 приведен пример применения функции `getAttribute()` для поиска элементов ввода, имеющих конкретный тип.

Листинг А.19. Поиск элемента разметки `<input>` с именованным текстом и копирование его значения в элемент с идентификатором предварительного просмотра

```
// Найти все элементы ввода в форме
var input = document.getElementsByTagName("input");
```

```
for ( var i = 0; i < input.length; i++ ) {

  // Найти элемент с именем "text"
  if ( input[i].getAttribute("name") == "text" ) {

    // Скопировать значение в другой элемент
    document.getElementById("preview").innerHTML =
    input[i].getAttribute("value");
  }
}
```

Функция removeAttribute(*attrName*)

Эта функция служит для удаления атрибута из элемента. Теоретически результат ее применения можно сравнить с вызовом рассматриваемой далее функции `setAttribute()`, принимающей пустую строку или пустое значение. Но на практике следует непременно удалить все лишние атрибуты во избежание любых непредвиденных последствий.

Функция `removeAttribute()` принимает в качестве единственного аргумента имя удаляемого атрибута. В листинге А.20 демонстрируется пример сброса некоторых флажков, установленных в форме.

Листинг А.20. Поиск и сброс всех флажков, установленных в документе

```
// Найти все элементы ввода в форме
var input = document.getElementsByTagName("input");
for ( var i = 0; i < input.length; i++ ) {
  // Найти все флажки
  if ( input[i].getAttribute("type") == "checkbox" ) {
    // Сбросить флажок
    input[i].removeAttribute("checked");
  }
}
```

Функция setAttribute(*attrName, attrValue*)

Эта функция служит для установки значения атрибута, содержащегося в элементе DOM. С его помощью можно также вводить специальные атрибуты, которые могут быть снова доступны в дальнейшем, не затрагивая внешний вид элементов DOM. Функция `setAttribute()` проявляет не совсем обычное поведение в браузере Internet Explorer, избавляя от необходимости устанавливать конкретные атрибуты (например, `class` или `maxlength`). Подробнее об этом см. в главе 5.

Функция `setAttribute()` принимает два аргумента: имя атрибута и устанавливаемое в нем значение. В листинге А.21 приведен пример установки значения атрибута в элементе DOM.

Листинг А.21. Применение функции `setAttribute()` для создания ссылки `<a>` на поисковый механизм Google

```
// Создать новый элемент разметки <a>
var a = document.createElement("a")

// Установить URL для посещения веб-сайта Google
a.setAttribute("href","http://google.com/");

// Ввести внутренний текст, предоставить пользователю возможность
// произвести щелчок на созданной ссылке
a.appendChild( document.createTextNode( "Visit Google!" ) );

// Ввести ссылку в конце документа
document.body.appendChild( a );
```

Модификация модели DOM

Ниже перечислены все свойства и функции, доступные для манипулирования моделью DOM.

Функция `appendChild(`*`nodeToAppend`*`)`

Эта функция служит для ввода узла-потомка в содержащий элемент. Если присоединяемый узел уже существует в документе, он перемещается из его текущего места и присоединяется к текущему элементу. Функцию `appendChild()` следует вызывать для того элемента, который требуется присоединить.

Эта функция принимает в качестве единственного аргумента ссылку на узел DOM. Это может быть только что созданный узел или ссылка на узел, существующий в другом месте документа. В листинге А.22 демонстрируется пример создания нового элемента разметки `<ul>` и перемещения в нем всех элементов разметки `<ul>` из их исходного положения в модели DOM, а затем присоединения к новому элементу разметки `<ul>` в теле документа.

Листинг А.22. Присоединение ряда элементов разметки `<ul>` к одному элементу разметки `<ul>`

```
// Создать новый элемент разметки <ul>
var ul = document.createElement("ul");

// Найти все первые элементы разметки <li>
var li = document.getElementsByTagName("li");
for ( var i = 0; i < li.length; i++ ) {

  // Присоединить каждый совпавший элемент разметки <li>
  // к новому элементу разметки <ul>
  ul.appendChild( li[i] );
}

// Присоединить новый элемент разметки <ul> в конце тела документа
document.body.appendChild( ul );
```

Функция cloneNode(*true*|*false*)

Эта функция позволяет разработчикам упростить прикладной код, дублируя существующие узлы, которые могут быть введены в модель DOM. В результате обычного вызова функции `insertBefore()` или `appendChild()` узел DOM может быть перемещен в документе, но вместо этого лучше воспользоваться функцией `cloneNode()`.

Функция `cloneNode()` принимает в качестве единственного аргумента логическое значение `true` или `false`. Если этот аргумент принимает логическое значение `true`, то клонируется сам узел и все, что в нем находится. А если он принимает логическое значение `false`, то клонируется только узел. В листинге А.23 демонстрируется пример применения данной функции для клонирования элемента и присоединения его копии к оригиналу.

Листинг А.23. Поиск первого элемента разметки `<ul>` в документе, создание полной его копии и ее присоединение к оригиналу

```
// Найти первый элемент разметки <ul>
var ul = document.getElementsByTagName("ul")[0];

// Клонировать узел и присоединить его копию к оригиналу
ul.parentNode.appendChild( ul.cloneNode( true ) );
```

Функция createElement(*tagName*)

Это основная функция для создания новых элементов в структуре DOM. Функция `createElement()` существует в качестве свойства документа, в котором требуется создать элемент.

■ ***На заметку*** Если вы пользуетесь обычной HTML-разметкой с типом содержимого **`text/html`** вместо XHTML-разметки с типом содержимого **`application/xhtml+xml`**, выберите функцию **`createElementNS()`** вместо функции **`createElement()`**.

Функция `createElement()` принимает в качестве единственного аргумента имя дескриптора создаваемого элемента. В листинге А.24 приведен пример применения данной функции для создания элемента и заключения внутри него других элементов в документе.

Листинг А.24. Заключение содержимого элемента разметки `<p>` в элементе разметки `<strong>`

```
// Создать новый элемент разметки <strong>
var s = document.createElement("strong");

// Найти первый абзац в документе
var p = document.getElementsByTagName("p")[0];

// Заключить содержимое элемента разметки <p> в
// оболочку элемента разметки <strong>
while ( p.firstChild ) {
  s.appendChild( p.firstChild );
}
```

```
// Переместить элемент разметки <strong> с прежним содержимым
// элемента разметки <p> обратно в элемент разметки <p>
p.appendChild( s );
```

Функция createElementNS(*namespace*, *tagName*)

Эта функция очень похожа на функцию `createElement()` в том отношении, что она создает новый элемент. Но в то же время эта функция дает возможность указать пространство имен для создаваемого элемента, например, в том случае, если в XML- или XHTML-документ вводится новый элемент.

Функция `createElementNS()` принимает следующие аргументы: пространство имен вводимого элемента, а также имя дескриптора этого элемента. В листинге А.25 приведен пример применения данной функции для создания элемента DOM в достоверном XHTML-документе.

Листинг А.25. Создание нового элемента разметки `<p>` в XHTML-документе, его заполнение текстом и присоединение к телу документа

```
// Создать новый элемент разметки <p>,
// совместимый со спецификацией XHTML
var p = document.createElementNS("http://www.w3.org/1999/xhtml", "p");

// Ввести текст в элемент разметки <p>
p.appendChild( document.createTextNode( "Welcome to my site." ) );

// Ввести элемент разметки <p> в документ
document.body.insertBefore( p, document.body.firstChild );
```

Функция createTextNode(*textString*)

Эта функция служит для создания новой текстовой строки для ввода в документ, построенный по модели DOM. Текстовые узлы служат лишь DOM-оболочками для текста, и поэтому не следует забывать, что их нельзя присоединять или выполнять их стилевое оформление. Функция `createTextNode()` существует только в качестве свойства документа, построенного по модели DOM.

Функция `createTextNode()` принимает в качестве единственного аргумента символьную строку, которая должна стать содержимым текстового узла. В листинге А.26 приведен пример применения данной функции для создания нового текстового узла и его присоединения к телу HTML-страницы.

Листинг А.26. Создание элемента разметки `<h1>` и присоединение нового текстового узла

```
// Создать новый элемент разметки <h1>
var h = document.createElement("h1");

// Создать текст заголовка и ввести его в элемент разметки <h1>
h.appendChild( document.createTextNode("Main Page") );

// Ввести заголовок в начале дескриптора <body>
document.body.insertBefore( h, document.body.firstChild );
```

Свойство innerHTML

Это свойство характерно для HTML-документов, созданных по модели DOM. Оно служит для доступа и манипулирования строковым представлением HTML-содержимого элемента в модели DOM. Если вы работаете только с HTML-документом, а не с XML-документом, то данное свойство может вам очень пригодиться, поскольку оно позволяет значительно сократить объем кода, требующегося для формирования нового элемента DOM, не говоря уже о том, что это делается намного быстрее, чем с помощью традиционных методов DOM. И хотя данное свойство не является частью какого-то конкретного стандарта, предложенного консорциумом W3C, оно поддерживается во всех современных браузерах. В листинге А.27 приведен пример применения свойства innerHTML для изменения содержимого элемента всякий раз, когда изменяется содержимое элемента разметки <textarea>.

Листинг А.27. Отслеживание изменений в элементе разметки <textarea> и обновление его значением текущего состояния предварительного просмотра

```
// Получить текстовую область для отслеживания обновлений
var t = document.getElementsByTagName("textarea")[0];

// Извлечь текущее значение из элемента разметки <textarea> и
// обновлять текущее состояние предварительного просмотра всякий раз,
// когда это значение изменяется
t.onkeypress = function() {
  document.getElementById("preview").innerHTML = this.value;
};
```

Функция insertBefore(*nodeToInsert*, *nodeToInsertBefore*)

Эта функция служит для ввода узла DOM в любом месте документа. Эту функцию следует вызывать для родительского элемента того узла, который требуется ввести. В качестве аргумента nodeToInsertBefore можно указать пустое значение, а в качестве аргумента nodeToInsert — последний порожденный узел.

Функция insertBefore() принимает два аргумента: узел, который требуется ввести в модель DOM, и узел DOM, перед которым он вводится. Это должна быть ссылка на достоверный узел. В листинге А.28 приведен пример применения данной функции для ввода пиктограммы веб-сайта, которая появляется рядом с URL, вводимым в строке адреса в окне браузера, а также рядом со ссылками на странице.

Листинг А.28. Обход всех элементов разметки <a> и ввод пиктограммы веб-сайта

```
// Найти все элементы разметки <a> со ссылками в документе
var a = document.getElementsByTagName("a");
for ( var i = 0; i < a.length; i++ ) {

  // Создать изображение пиктограммы сайта, на который делается ссылка
  var img = document.createElement("img");
  img.src = a[i].href.split('/').splice(0,3).join('/') + '/favicon.ico';

  // Ввести изображение пиктограммы сайта перед ссылкой
```

```
  a[i].parentNode.insertBefore( img, a[i] );
}
```

Функция removeChild(*nodeToRemove*)

Эта функция служит для удаления узла из документа, созданного по модели DOM. Функцию `removeChild()` следует вызывать для родительского элемента удаляемого узла. Эта функция принимает в качестве единственного аргумента ссылку на узел DOM, удаляемый из документа. В листинге А.29 приведен пример обхода всех элементов разметки `<div>` в документе и удаления любого из них, где имеется единственный класс оформления *warning*.

Листинг А.29. Удаление всех элементов, имеющих конкретно указанный класс оформления

```
// Найти все элементы разметки <div>
var div = document.getElementsByTagName("div");
for ( var i = 0; i < div.length; i++ ) {
  // Если у элемента разметки <div> имеется класс оформления 'warning'
  if ( div[i].className == "warning" ) {
    // Удалить этот элемент разметки <div> из документа
    div[i].parentNode.removeChild( div[i] );
  }
}
```

Функция replaceChild(*nodeToInsert*, *nodeToReplace*)

Эта функция служит в качестве альтернативы процессу удаления узла и вводу на его место другого узла. Функцию `replaceChild()` следует вызывать для родительского элемента заменяемого узла.

Данная функция принимает два аргумента: узел, который требуется ввести в модель DOM, и узел, который требуется заменить. В листинге А.30 приведен пример замены всех элементов разметки `<a>` на элементы разметки `<strong>`, содержащие URL с первоначальными ссылками на них.

Листинг А.30. Преобразование ряда ссылок в простые URL

```
// Преобразовать все ссылки в видимые URL ради удобства их вывода
// Найти все элементы разметки <a> в документе
var a = document.getElementsByTagName("a");
while ( a.length ) {

  // Создать элемент разметки <strong>
  var s = document.createElement("strong");

  // Сделать содержимое равным <a> Ссылка URL
  s.appendChild( document.createTextNode( a[i].href ) );

  // Заменить исходный элемент разметки <a> новым элементом разметки <strong>
  a[i].replaceChild( s, a[i] );
}
```

Предметный указатель

М

Н

О

П

Р

С

Т

Ф

Ш

Я

ЭФФЕКТИВНЫЙ И СОВРЕМЕННЫЙ C++

42 рекомендации по использованию C++11 и C++14

Скотт Мейерс

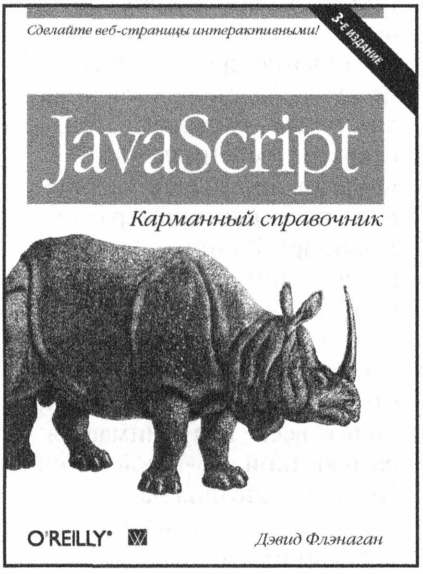
www.williamspublishing.com

В этой книге отражен бесценный опыт ее автора как программиста на C++. Глобальные изменения в языке программирования C++, приведшие к появлению стандартов C++11/14, приводят к необходимости изучения C++ если не заново, то по крайней мере как очень сильно изменившегося языка программирования.
Пройти путь изучения и освоения современного C++ вам поможет книга Скотта Мейерса, показывающая наиболее интересные места языка и предупреждающая о возможных проблемах и ловушках.
Хотя эта книга в первую очередь предназначена для энтузиастов и профессионалов, она достойна места на полке любого программиста — как профессионала, так и зеленого новичка.

ISBN 978-5-8459-2000-3 **в продаже**

JAVASCRIPT
КАРМАННЫЙ СПРАВОЧНИК
3-Е ИЗДАНИЕ

Дэвид Флэнаган



www.williamspublishing.com

JavaScript — популярнейший язык программирования, который уже более 15 лет применяется для написания сценариев интерактивных веб-страниц. В книге представлены наиболее важные сведения о синтаксисе языка и показаны примеры его практического применения. Несмотря на малый объем карманного издания, в нем содержится все, что необходимо знать для разработки профессиональных веб-приложений.

Главы 1–9 посвящены описанию синтаксиса последней версии языка (спецификация ECMAScript 5).

- Типы данных, значения и переменные
- Инструкции, операторы и выражения
- Объекты и массивы
- Классы и функции
- Регулярные выражения

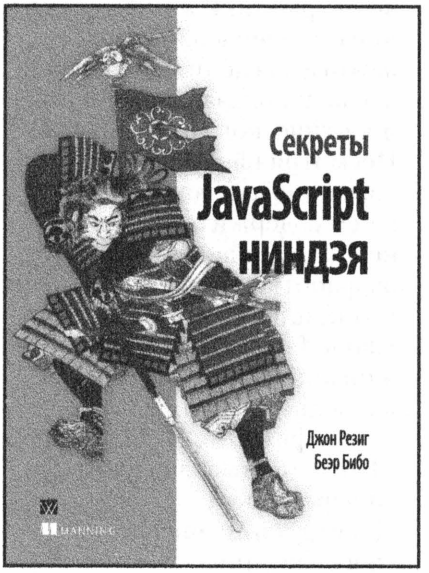
В главах 10–14 рассматриваются функциональные возможности языка наряду с моделью DOM и средствами поддержки HTML5.

ISBN 978-5-8459-1830-7

в продаже

СЕКРЕТЫ JAVASCRIPT НИНДЗЯ

ДЖОН РЕЗИГ
БЕЭР БИБО



www.williamspublishing.com

Эта книга раскрывает секреты мастерства разработки веб-приложений на JavaScript. Начиная с пояснения таких основных понятий, как функции, объекты, замыкания, прототипы, регулярные выражения и таймеры, авторы постепенно проводят читателя по пути обучения от ученика до мастера, раскрывая немало секретов и специальных приемов программирования на конкретных примерах кода JavaScript. В книге уделяется немало внимания вопросам написания кросс-браузерного кода и преодолению связанных с этим типичных затруднений, что может принести немалую пользу всем, кто занимается разработкой веб-приложений. Книга рассчитана на подготовленных читателей, стремящихся повысить свой уровень мастерства в программировании на JavaScript в частности и разработке веб-приложений вообще.

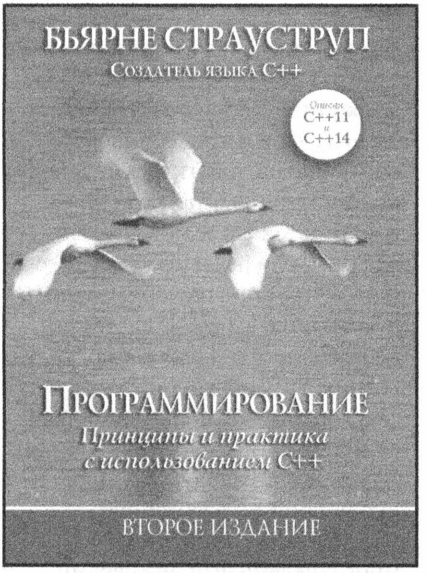
ISBN 978-5-8459-1959-5 **в продаже**

ПРОГРАММИРОВАНИЕ.

ПРИНЦИПЫ И ПРАКТИКА С ИСПОЛЬЗОВАНИЕМ C++

ВТОРОЕ ИЗДАНИЕ

Бьярне Страуструп



www.williamspublishing.com

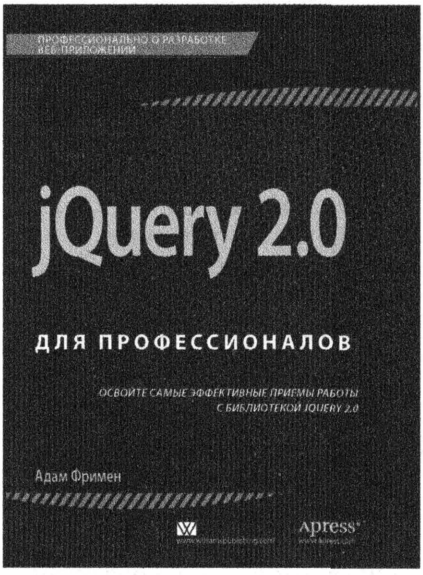
Эта книга — учебник по программированию. Несмотря на то что его автор — создатель языка C++, книга не посвящена этому языку; он играет в большей степени иллюстративную роль. Книга задумана как вводный курс по программированию с примерами программных решений на языке C++ и описывает широкий круг понятий и приемов программирования, необходимых для того, чтобы стать профессиональным программистом.

В первую очередь книга адресована начинающим программистам, но она будет полезна и профессионалам, которые найдут в ней много новой информации, а главное, смогут узнать точку зрения создателя языка C++ на современные методы программирования.

ISBN 978-5-8459-1949-6 **в продаже**

JQUERY 2.0 ДЛЯ ПРОФЕССИОНАЛОВ

Адам Фримен



www.williamspublishing.com

Автор книги, Адам Фримен, делится с читателями секретами наиболее эффективных приемов работы с jQuery, фокусируя основное внимание на практических аспектах использования этой технологии и демонстрируя ее применение для решения реальных задач.
В этом поистине исчерпывающем руководстве вы найдете ответы на все вопросы, которые могут возникать у вас в процессе разработки веб-приложений на основе jQuery.
Благодаря подробному и тщательно продуманному изложению материала, дополненному многочисленными примерами готового работающего кода, демонстрирующими мощь и гибкость jQuery, эта книга поможет быстро приобрести знания и навыки, необходимые профессионалам в области веб-разработки.

ISBN 978-5-8459-1919-9 в продаже

ПРОФЕССИОНАЛАМ ОТ ПРОФЕССИОНАЛОВ

JavaScript для профессионалов

2-е издание

Эта книга станет незаменимым пособием для профессиональных разработчиков современных веб-приложений на JavaScript. В ней представлено все, что требуется знать о современном состоянии JavaScript, а также поясняется, как пользоваться JavaScript при создании веб-сайтов. В этой книге не тратится впустую место на обсуждение того, что должно быть уже известно читателю, а вместо этого уделяется внимание основополагающим и актуальным вопросам программирования на JavaScript и таящимся в нем скрытым препятствиям.

В этой книге вам предстоит ознакомиться с ключевым словом `this` и новыми типами объектов. В ней поясняется, как создавать повторно используемый код посредством инкапсуляции, перегрузки и наследования. В книге обстоятельно рассматриваются современные приемы отладки и тестирования кода, а также инструментальные средства разработки вроде Jasmine, PhantomJS и Protractor. Настоящее издание книги завершается главами, посвященными построению одностраничных веб-приложений, господствующих в современной веб-разработке.

Книга изобилует многочисленными практическими и подробно разбираемыми примерами кода, повторно используемых функций и классов, экономящих время, отводимое на разработку. Она позволяет разработчикам овладеть практическими навыками написания динамических веб-приложений на высоком профессиональном уровне, а также помогает им повысить свою квалификацию.

ОСНОВНЫЕ ТЕМЫ КНИГИ:

- Современное состояние JavaScript
- Рост популярности библиотек
- Поддержка мобильных устройств
- Отладка кода JavaScript
- Объектная модель документов
- Обработка событий
- AngularJS и тестирование
- Инструментальные средства веб-разработки

НА ВЕБ-САЙТЕ

Исходные коды всех примеров, рассмотренных в книге, можно загрузить с веб-сайта издательства по адресу: http://www.williamspublishing.com/Books/978-5-8459-2054-6.html.

Джон Резиг работает разработчиком в Академии Хана и является создателем библиотеки jQuery для JavaScript. Помимо данной книги, он является автором книги *Secrets of the JavaScript Ninja* (издательство Manning, 2012 г.; в русском переводе эта книга вышла под названием *Секреты JavaScript ниндзя* в ИД "Вильямс", 2013 г.).

Расс Фергюсон работает разработчиком и инструктором в районе Нью-Йорка. В настоящее время он руководит компанией SunGard Consulting Services, занимающейся разработкой приложений для таких клиентов, как Morgan Stanley и Comcast.

Джон Пакстон является программистом, инструктором, автором книг и презентатором, проживающим в своем родном штате Нью-Джерси. За последние пятнадцать лет ему пришлось программировать на самых разных языках, применяемых в веб-разработке. В настоящее время Джон остановил свой выбор на языках JavaScript и Java, хотя иногда он испытывает ностальгические порывы к Perl и XML.

Категория: программирование

П **ст** разработка веб-приложений на JavaScript

окой квалификации

ISBN 978-5-8459-2054-6
16013
9 785845 920546

www.williamspubl

Apress®
www.apress.com