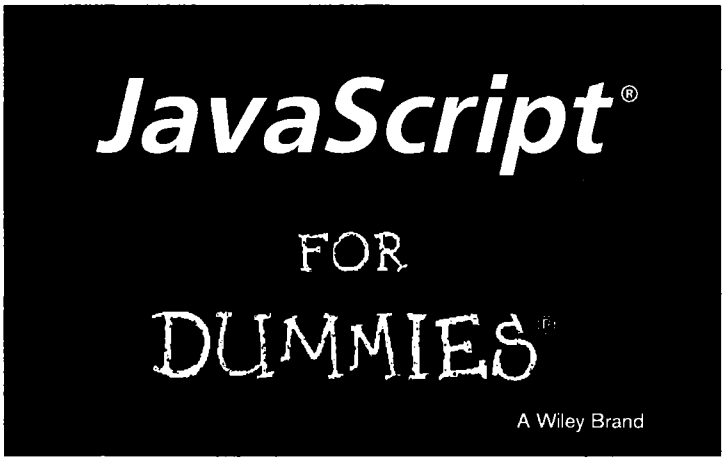

JavaScript®
ДЛЯ
ЧАЙНИКОВ®



by Chris Minnick and Eva Holland

FOR DUMMIES®
A Wiley Brand

JavaScript® для ЧАЙНИКОВ®

Крис Минник, Ева Холланд

ДИАЛЕКТИКА

Москва • Санкт-Петербург • Киев
2017

ББК 32.973.26-018.2.75
М62
УДК 681.3.07

Компьютерное издательство "Диалектика"
Главный редактор *С.Н. Тригуб*
Зав. редакцией *В.Р. Гинзбург*
Перевод с английского и редакция канд. хим. наук *А.Г. Гузикевича*

По общим вопросам обращайтесь в издательство "Диалектика" по адресу:
info@dialektika.com, http://www.dialektika.com

Минник, Крис, **Холланд**, Ева.
М62 JavaScript для чайников. : Пер. с англ. — М. : ООО "И.Д. Вильямс", 2017. — 320 с. : ил. — Парал. тит. англ.

ISBN 978-5-8459-2036-2 (рус.)

ББК 32.973.26-018.2.75

Научно-популярное издание

Крис Минник, Ева Холланд

JavaScript для чайников

В издании использованы карикатуры Банка изображений Cartoonbank.ru

Литературный редактор *И.А. Попова*
Верстка *Л.В. Чернокозинская*
Художественный редактор *Е.П. Дынник*
Корректор *Л.А. Гордиенко*

Подписано в печать 15.03.2017. Формат 70x100/16
Усл. печ. л. 25,8. Уч.-изд. л. 14,98
Доп. тираж 700 экз. Заказ № 1827.

Отпечатано в АО «Первая Образцовая типография»
Филиал «Чеховский Печатный Двор»
142300, Московская область, г. Чехов, ул. Полиграфистов, д.1
Сайт: www.chpd.ru, E-mail: sales@chpd.ru, тел. 8(499)270-73-59

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

ISBN 978-5-8459-2036-2 (рус.)
ISBN 978-1-119-05607-2 (англ.)

Оглавление

Содержание

Об авторах

Крис Минник — писатель, преподаватель и веб-разработчик, автор книги *HTML5 и CSS3 для чайников*. Прежде чем основать компанию WatzThis?, Крис восемнадцать лет проработал исполнительным директором компании Minnick Web Services, где под его руководством и с его участием были разработаны сотни проектов, выполненных как по заказу небольших компаний, так и компаний с мировым именем. Крис обучил HTML, JavaScript, CSS и мобильной разработке не одну тысячу студентов.

Ева Холланд — опытный писатель, преподаватель и соучредитель компании WhatzThis?. Она превосходно умеет излагать самые сложные концепции на языке, понятном для новичков с любым уровнем подготовки. Ева занималась писательской, дизайнерской и преподавательской деятельностью в онлайновом режиме, персонально и посредством видеокурсов. Ею созданы учебные программы по веб-дизайну, мобильной разработке и поисковой оптимизации (SEO). До учреждения компании WatzThis? Ева занимала должность операционного директора компании MWS, где зарекомендовала себя как умелый руководитель, способный видеть перспективу и обеспечивать достижение стоящих перед компанией задач.

Введение

JavaScript сейчас на пике популярности. Когда-то это был наскоро скроенный язык для одного из первых браузеров, теперь же он стал самым популярным в мире языком программирования. Спрос на JavaScript-программистов как никогда высокий и продолжает неуклонно расти.

Эта книга — ваш ключ к овладению основными концепциями JavaScript. Независимо от того, стремитесь ли вы оказаться на высокооплачиваемой должности программиста или хотите создать собственный интерактивный сайт, смеем вас заверить, что содержание книги и описанные в ней методики полностью соответствуют самым последним стандартам JavaScript и наилучшей практике программирования. Каждая глава содержит примеры реального кода, которые вы сможете протестировать в браузере в домашних условиях.

Точно так же, как для музыканта единственный способ оказаться на сцене Карнеги-Холла — это репетировать, репетировать и еще раз репетировать, единственный способ стать лучшим программистом — кодировать, кодировать и еще раз кодировать!

О чем эта книга

Эта книга представляет собой доступное руководство, предназначенное для тех, кто только учится писать код на JavaScript. По сравнению с другими языками программирования JavaScript отличается простотой, и вам будет нетрудно начать его применять. Ввиду легкости освоения этого языка многие из тех, кто начинал свою деятельность как веб-дизайнер, вскоре обнаруживали, что в состоянии самостоятельно изменять и писать JavaScript-код. Если вы относитесь к этой категории людей, то книга поможет вам быстро и без лишних сложностей войти в курс дела.

Независимо от того, знакомы ли вы с JavaScript-кодом или никогда в жизни его не видели, эта книга научит вас самостоятельному созданию корректного кода.

В частности, в книге рассматриваются следующие темы:

- ✓ базовая структура JavaScript-программ;
- ✓ интеграция JavaScript с HTML и CSS;
- ✓ структурирование программ за счет использования функций;
- ✓ работа с объектами JavaScript;
- ✓ использование передовых методик JavaScript, таких как AJAX, функции обратного вызова и замыкания;
- ✓ основы jQuery.

Освоение языка JavaScript не сводится к изучению одного только синтаксиса. Вы также должны знать, как получить доступ к инструментам разработчика и присоединиться к сообществу, успевшему сформироваться вокруг этого языка. За всю длительную и увлекательную историю развития JavaScript профессиональные программисты идеально отшлифовали инструментальные средства и методики, применяемые для создания программ на этом языке. В книге мы постоянно обращаем ваше внимание на наилучшие из существующих подходов и средств, предназначенных для тестирования и документирования кода и повышения эффективности всего процесса разработки в целом.

Сделаем несколько замечаний, которые облегчат вам чтение книги.

- ✓ Весь JavaScript-код, а также разметка HTML и CSS выделяются в тексте моноширинным шрифтом, например:

```
document.write("Привет!");
```

- ✓ Поскольку размеры книжных страниц не совпадают с размерами экранов компьютерных мониторов, длинные цепочки HTML-, CSS- и JavaScript-кода могут продолжаться на нескольких строках. Помните, что компьютер рассматривает подобный код так, будто это одна строка. Мы указываем на то, что код в действительности представляет собой одну строку, разрывая его на знаке препинания или пробеле и располагая остальную часть на следующей строке с отступом примерно так, как показано ниже.

```
document.getElementById("anElementInTheDocument").
        addEventListener("click",doSomething,false);
```

- ✓ HTML и CSS не особо чувствительны к регистру букв, чего нельзя сказать о JavaScript, где это очень важно. Чтобы получать правильные результаты, используйте прописные и строчные буквы в строгом соответствии с текстом примеров.

Исходные предположения

Чтобы разбираться в программировании, вовсе не требуется быть программным экспертом или хакером. Доскональное знание работы компьютера для этого также не требуется, и вы даже не обязаны уметь считать в двоичной системе.

И все же нам придется сделать некоторые предположения относительно вас. Мы предполагаем, что вы умеете включать компьютер, знаете, как пользоваться мышью и клавиатурой, имеете подключение к Интернету и на вашем компьютере установлен браузер. Если вам уже известно кое-что о том, как создавать веб-страницы (это не так уж и сложно!), то считайте, что у вас есть небольшая фора.

Все остальные детали, которые потребуются вам для того, чтобы начать писать и выполнять программы на JavaScript, изложены в этой книге. А внимание к деталям, в чем вы сами убедитесь, — это непреложная истина программирования.

Пиктограммы, используемые в книге

Ниже приведены пиктограммы, используемые в книге для выделения фрагментов, на которые мы хотели бы особо обратить ваше внимание.

Эта пиктограмма обозначает полезные советы, подсказки и приемы, позволяющие экономить время и усилия.

Концентрируйте свое внимание всякий раз, когда вам встречается эта пиктограмма. Она говорит о том, что обозначенную ею информацию стоит запомнить.

Будьте внимательны, очень внимательны! Эта пиктограмма предупреждает о разного рода ловушках, которых следует избегать.

Эта пиктограмма обозначает информацию технического характера, которая может быть интересной для вас. Можете смело пропускать такую информацию, но если хотите узнать технические подробности, то чтение этих абзацев доставит вам удовольствие.

Дополнительные материалы

Изложенный в книге основной материал дополняют следующие информационные ресурсы.

- ✓ **Примеры.** Коды примеров, используемых в книге, можно загрузить по следующему адресу:

  `http://www.codingjsfordummies.com/code/`

- ✓ **Упражнения.** Читателям, владеющим английским языком, будет полезно проделать упражнения, подготовленные для книги на сайте `www.codecademy.com`. Чтобы получить доступ к упражнениям, посетите следующий сайт:

  `http://www.dummies.com/go/codingwithjavascript`

Что дальше

Написание программ на JavaScript доставляет большое удовольствие, и стоит вам овладеть хотя бы минимальным багажом знаний в этой области, как перед вами откроется удивительный мир интерактивных веб-приложений. Поэтому будьте смелее! Мы надеемся, что книга станет хорошим подспорьем в ваших начинаниях.

Ждем ваших отзывов!

Вы, читатель этой книги, и есть главный ее критик. 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 интересны любые ваши замечания в наш адрес.

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

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

Наши электронные адреса:

E-mail: `info@dialektika.com`
WWW: `www.dialektika.com`

Наши почтовые адреса:

в России: 127055, г. Москва, ул. Лесная, д. 43, стр. 1
в Украине: 03150, Киев, а/я 152

Часть I

Приступаем к программированию на JavaScript

В этой части...

- ✓ Написание первой программы на JavaScript
- ✓ Работа с переменными и массивами
- ✓ Операторы, выражения и инструкции
- ✓ Циклы и ветвление кода в программах на JavaScript

Глава 1

Самый неправильно понятый язык программирования в мире

В этой главе...

- Знакомство с JavaScript
- Возможности JavaScript
- Преимущества JavaScript

"Люди настолько плохо понимают меня, что даже мои переживания по поводу того, что меня не понимают, им непонятны".

Серен Кьеркегор

Язык программирования JavaScript не всегда пользовался такой большой популярностью, как в наши дни. Одни считали его лучшим языком программирования в мире, другие — наихудшим. За последние несколько лет как в методику программирования на JavaScript, так и в интерпретаторы этого языка были внесены многочисленные усовершенствования. В результате современный язык JavaScript значительно улучшился по сравнению с ранними версиями.

В этой главе вы узнаете о том, что собой представляет JavaScript, и познакомитесь с краткой историей развития и становления этого языка программирования. Кроме того, вы получите общее представление о возможностях JavaScript и сферах его применения.

Что такое JavaScript?

На заре Интернета браузеры представляли собой простые программы для чтения веб-страниц (рис. 1.1). Они практически не располагали собственными функциональными возможностями, в лучшем случае позволяя лишь изменять размер шрифта отображаемого текста. Как только компания Microsoft выпустила браузер Internet Explorer, разгорелась так называемая *война браузеров* — жесткое конкурентное сражение за господство на рынке, резко ускорившее разработку новых средств. В браузерах одна за одной стали предлагаться такие возможности, как отображение графики, использование шрифтов различных видов, мерцающий текст, бегущая строка и многие другие.

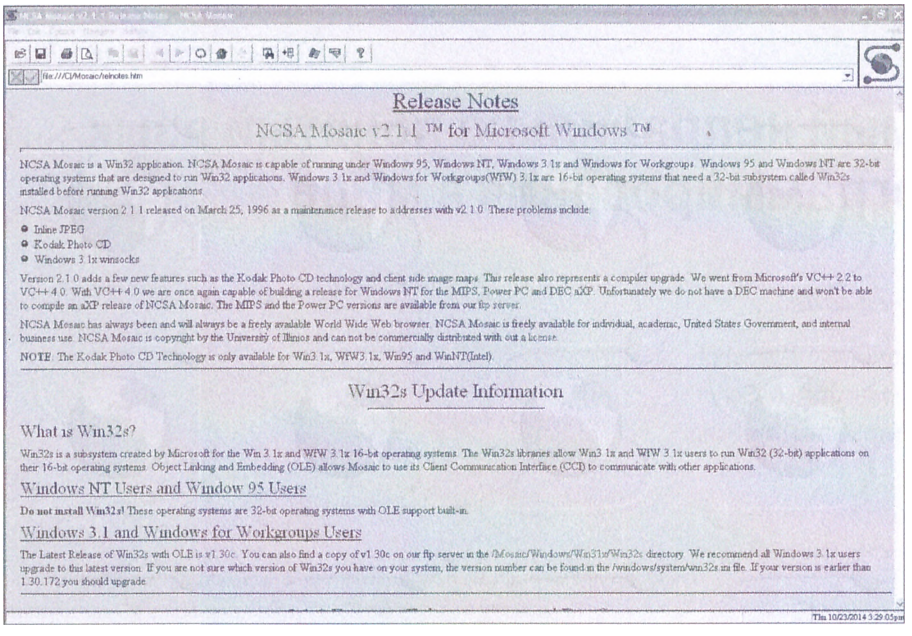


Рис. 1.1. Непритязательный внешний вид веб-страницы в окне одного из первых браузеров

Прошло совсем немного времени, пока одного человека не осенила мысль, что браузер может не только быть программой для отображения текста в красивом виде, но и самостоятельно выполнять массу полезных вещей.

И тут пришел Эйх...

Проект JavaScript стартовал в компании Netscape в далеком 1995 году. Брендан Эйх, создатель языка JavaScript, написал его в рекордно короткие сроки (говорят, что для этого ему понадобилось всего лишь десять дней!), заимствуя все лучшее из ряда других языков программирования. Из-за спешки, вызванной стремлением как можно быстрее выйти на рынок с готовым продуктом, в язык вкрались некоторые, скажем так, "странности" (а если говорить без обиняков — ошибки). Результат представлял собой своеобразный язык наподобие эсперанто, который обманчиво воспринимался людьми, имеющими опыт работы с другими языками программирования, как язык, с которым они уже знакомы.

Сначала был Mocha

Язык JavaScript первоначально назывался Mocha. С выпуском первой бета-версии браузера Netscape Navigator он был переименован в LiveScript, но в версию браузера Netscape 2, выпущенную в 1995 г., он был встроен уже как язык JavaScript. Оперативно воспользовавшись методами обратного инжиниринга (обратной разработки), компания Microsoft встроила в свой браузер Internet Explorer точную копию JavaScript, назвав ее Jscript, чтобы избежать проблем, связанных с незаконным использованием торговой марки.

Компания Netscape представила язык JavaScript в организацию Ecma International, занимающуюся стандартизацией информационных и коммуникационных технологий, где он был утвержден в качестве стандарта ECMAScript в 1997 г.

Брендан Эйх, создатель JavaScript, отпустил по поводу названия стандартизированного языка нашумевшее замечание, в котором охарактеризовал ECMAScript как "неудачное название торговой марки, звучащее подобно названию кожного заболевания".

Признавая ECMAScript неудачным названием для языка программирования, можно сказать то же самое и в отношении названия JavaScript, которое было дано языку компанией Netscape и используется большинством тех, кто с ним работает. Если вы уже знаете, как писать программы на языке Java, или планируете этому научиться, то должны отдавать себе отчет в том, что, несмотря на определенное сходство между ними, в действительности Java и JavaScript — это два совершенно разных языка.

Даешь больше эффектов!

Сразу же после своего дебюта язык JavaScript быстро приобрел популярность в качестве средства, позволяющего делать веб-страницы более динамичными. Одним из первых результатов встраивания JavaScript в браузеры стало появление так называемого динамического HTML (DHTML) — языка разметки веб-страниц, обеспечившего реализацию не только всевозможных забавных эффектов, таких как падающие снежинки (рис. 1.2), всплывающие окна и фигурные уголки страниц, но и более полезных вещей, например раскрывающихся меню и средств валидации (проверки корректности данных) веб-страниц.

Рис. 1.2. Эффект падающих снежинок на веб-странице, созданный с помощью JavaScript

JavaScript взрослеет

Разменяв третий десяток лет, JavaScript стал наиболее широко используемым языком программирования в мире, и практически на каждом из эксплуатируемых в настоящее время персональных компьютеров установлен по крайней мере один браузер, способный выполнять JavaScript-код.

Язык JavaScript достаточно гибок для того, чтобы быть пригодным для использования непрограммистами, и одновременно располагает достаточно мощными возможностями, которых профессиональным программистам вполне хватает для того, чтобы поддерживать функциональность практически любого современного сайта в Интернете, начиная с одностраничных сайтов и кончая такими "монстрами", как сайты Google, Amazon, Facebook и многие другие.

Динамический язык сценариев

JavaScript часто описывают как *динамический сценарный язык*, т.е. язык для написания сценариев (жарг. *скриптовый язык*). Чтобы смысл этого стал вам понятен, потребуется сначала определить пару терминов и разобраться с контекстом применения языка.

Распространенные заблуждения относительно JavaScript

За долгие годы существования JavaScript в его адрес было высказано немало острых замечаний. Некоторые из ходящих о нем слухов довольно интересны, однако не всегда справедливы. Ниже перечислены некоторые распространенные заблуждения, касающиеся JavaScript.

- **Миф:** JavaScript не является настоящим языком программирования. **Реальность:** JavaScript часто используют для выполнения простейших задач в браузерах, но это никоим образом не лишает его статуса полноценного языка программирования. В действительности многие передовые характеристики JavaScript установили высокую планку для языков программирования и теперь имитируются во многих языках, включая PHP, C++ и даже Java.
- **Миф:** JavaScript родствен Java. **Реальность:** вовсе нет. Название JavaScript было выбрано исключительно из маркетинговых соображений, поскольку на момент выхода JavaScript язык Java пользовался невероятной популярностью.
- **Миф:** JavaScript — новый язык. **Реальность:** некоторые из известных нам профессиональных программистов на JavaScript к моменту создания этого языка даже не успели появиться на свет.
- **Миф:** В движке JavaScript имеются ошибки, и в разных браузерах он выполняется поразному. **Реальность:** хотя в некоторых случаях это действительно является правдой, производители браузеров давным-давно приняли решение поддерживать стандартизированную версию JavaScript. В наше время JavaScript выполняется одинаково во всех браузерах.

Компьютерные программы — это наборы инструкций, с помощью которых можно заставить компьютер выполнять определенные действия. Каждый компьютерный язык программирования располагает собственным набором инструкций и правилами их записи, которыми должны руководствоваться программисты. Компьютер неспособен понимать эти инструкции в непосредственном виде. Чтобы компьютер мог понимать

язык программирования, требуется предварительный процесс преобразования инструкций, в ходе которого они транслируются (переводятся) из формы, удобной для чтения (и записи) человеком, в машинный язык. В зависимости от того, когда именно осуществляется процесс трансляции инструкций, языки программирования делятся на два основных типа: компилируемые и интерпретируемые (рис. 1.3).

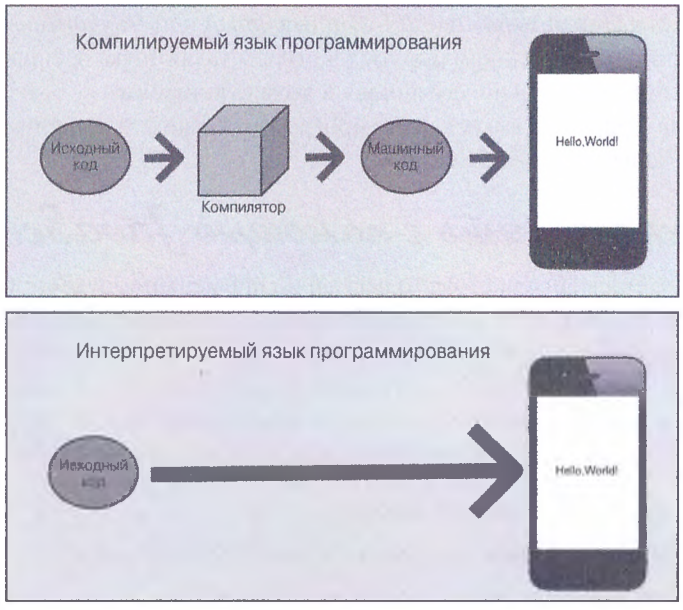


Рис. 1.3. Классификация языков программирования в зависимости от того, когда именно осуществляется компиляция программы

Компилируемые языки программирования

Компилируемые языки программирования — это языки, которые требуют, чтобы написанный программистом код был обработан специальной программой, так называемым *компилятором*, который анализирует данный код, а затем преобразует его в машинный язык. После этого компьютер может выполнить скомпилированную программу.

В качестве примера компилируемых языков программирования можно привести C, C++, Fortran, Java, Objective-C и COBOL.

Интерпретируемые языки программирования

Интерпретируемые языки программирования — это языки, которые также компилируются компьютером в машинный язык, но процесс компиляции осуществляется в браузере пользователя непосредственно во время выполнения программы. Программистам, которые пишут программы с использованием интерпретируемых языков, нет нужды компилировать свой код, прежде чем передавать его компьютеру для выполнения.

Программирование с использованием интерпретируемых языков обладает тем преимуществом, что изменения могут быть легко внесены в программу в любой момент.

Но у таких языков есть и недостаток, заключающийся в том, что компиляция кода во время его выполнения создает дополнительную стадию процесса, которая может замедлять выполнение программ.

Отчасти именно из-за такого снижения производительности интерпретируемые языки получили репутацию не очень серьезных языков программирования. Однако с появлением улучшенных компиляторов, осуществляющих компиляцию кода на стадии его выполнения (так называемые *JIT-компиляторы*[1] или *оперативные компиляторы*), и процессоров с повышенным быстродействием такая точка зрения очень быстро устарела. JavaScript значительно улучшился в этом отношении.

Примерами интерпретируемых языков программирования могут служить PHP, Perl, Haskell, Ruby и, конечно же, JavaScript.

Что можно делать с помощью JavaScript?

Если вы пользуетесь Интернетом, то регулярно применяете JavaScript. Список того, что позволяет делать JavaScript, довольно обширен и охватывает как простые напоминания, которые вы получаете всякий раз, когда забываете заполнить обязательное поле формы, так и сложные приложения, такие как Google Docs или Facebook. Вот лишь краткий перечень наиболее распространенных вариантов применения JavaScript:

- ✓ эффекты;
- ✓ проверка вводимых данных;
- ✓ трансформация изображения при наведении мыши;
- ✓ раскрывающиеся и выдвигающиеся меню;
- ✓ возможности перетаскивания и вставки;
- ✓ бесконечная прокрутка веб-страниц;
- ✓ автозаполнение;
- ✓ индикаторы выполнения;
- ✓ вкладки внутри веб-страниц;
- ✓ сортируемые списки;
- ✓ Magic Zoom (рис. 1.4).

Почему именно JavaScript?

Язык JavaScript стал стандартом для создания динамических пользовательских интерфейсов интернет-приложений. Всякий раз, когда вы посещаете веб-страницу с анимацией, автоматически обновляемыми данными, кнопкой, изменяющей свой вид при наведении на нее мыши, или раскрывающимся меню, в большинстве случаев вы имеете дело с результатами работы JavaScript. Благодаря тому, что программы, написанные на JavaScript, могут применяться для решения различных задач и способны

[1] От англ. "just-in-time" — непосредственно в момент возникновения необходимости. — *Примеч. ред.*

выполняться в любом браузере, программирование на JavaScript приобрело необычайную популярность, а владение этим языком стало обязательным требованием в современной веб-разработке.

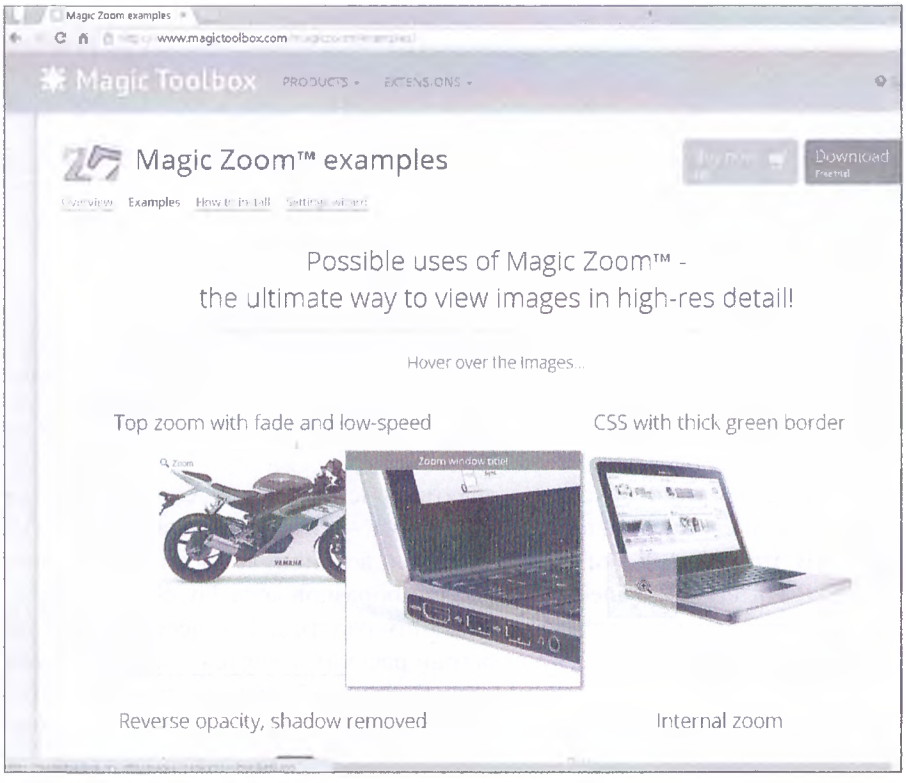


Рис. 1.4. Эффект Magic Zoom, активизированный с помощью JavaScript

Язык JavaScript легок в изучении

Языки программирования создавались для того, чтобы предоставить людям простой способ общения с компьютерами и возможность отдания им приказов относительно того, что именно они должны делать. По сравнению с машинным языком, т.е. языком, на котором "говорит" центральный процессор (CPU) компьютера, любой язык программирования гораздо проще и понятнее. Чтобы вы могли получить представление о том, какого рода инструкциям в действительности подчиняется ваш компьютер, взгляните на приведенный ниже образец программы на машинном языке, которая выводит строку "Hello World".

```
b8    21 0a 00 00
a3    0c 10 00 06
b8    6f 72 6c 64
a3    08 10 00 06
b8    6f 2c 20 57
a3    04 10 00 06
```

```
b8    48 65 6c 6c
a3    00 10 00 06
b9    00 10 00 06
ba    10 00 00 00
bb    01 00 00 00
b8    04 00 00 00
cd    80
b8    01 00 00 00
cd    80
```

А теперь посмотрите, как можно решить эту же простую задачу с помощью JavaScript:

```
alert("Hello World");
```

Намного проще, не правда ли?

Как только вы изучите основные правила дорожного движения (называемые *синтаксисом*), например, в каких ситуациях следует использовать круглые или фигурные скобки, JavaScript станет для вас столь же привычным, как обычный разговорный язык.

Первое, что необходимо сделать, приступая к изучению любого языка, в том числе и языка программирования, — это избавиться от страха неудачи. В этом отношении у вас не должно возникнуть сложностей с JavaScript. В Интернете существуют тысячи образцов кода JavaScript, любой из которых можно скачать и начать с ним работать. Все необходимые для этого инструменты у вас уже есть (они рассматриваются в главе 2), и вам остается лишь начать изучение JavaScript с небольших примеров, чтобы впоследствии постепенно дорабатывать их для решения полезных задач.

Где работает JavaScript? JavaScript работает везде!

Изначально язык JavaScript предназначался для использования в браузерах, однако он нашел применение во многих местах. В наши дни JavaScript используется в смартфонах и планшетах, в веб-серверах и в настольных приложениях, а также во всякого рода мобильных устройствах.

JavaScript в браузере

Чаще всего JavaScript можно обнаружить там, где его первоначально предполагалось использовать, — в веб-браузерах. В подобных случаях о JavaScript говорят как о *клиентском JavaScript* или *JavaScript на стороне клиента.*

Клиентский JavaScript позволяет сделать веб-страницы интерактивными. Это осуществляется следующими способами:

- ✓ путем управления самим браузером или использования его функциональности;
- ✓ манипулированием структурой и содержимым веб-страниц;

- ✓ манипулированием стилями отображения (например, шрифтами и компоновкой) веб-страниц;
- ✓ доступом к данным из других источников.

Чтобы понять, как JavaScript манипулирует структурой и стилями веб-страниц, надо немного знать HTML5 и CSS3.

HTML5

Язык гипертекстовой разметки (Hypertext Markup Language — HTML) — это язык, применяемый для структурирования веб-страниц. Принцип его работы заключается в разметке содержимого (текста и изображений) с целью предоставления браузеру информации о расположении заголовков, абзацев, изображений и т.п. Пример простого HTML-документа приведен в листинге 1.1. На рис. 1.5 показано, как этот документ отображается в браузере.

Листинг 1.1. Простой HTML-документ

```
<!DOCTYPE html>
<html>
<head>
  <title>Привет, HTML!</title>
</head>
<body>
  <h1>Это HTML</h1>
  <p id="introduction">Этот простой документ написан с
          использованием языка разметки гипертекста.</p>
</body>
</html>
```

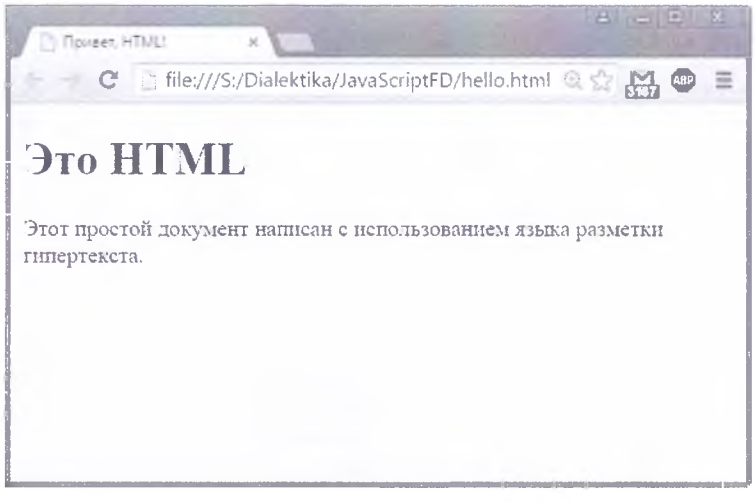


Рис. 1.5. Вид веб-страницы, визуализированной в окне браузера с помощью HTML

Ниже перечислено все, что вам надо знать прямо сейчас для того, чтобы вы могли продвигаться дальше в изучении JavaScript.

- ✓ В HTML символы, заключенные в угловые скобки, называются *тегами*, или *дескрипторами*.
- ✓ *Конечный*, или *закрывающий, тег* (тот, который идет после размечаемого содержимого) содержит символ обратной косой черты после первой угловой скобки. Например, `</p>` — конечный тег.
- ✓ Группа из двух тегов (начального и конечного) вместе с заключенным между ними содержимым называется *элементом*.
- ✓ Обычно элементы организуются в виде иерархической структуры (в которой одни элементы вложены в другие).
- ✓ Элементы могут содержать пары *"имя-значение"*, называемые *атрибутами*. Если элемент имеет атрибуты, то они указываются в начальном теге. В парах *"имя–значение"* присваивание значений (заключаемых в кавычки) именам (не заключаются в кавычки) осуществляется путем помещения между ними знака равенства. Например, в следующем теге как `width`, так и `height` являются атрибутами элемента `div`:

  ```
  <div width="100" height="100"></div>
  ```

- ✓ Некоторые элементы не имеют содержимого и поэтому не нуждаются в конечном теге. Например, тег `img`, с помощью которого осуществляется вставка изображения на страницу, выглядит так:

  ```
  <img src="myimage.jpg" width="320" height="200"
           alt="Это фотография моей собаки.">
  ```

Все данные, необходимые для вывода изображения, включены в начальный тег с помощью атрибутов, и поэтому для тега `img` конечный тег использовать не требуется.

При описании веб-страницы с помощью HTML можно включать в документ либо сам JavaScript-код, либо ссылку на файл (с расширением `.js`), в котором содержится этот код. В любом случае браузер загрузит JavaScript-код и выполнит его, как только пользователь обратится к веб-странице, содержащей JavaScript-сценарий.

Клиентский JavaScript выполняется в браузере пользователя.

CSS3

Каскадные таблицы стилей (Cascading Style Sheets — CSS) — это язык, используемый для описания форматирования и компоновки веб-страниц. Применительно к CSS термин *стиль* охватывает целую совокупность свойств, определяющих способ представления документа пользователю, включая следующие:

- ✓ гарнитура шрифта;
- ✓ размер шрифта;
- ✓ цвет;
- ✓ расположение элементов в окне браузера;
- ✓ размеры элементов;
- ✓ границы;
- ✓ фон;
- ✓ создание закругленных углов на границах элементов.

Как и в случае JavaScript, CSS-описания могут либо непосредственно помещаться в HTML-документ, либо подключаться к нему с помощью соответствующей ссылки. Сразу же после загрузки стиля он выполняет назначенные ему функции и форматирует документ в соответствии со своими спецификациями.

В CSS таблицы стилей состоят из правил CSS, содержащих свойства и значения, которые должны быть применены к элементу или группе элементов. Вот пример записи одного из правил CSS:

```
p{font-size: 14px; font-color: black; font-family: Arial, sans-serif}
```

В этом правиле, которое следует читать слева направо, указано, что во всех элементах `p` (которым в HTML-документе соответствуют абзацы) для отображения текста должен использоваться шрифт Arial размером 14 пикселей черного цвета. Если в компьютере пользователя шрифт Arial не установлен, то для отображения текста будет использоваться одна из гарнитур типа Sans Serif.

Часть CSS-правила, расположенная вне фигурных скобок, называется *селектором*. Она выбирает элементы, к которым должны применяться свойства, указанные в фигурных скобках.

На протяжении всей книги вам будут постоянно встречаться примеры совместного использования JavaScript, HTML и CSS. В этой книге мы предоставляем ровно столько информации, сколько необходимо для того, чтобы продемонстрировать вам принципы работы HTML и CSS. Для более подробного изучения HTML и CSS существуют другие прекрасные книги, например *HTML5 и CSS3 для чайников*.

JavaScript — мощный язык

Раньше JavaScript выполнялся в браузерах очень медленно, чем заслужил среди программистов плохую славу. В наши дни код на JavaScript выполняется всего на 20% медленнее, чем скомпилированный код, причем с течением времени данный показатель только улучшается. Это означает, что современный JavaScript обладает намного большей производительностью, чем те его версии, которые существовали всего лишь несколько лет тому назад, не говоря уже о первоначальном варианте, выпущенном в 1995 году.

JavaScript — востребованный язык

JavaScript — не просто самый известный язык программирования. Специалисты, владеющие этим языком, наиболее востребованы на рынке труда. Согласно некоторым прогнозам, за период с 2010 по 2020 год рост количества рабочих мест для специалистов этой категории составит 22%. JavaScript бурно развивается, и лучше начать изучать его прямо сейчас.

Глава 2

Моя первая программа на JavaScript

В этой главе...

- Организация среды разработки
- Работа с JavaScript-кодом
- Пример простой программы на JavaScript
- О важности комментирования кода

"Секрет неуклонного движения вперед в том, чтобы сделать первый шаг".

Марк Твен

Научиться писать на JavaScript простые программы совсем не сложно. В этой главе мы проведем вас через весь процесс настройки компьютера для написания JavaScript-кода. Кроме того, вы напишете свою первую программу на JavaScript и ознакомитесь с базовым синтаксисом, лежащим в основе всего, что вы будете делать с помощью JavaScript на протяжении своей будущей карьеры программиста.

Настройка среды разработки

Прежде чем приступить к написанию первой программы на JavaScript, очень важно заранее позаботиться о том, чтобы все необходимые для этого инструменты были настроены и находились на своих местах. Мы проведем вас через весь процесс загрузки и установки наших излюбленных средств разработки на JavaScript, которые, естественно, используются и в этой книге. Если вы уже располагаете аналогичными инструментами, которые находите более удобными для себя, то можете смело использовать их. Однако в любом случае мы рекомендуем вам прочитать этот раздел, чтобы узнать о тех соображениях, которыми мы руководствовались при выборе инструментов, и уже после этого принимать решение относительно того, стоит или не стоит их использовать.

Закончив с установкой инструментальных средств, мы поделимся с вами некоторыми советами и подсказками относительно наиболее эффективных способов их использования.

Загрузка и установка браузера Chrome

Для работы с JavaScript мы предпочитаем использовать браузер Google Chrome. Если в повседневной работе вы привыкли пользоваться другим браузером, то, конечно же, ничего предосудительного в этом нет. Во всех браузерах JavaScript работает быстро и корректно. Однако некоторые из описанных в данной книге инструкций специфичны для браузера Google Chrome. Поэтому рекомендуется, чтобы вы в любом случае установили его на своем компьютере, следуя приведенному ниже описанию. Мы используем в этой книге Google Chrome по той причине, что в нем предлагаются великолепные инструменты, значительно облегчающие жизнь программистам, а также потому, что в настоящее время именно этот браузер является лидером по использованию в Интернете. (Это действительно так — он даже более популярен, чем Internet Explorer.)

Если на вашем компьютере браузер Chrome еще не установлен, то выполните процедуру его установки, следуя приведенному ниже описанию.

1. **Выполните переход по адресу `www.google.com/chrome`.**

   Вы будете автоматически перенаправлены на страницу, которая представлена на рис. 2.1.

2. **Щелкните на кнопке Скачать Chrome и следуйте экранным инструкциям.**
3. **Откройте загруженный файл и следуйте дальнейшим инструкциям по установке Chrome.**

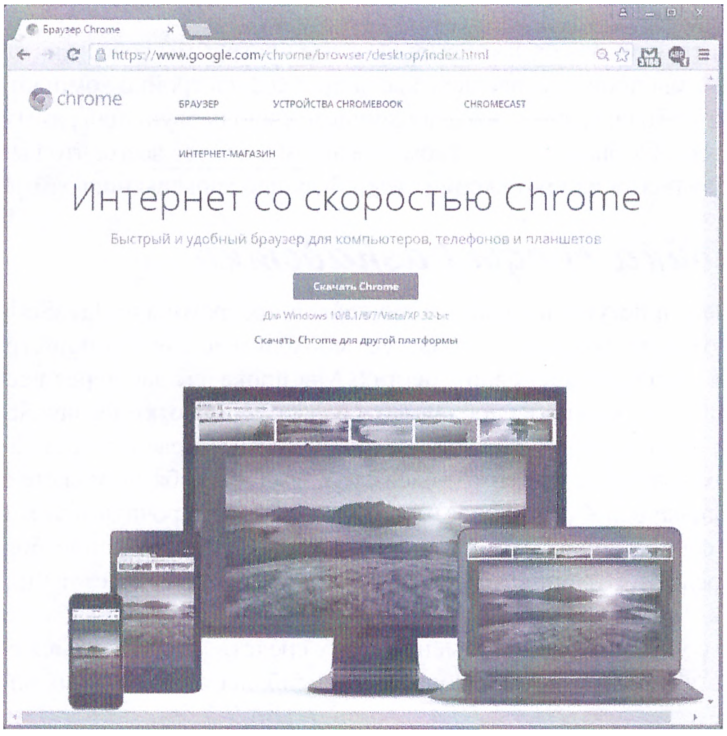


Рис. 2.1. Установка браузера Chrome не составляет большого труда

Теперь у вас имеется движок JavaScript с турбонаддувом!

В Google Chrome парсинг (синтаксический анализ), компиляция и выполнение JavaScript-кода осуществляются с помощью разработанного компанией Google JavaScript-движка V8. В зависимости от того, чьим эталонным тестам доверять больше, можно считать, что Chrome является либо браузером-лидером, либо одним из браузеров, обеспечивающих самое высокое быстродействие при выполнении JavaScript-кода. Основные производители браузеров постоянно соревнуются между собой, стремясь превзойти соперников. Фактически не имеет особого значения, чей браузер в какой-то определенный момент времени оказывается самым быстрым. Важно лишь то, что в результате конкуренции производительность движков JavaScript во всех браузерах стремительно увеличивается на протяжении всех последних лет.

Если вас интересуют фактические данные сравнительного тестирования движков JavaScript в различных браузерах, то посетите сайт `http://arewefastyet.com` (данные тестов там представлены в виде графиков), который поддерживается компанией Mozilla, создавшей браузер Firefox. Этот сайт автоматически проверяет и отображает в графическом виде результаты тестирования производительности движков JavaScript в наиболее популярных моделях браузеров, обновляя данные несколько раз на протяжении дня.

Загрузка и установка редактора исходного кода

Редактор исходного кода, или просто *редактор кода*, — это текстовый редактор с дополнительной функциональностью, упрощающей написание и редактирование программного кода. В качестве такового мы будем использовать редактор Sublime Text.

Учитывая, что спектр выбора редакторов кода очень широк, не исключено, что вы уже пользуетесь одним из них и успели к нему привыкнуть. Вам ничто не мешает и далее продолжать использовать его. Выбор редактора кода определяется исключительно личными предпочтениями, и, наверное, многие из вас уже привыкли работать с каким-то определенным редактором. Если по каким-либо причинам редактор Sublime Text вам не подходит, то просмотрите другие возможные варианты выбора, представленные в табл. 2.1.

Таблица 2.1. Редакторы исходного кода

Название	Адрес для загрузки	Совместимость
Coda	`http://panic.com/coda`	Mac
Aptana	`www.aptana.com`	Mac, Windows
Komodo Edit	`www.activestate.com/komodo-edit/downloads`	Mac, Windows
Dreamweaver	`http://adobe.com/products/dreamweaver.html`	Mac, Windows
Eclipse	`www.eclipse.org`	Mac, Windows
Notepad++	`http://notepad-plus-plus.org`	Windows
TextMate	`http://macromates.com`	Mac
BBEdit	`www.barebones.com/products/bbedit`	Mac
EMacs	`www.gnu.org/software/emacs`	Mac, Windows
TextPad	`www.textpad.com`	Windows
vim	`www.vim.org`	Mac, Windows
Netbeans	`https://netbeans.org`	Mac, Windows

Для этой книги мы выбрали редактор Sublime Text (рис. 2.2), поскольку он популярен среди программистов на JavaScript и предоставляет простой пользовательский интерфейс наряду с большим количеством плагинов, которыми пользователи после приобретения определенных навыков могут воспользоваться для решения более сложных задач программирования.

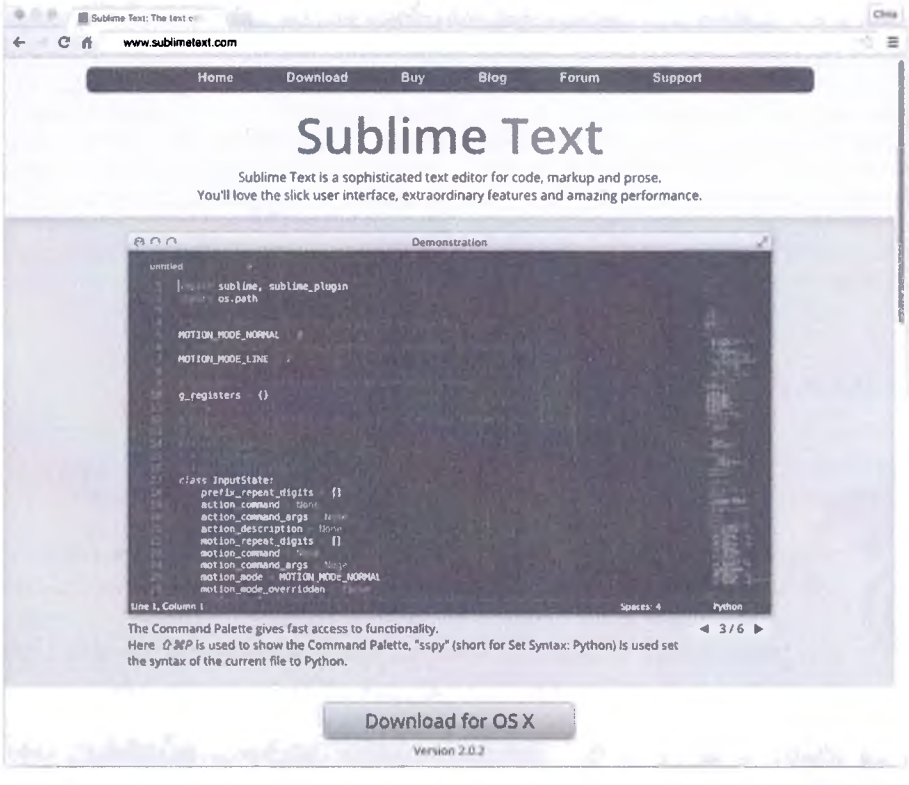


Рис. 2.2. С виду обычный текстовый редактор, Sublime Text располагает мощными функциональными возможностями

Если вы хотите установить Sublime Text, то следуйте приведенным ниже инструкциям.

1. **Посетите сайт `http://sublimetext.com` и выберите версию, совместимую с установленной на вашем компьютере операционной системой.**
2. **Откройте загруженный файл и следуйте дальнейшим инструкциям по установке Sublime Text.**

Начало работы с Sublime Text

Когда вы впервые откроете приложение Sublime Text, на экране отобразится пустая страница с мерцающим курсором (рис. 2.3).

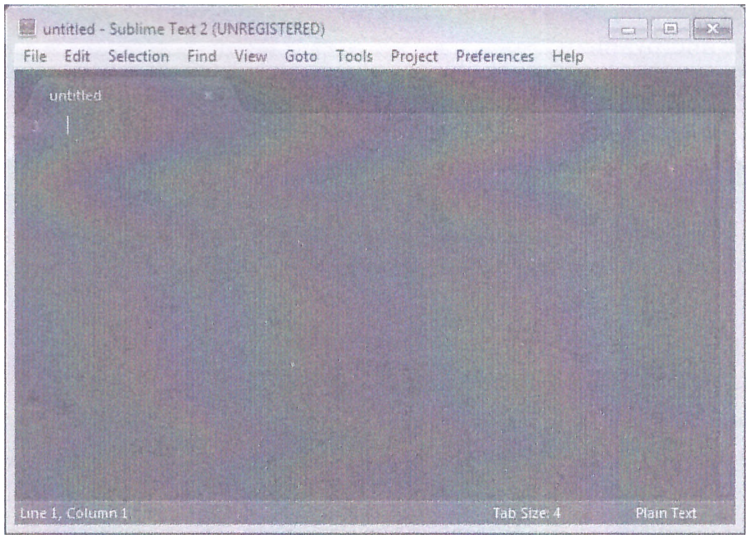


Рис. 2.3. Начальный интерфейс редактора Sublime Text. Как вам эта простота?

Если вы уже использовали Sublime Text, то слева может появиться боковая панель (рис. 2.4), на которой отображаются открытые файлы, а также файлы проекта, если таковой был создан ранее. Боковая панель упрощает работу, и мы рекомендуем держать ее открытой.

Чтобы открыть боковую панель, выберите пункты меню View⇨Sidebar⇨Show Sidebar (Вид⇨Боковая панель⇨Показать).

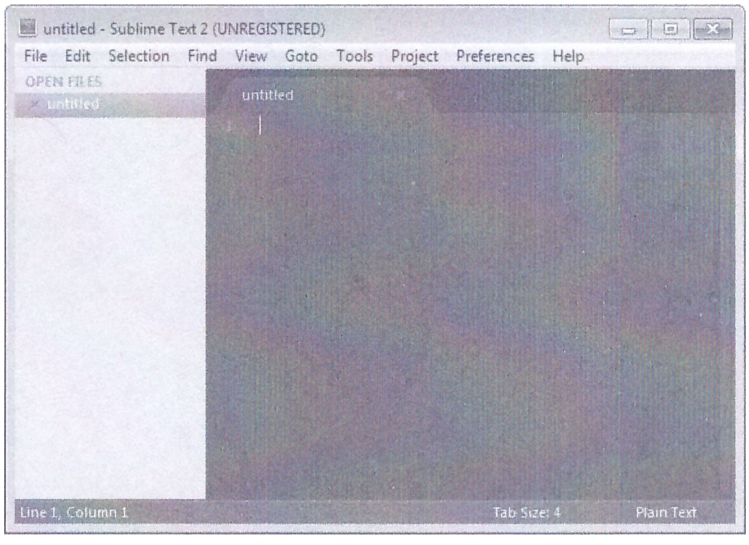


Рис. 2.4. Окно Sublime Text с открытой боковой панелью

Чтобы приступить к работе с вашим первым проектом Sublime Text, следуйте приведенным ниже инструкциям.

1. **Выберите пункты меню File⇨Save As (Файл⇨Сохранить как).**

   В открывшемся диалоговом окне отобразится заданная по умолчанию папка для сохранения файлов. Если предложенный вариант вас удовлетворяет (скорее всего, это будет папка `Documents` (Документы) в случае OS X или папка `My Documents` (Мои документы) в случае Windows), перейдите к выполнению п. 2. В противном случае перейдите в другое расположение на жестком диске компьютера, в котором хотите хранить свои файлы с исходным кодом.

2. **Создайте новую папку и присвойте ей имя.**

3. **Введите в поле Имя файла нужное имя файла и щелкните на кнопке Сохранить.**

   Имя нового файла отобразится в боковой панели, а также на открытой вкладке вместо ее прежнего имени.

4. **Выберите пункты меню Project⇨Save Project As (Проект⇨Сохранить как) и сохраните файл проекта Sublime Text в папке, которую создали перед этим.**

5. **Выберите пункты меню Project⇨Add Folder to Project (Проект⇨Добавить папку в проект), выберите папку, созданную в п. 1, и щелкните на кнопке Open (Открыть).**

   На боковой панели появится новый сворачиваемый список Folders (Папки), в котором отобразится ваша папка вместе с ее содержимым (включая файл проекта и файл `MyFirstProgram.html`), как показано на рис. 2.5.

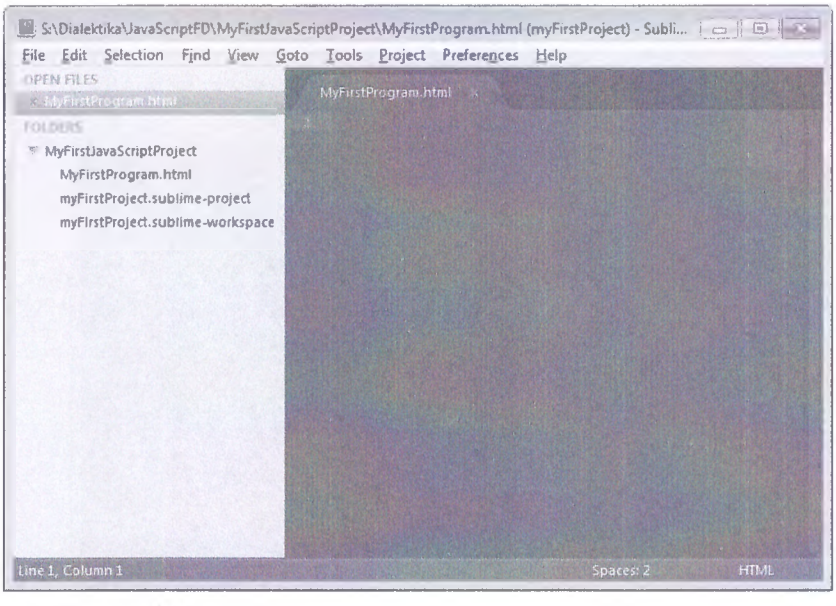


Рис. 2.5. Ваш первый проект Sublime Text подготовлен к работе

Чтобы ваши файлы и папки хранились в организованном виде, рекомендуем придерживаться определенной системы при присвоении им имен. Например, папку из п. 2 можно назвать `MyFirstJavaScriptProject`, файл из п.3 — `MyFirstProgram`, а проект из п. 4 — `myFirstProject`.

Выбор цветовой схемы для подсветки синтаксиса

В Sublime Text цветовая подсветка синтаксиса основана на типе кода, который вы пишете, и расширении имени файла. Чтобы увидеть цветовую схему, заданную по умолчанию, введите в только что созданный вами файл код на основе HTML и JavaScript, представленный в листинге 2.1.

JavaScript требует педантичного отношения к себе, в чем вы вскоре сами сможете убедиться. Текст необходимо вводить в точности так, как показано в листинге, строго соблюдая регистр букв, иначе ваш сценарий вообще не будет работать, а если и будет, то неправильно.

Листинг 2.1. Пример HTML-файла, содержащего сценарий JavaScript

```
<!DOCTYPE html>
<html>
<head>
  <title>Привет, HTML!</title>
  <script>
    function countToTen(){
      var count = 0;
      while (count < 10) {
        count++;
        document.getElementById("theCount").innerHTML +=
          count + "<br>";
      }
    }
  </script>
</head>
<body onload="countToTen();">
  <h1>Посчитаем до 10 вместе с JavaScript!</h1>
  <p id="theCount"></p>
</body>
</html>
```

На рис. 2.6 показано, как этот файл отображается в окне Sublime Text.

Если текущая цветовая схема вас не устраивает, то ее можно изменить, выбрав пункты меню Preferences⇨Color Scheme (Установки⇨Цветовая схема) и задав другую схему.

Просмотрите несколько цветовых схем и выберите ту, которая вам больше всего по душе. В этой книге используется цветовая схема Monokai Bright.

Если хотите проверить работу программы, которую только что ввели, то выполните следующие действия.

1. **Сохраните файл, выбрав пункты меню File⇨Save (Файл⇨Сохранить).**
2. **Откройте браузер Chrome и нажмите комбинацию клавиш <Ctrl+O>.**
   Откроется окно Открыть.
3. **Перейдите к файлу на своем компьютере и выберите его.**
4. **Щелкните на кнопке Open (Открыть).**
   Содержимое файла отобразится в окне браузера.

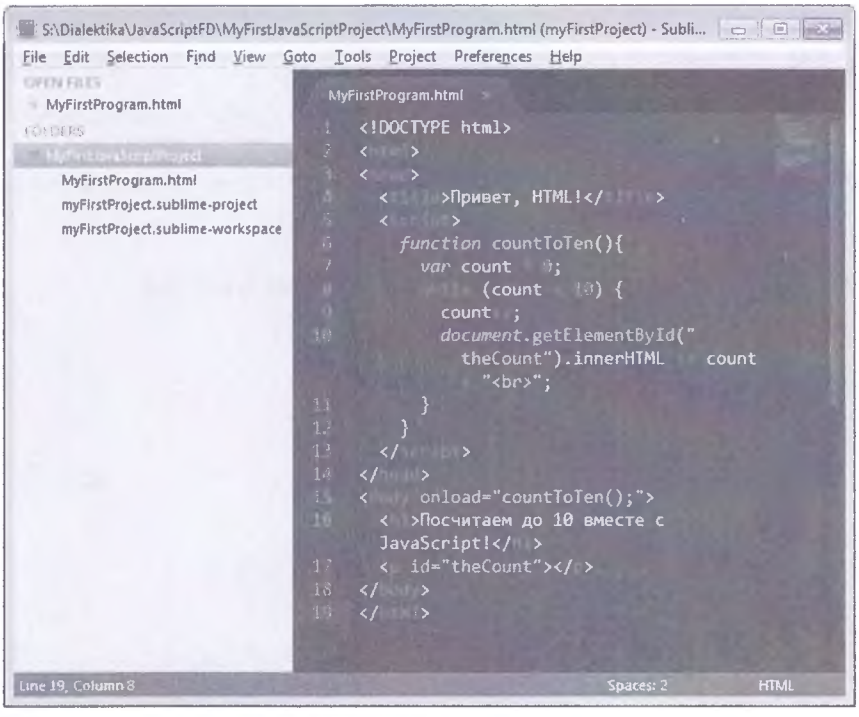


Рис. 2.6. Применение цветовой подсветки кода в окне редактора Sublime Text

Окно вашего браузера должно выглядеть так, как показано на рис. 2.7. Если оно будет иметь другой вид, тщательно проверьте свой код — возможно, вы где-то допустили опечатку. После внесения необходимых изменений не забудьте сохранить файл!

Для сохранения файлов можно также использовать комбинации клавиш <Command+S> (Mac) и <Ctrl+S> (Windows). Когда вы привыкнете с ними работать, они сэкономят вам массу времени.

Полезные комбинации клавиш в Sublime Text

Редактор кода Sublime Text на первый взгляд кажется простым текстовым редактором, но это обманчивое впечатление. Истинным показателем профессионализма программиста является его умение эффективно использовать комбинации клавиш, обеспечивающие значительную экономию времени при редактировании исходного кода.

В редакторе Sublime Text предусмотрено большое количество комбинаций клавиш (так называемых "горячих клавиш"), частичный перечень которых приведен в табл. 2.2. Освойте их, и вскоре вы поразите друзей и коллег тем, как быстро вы работаете.

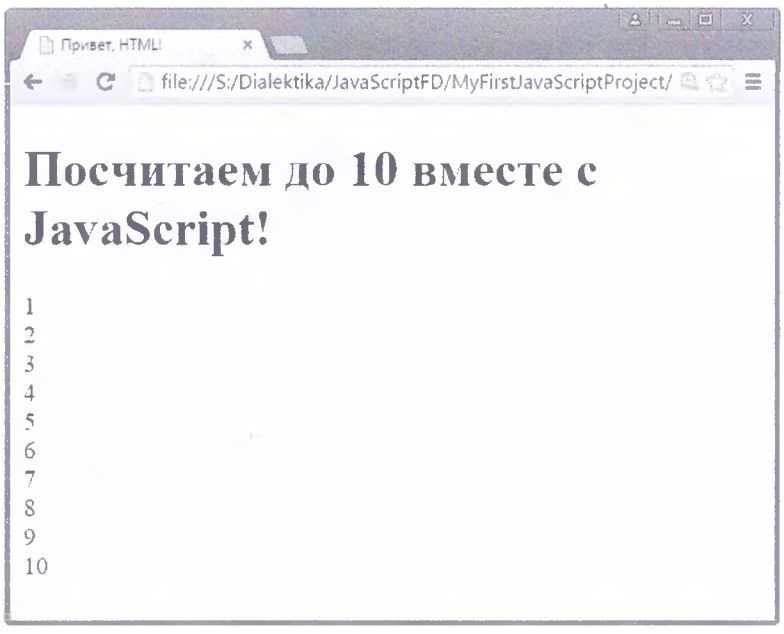


Рис. 2.7. Выполнение простой программы подсчета в Chrome

Таблица 2.2. Часто используемые комбинации клавиш для работы с текстом в редакторе Sublime Text

Mac	Windows	Описание
<Command+X>	<Ctrl+X>	Удалить строку
<Command+Return>	<Ctrl+Enter>	Вставить строку снизу
<Command+]>	<Ctrl+Shift+Enter>	Вставить строку сверху
<Command+Control+↑>	<Ctrl+Shift+↑>	Переместить строку/выделенный объект вверх
<Command+Control+↓>	<Ctrl+Shift+↓>	Переместить строку/выделенные строки вниз
<Command+L>	<Ctrl+L>	Выделить строку; повторить для выделения следующих строк
<Command+D>	<Ctrl+D>	Выделить слово; повторить для выделения других вхождений этого же слова
<Control+M>	<Ctrl+M>	Перейти к закрывающей скобке; повторить для перехода к открывающей скобке
<Control+Shift+M>	<Ctrl+Shift+M>	Выделить все содержимое в текущих скобках
<Command+K+Command+K>	<Ctrl+K+K> <Ctrl+Shift+Delete>	Удалить текст от текущей позиции курсора до конца строки
<Command+K+Delete>	<Ctrl+K+Backspace> <Ctrl+Shift+Backspace>	Удалить текст от текущей позиции курсора до начала строки

Окончание табл. 2.2

Mac	Windows	Описание
<Command+]>	<Ctrl+]>	Увеличить отступ текущей строки (выделенных строк)
<Command+[>	<Ctrl+[>	Уменьшить отступ текущей строки (выделенных строк)
<Command+Shift+D>	<Ctrl+Shift+D>	Дублировать строку
<Command+J>	<Ctrl+J>	Присоединить следующую строку к концу текущей строки
<Command+/>	<Ctrl+/>	Закомментировать/раскомментировать текущую строку (выделенные строки)
<Command+Option+/>	<Ctrl+Shift+/>	Закомментировать/раскомментировать блок выделенного текста
<Command+Y>	<Ctrl+Y>	Повторить последнее действие
<Command+Shift+V>	<Ctrl+Shift+V>	Вставить с отступом
<Control+пробел>	<Ctrl+пробел>	Выбор варианта автозаполнения
<Control+U>	<Ctrl+U>	"Мягкая" отмена; переход к месту внесения последнего изменения и отмена изменения после повторного применения этой команды
<Control+Shift+↑>	<Ctrl+Alt+↑>	Добавить курсор на предыдущей строке
<Control+Shift+↓>	<Ctrl+Alt+↓>	Добавить курсор на следующей строке
<Control+Shift+W>	<Alt+Shift+W>	Заключить выделенный текст в тег HTML

Оформление JavaScript-кода

Прежде чем приступить к написанию программ, вам надо узнать некоторые правила оформления кода на JavaScript.

- ✓ **JavaScript чувствителен к регистру символов.** Это напоминание будет неоднократно встречаться вам на протяжении всей книги, поскольку многие новички часто забывают об этом. В JavaScript слова "Baby" и "baby" — совершенно разные.
- ✓ **В JavaScript количество пробельных символов в коде не играет особой роли.** К числу пробельных символов относятся пробелы, а также символы табуляции и разрыва строки, т.е. любой символ, не имеющий визуального представления. При написании кода JavaScript не имеет значения, используете ли вы один пробел, два пробела или даже (в большинстве случаев) символ разрыва строки там, где требуется пробел. JavaScript игнорирует пробельные символы. Единственным исключением являются случаи, когда вы записываете текст, который должен быть выведен на экран. В подобных случаях пробелы сказываются на конечном результате. Наилучший подход заключается в том, чтобы использовать пробелы для повышения удобочитаемости кода, придерживаясь при этом определенной системы.

- ✓ **Избегайте использования зарезервированных слов.** В JavaScript есть список слов, имеющих специальное значение в этом языке. Список этих слов приведен в главе 3. А на данном этапе вам надо просто знать, что такие, например, слова, как `function`, `while`, `break` и `with`, имеют специальное значение.

  В JavaScript повсеместно используется символ **точка с запятой** (`;`). Код JavaScript состоит из отдельных предложений, или инструкций. Инструкции — фундаментальные строительные блоки программ на JavaScript, которые напоминают предложения в обычном языке, являющиеся строительными блоками абзацев. В JavaScript инструкции заканчиваются точкой с запятой.

Если вы не закончите инструкцию точкой с запятой, то JavaScript сделает это за вас. Однако это может приводить к неожиданным результатам, поэтому обозначение конца инструкций точкой с запятой считается наилучшей практикой.

Выполнение JavaScript в окне браузера

Несмотря на то что JavaScript может встретиться вам в самых различных средах, чаще всего программы на JavaScript выполняются в браузерах. Управление вводом и выводом, манипулирование веб-страницами, обработка стандартных событий браузера, таких как щелчки и прокрутка, управление различными возможностями браузеров — вот для чего был создан JavaScript.

Существуют три способа выполнения JavaScript в браузере, каждый из которых рассматривается в последующих разделах:

- ✓ поместить код непосредственно в атрибут события HTML-элемента;
- ✓ поместить код между открывающим и закрывающим тегами `script`;
- ✓ поместить код в отдельный документ, который включается в документ HTML.

Вы будете не раз использовать различные комбинации всех этих трех подходов на одной и той же веб-странице. Однако знание того, когда именно следует применять тот или иной подход, имеет огромное значение, и это знание приходит лишь с практикой.

Использование JavaScript в атрибутах событий HTML-элементов

В HTML предусмотрено несколько специальных атрибутов, предназначенных для запуска JavaScript при наступлении определенных событий в браузере или при выполнении пользователем определенных действий. Вот пример HTML-элемента `button` с атрибутом события, обеспечивающим реакцию на щелчки мышью:

```
<button id="bigButton" onclick="alert('Привет,
        мир!');">Щелкните здесь</button>
```

В данном случае, когда пользователь щелкает на кнопке, созданной этим HTML-элементом, на экране появляется всплывающее окно, в котором отображается текст "Привет, мир!".

В HTML насчитывается около 70 различных атрибутов событий. Наиболее часто используемые из них приведены в табл. 2.3.

Таблица 2.3. Часто используемые атрибуты событий элементов HTML

Атрибут	Условие запуска сценария
`onload`	Окончание загрузки страниц
`onfocus`	Получение элементом фокуса ввода (например, при активизации текстового поля)
`onblur`	Потеря элементом фокуса ввода (например, в результате выполнения пользователем щелчка в другом текстовом поле)
`onchange`	Изменение значения элемента
`onselect`	Выделение текста
`onsubmit`	Отправка формы
`onkeydown`	Нажатие клавиши
`onkeypress`	Нажатие и последующее отпускание клавиши
`onkeyup`	Отпускание клавиши
`onclick`	Щелчок мышью на элементе
`ondrag`	Перетаскивание элемента
`ondrop`	Вставка перетаскиваемого элемента
`onmouseover`	Перемещение указателя мыши над элементом

Вообще говоря, несмотря на простоту описанного способа, многие JavaScript-программисты считают этот подход не лучшим решением. Мы демонстрируем его применение в книге, поскольку атрибуты событий широко применяются и просты в изучении. Но уже сейчас вы должны знать, что для написания JavaScript-кода, реагирующего на события, существуют лучшие способы, чем использование атрибутов событий. Более подробно этот вопрос рассматривается в главе 11.

Использование JavaScript в элементе `script`

HTML-элемент `script` позволяет внедрить код JavaScript в HTML-документ. Элементы `script` часто помещают в элемент `head`, и ранее этот способ считался чуть ли не обязательным. Однако в наши дни элементы `script` используются как в элементе `head`, так и в теле веб-страниц.

Элемент `script` имеет очень простой формат.

```
<script>
    Сюда вставляется код JavaScript
</script>
```

С примером такого способа внедрения сценариев вы уже встречались в листинге 2.1. В листинге 2.2 представлен другой пример HTML-документа с тегом `script`, содержащим JavaScript-код. Однако в данном случае элемент `script` находится в конце тела документа (элемента `body`).

Листинг 2.2. Внедрение JavaScript-кода в элемент `script`

```
<!DOCTYPE html>
<html>
<head>
  <title>Привет, HTML!</title>
</head>
<body>
  <h1>Посчитаем до 10 вместе с JavaScript!</h1>
  <p id="theCount"></p>
  <script>
    var count = 0;
    while (count < 10) {
      count++;
      document.getElementById("theCount").innerHTML +=
            count + "<br>";
    }
  </script>
</body>
</html>
```

Если вы создадите новый файл в Sublime Text, введете в него содержимое листинга 2.2, а затем откроете его в браузере, то увидите, что этот листинг делает то же самое, что и листинг 2.1.

Зависимость момента запуска сценария JavaScript от расположения элемента `script`

Как правило, выполнение сценариев браузерами происходит по мере их загрузки. Браузер всегда читает веб-страницу сверху вниз, как это делаете и вы, читая обычную текстовую страницу. Но иногда желательно, чтобы выполнение сценария начиналось лишь после того, как браузер загрузит полностью всю страницу. В листинге 2.1 это достигалось за счет использования атрибута события `onload` в элементе `body`. Другой общепринятый способ, позволяющий задержать выполнение сценария, состоит в том, чтобы поместить подлежащий выполнению код в самом конце документа, как это сделано в листинге 2.2.

Ограничения JavaScript-кода, находящегося в элементах `script`

Несмотря на то что внедрение сценариев JavaScript в элементы `script` более предпочтительно по сравнению со встраиванием в атрибуты событий, у такого подхода имеются серьезные ограничения.

Наибольшим из них является то, что сценарии, внедренные таким способом, может использовать лишь та веб-страница, в которую они помещены. Иными словами, если вы помещаете свой сценарий в элемент `script`, то должны скопировать этот элемент

и вставить его во все веб-страницы, в которых он используется. Нетрудно понять, что сопровождение сайтов, содержащих сотни таких страниц, превращается в сплошной кошмар.

Когда целесообразно располагать JavaScript-код в элементах `script`

Этот метод внедрения JavaScript-кода имеет свои области применения. В случае небольших сценариев, которые просто вызывают другие сценарии JavaScript и изменяются лишь изредка, данный метод вполне приемлем и даже может привести к сокращению времени загрузки и отображения веб-страниц, поскольку количество запросов к серверу при этом уменьшается.

Одностраничные приложения, которые (как следует из самого их названия) состоят из единственной HTML-страницы, также великолепно подходят для использования данного типа внедрения сценариев, поскольку вносить изменения, если это потребуется, надо будет только в одном месте.

Однако, как правило, следует всегда использовать любую возможность минимизировать объем JavaScript-кода, внедряемого непосредственно в HTML-документ. Это упрощает сопровождение сайта и позволяет лучше структурировать код.

Включение внешних JavaScript-файлов

Третий и наиболее популярный способ включения кода JavaScript в HTML-документы основан на использовании атрибута `src` элемента `script`.

Элемент `script` с атрибутом `src` работает точно так же, как и элемент `script`, в котором JavaScript-код располагается между тегами, за исключением того, что в случае использования атрибута `src` код JavaScript загружается в HTML-документ из отдельного файла. Вот пример элемента `script` с атрибутом `src`:

```
<script src="myScript.js"></script>
```

В данном случае должен существовать отдельный файл `myScript.js`, располагающийся в той же папке, что и HTML-документ. Использование внешних файлов JavaScript предоставляет следующие преимущества:

- ✓ улучшается удобочитаемость HTML-файлов;
- ✓ упрощается сопровождение кода, поскольку вносить изменения или исправлять ошибки приходится только в одном месте.

Создание `.js`-файла

Создание внешнего JavaScript-файла напоминает создание HTML-файла и вообще файла любого другого типа. Чтобы заменить внедренный JavaScript-код в листинге 2.1 внешним JavaScript-файлом, выполните следующие действия.

1. **Выберите в редакторе Sublime Text пункты меню File⇨New File (Файл⇨ Создать).**
2. **Скопируйте все, что находится между тегами `<script>` и `</script>` в файле `MyFirstProgram.html`, и вставьте его во вновь созданный `.js`-файл.**

Обратите внимание на то, что во внешние файлы копируется только JavaScript-код без элементов `<script>`.

3. **Сохраните новый файл с именем `countToTen.js` в той папке, в которой уже находится файл `MyFirstProgram.html`.**
4. **Измените элемент `<script>` в файле `MyFirstProgram.html`, добавив в него атрибут `src`, как показано ниже.**

```
<script src="countToTen.js"></script
```

Теперь файл `MyFirstProgram.html` должен содержать следующий текст.

```
<!DOCTYPE html>
<html>
<head>
  <title>Привет, HTML!</title>
  <script src="countToTen.js"></script>
</head>
<body onload="countToTen();">
  <h1>Посчитаем до 10 вместе с JavaScript!</h1>
  <p id="theCount"></p>
</body>
</html>
```

Содержимое нового файла `countToTen.js` должно быть таким.

```
function countToTen(){
  var count = 0;
  while (count < 10) {
    count++;
    document.getElementById("theCount").innerHTML +=
            count + "<br>";
  }
}
```

После того как вы сохраните оба файла, они должны появиться в папке вашего проекта, отображаемой в боковой панели редактора Sublime Text (рис. 2.8).

Организация `.js`-файлов проекта

Некоторые JavaScript-файлы могут достигать очень больших размеров. Во многих случаях целесообразно разбивать их на меньшие файлы, организованные по типу содержащихся в них функций. Например, один файл может содержать сценарии, связанные со входом пользователя в вашу программу, а второй — сценарии, связанные с ведением блога.

Однако для небольших программ обычно достаточно иметь всего лишь один файл, и, как правило, такому файлу присваивают какое-либо типовое имя, например `app.js`, `main.js` или `scripts.js`.

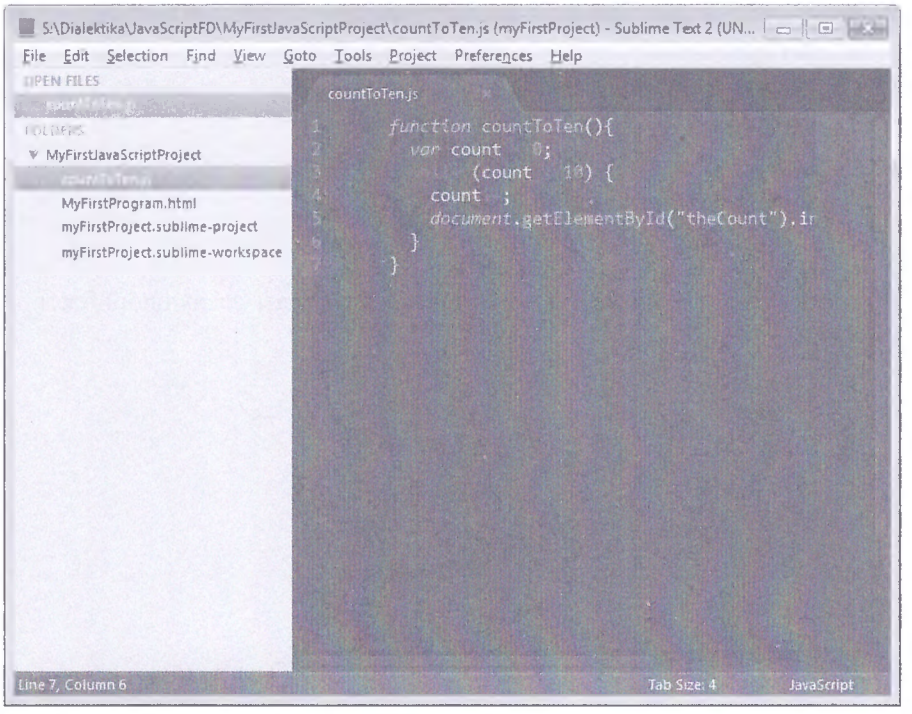


Рис. 2.8. Просмотр файлов, находящихся в папке проекта, в редакторе Sublime Text

JavaScript-файлы не обязательно должны находиться в той же папке, что и HTML-файл, который их включает. Более того, мы рекомендуем вам создавать новую папку, предназначенную специально для хранения внешних JavaScript-файлов. Большинство людей, которых мы знаем, присваивают таким папкам имена наподобие `js`.

Чтобы создать папку `js` в вашем проекте Sublime Text и переместить в нее `.js`-файл, выполните следующие действия.

1. **Щелкните правой кнопкой мыши на имени проекта в боковой панели Sublime Text.**

   Откроется подменю.

2. **Выберите в подменю пункт New Folder (Создать папку).**

   В нижней части окна Sublime Text появится поле `Folder Name` (Имя папки).

3. **Введите имя папки `js` в текстовом поле и нажмите клавишу <Enter>.**

   В боковой панели отобразится новая папка `js`.

4. **Откройте файл `countToTen.js`, выберите пункты меню File⇨Save As и сохраните файл в новой папке `js`.**

5. **Щелкните правой кнопкой мыши на версии файла `countToTen.js`, хранящейся вне папки `js`, и выберите в открывшемся подменю пункт Delete File (Удалить файл).**

6. **Откройте файл `MyFirstProgram.html` и измените элемент `<script>` таким образом, чтобы он отражал новое расположение `.js`-файла.**

```
<script src="js/countToTen.js"></script
```

Когда вы откроете файл `MyFirstProgram.html` в браузере (или просто обновите страницу), страница должна выглядеть точно так же, как и до перемещения JavaScript-файла в его собственную папку.

Использование консоли разработчика JavaScript

Иногда полезно иметь возможность запускать команды JavaScript без создания веб-страницы и последующего включения в нее кода сценариев или блоков `<script>`. В подобных ситуациях можно воспользоваться консолью JavaScript браузера Chrome (рис. 2.9).

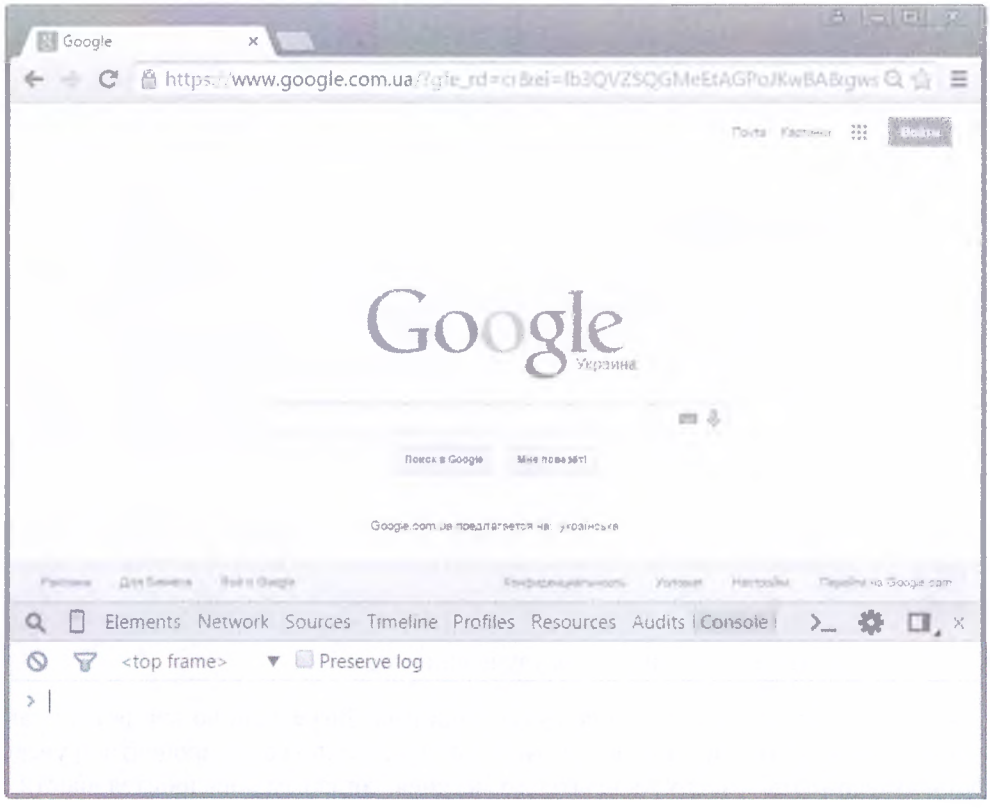


Рис. 2.9. Консоль JavaScript браузера Chrome

Чтобы получить доступ к консоли JavaScript, откройте меню Chrome, расположенное в правом верхнем углу окна браузера. Ему соответствует пиктограмма в виде трех

горизонтальных линий. Щелкните на ней и выберите в раскрывшемся меню пункт Дополнительные инструменты, а в следующем меню — пункт Консоль JavaScript.

Ах да! Для открытия окна консоли JavaScript существует гораздо более быстрый способ, требующий простого нажатия комбинации клавиш <Alt+Command+J> (Mac) или <Ctrl+Shift+J> (Windows).

Консоль JavaScript является, пожалуй, наилучшим другом разработчика на JavaScript. Она не только позволяет быстро и просто тестировать и выполнять JavaScript-код, но и сообщает, в каких местах кода содержатся ошибки, и предлагает средства, облегчающие обнаружение и разрешение проблем, связанных с работой кода.

Сразу же после открытия консоли можно начать ввод команд, которые будут выполняться после нажатия клавиши <Enter>. Чтобы попрактиковаться в этом, откройте консоль JavaScript и последовательно вводите приведенные ниже команды, нажимая клавишу <Enter> после ввода каждой из них.

```
1080/33
40 + 2
40 * 34
100%3
34++
34--
```

Комментирование кода

Когда вы изучите большее количество команд JavaScript и приступите к написанию более крупных программ, вы поймете, насколько полезными могут быть небольшие памятные записки, в которых вы фиксируете свои мысли относительно тех или иных аспектов программы или кратко описываете назначение отдельных фрагментов кода. На языке программистов эти записки, предназначенные для вас самих (а также для других людей, которые могут работать с вашим кодом), называются *комментариями*, а сам процесс их написания — *комментированием кода*.

Движок JavaScript полностью игнорирует комментарии, поскольку они предназначены исключительно для чтения людьми. Вы можете использовать их для записи необходимых пояснений или уточнений, изложения логики своих рассуждений, а также для фиксации того, что вы собираетесь сделать в дальнейшем для улучшения кода.

Комментарии в коде никогда не будут лишними. Даже если во время написания кода вы считаете, что он и так достаточно понятен, можно со стопроцентной уверенностью утверждать, что спустя несколько месяцев, когда вам понадобится внести изменения в код, восстановить в памяти логику его работы вам будет не так-то легко.

В JavaScript существуют два типа комментариев:

- ✓ однострочные;
- ✓ многострочные.

Однострочные комментарии

Однострочные комментарии начинаются двумя символами косой черты (//). Все, что находится после них до конца строки, игнорируется *парсером* (синтаксическим анализатором) JavaScript.

Однострочные комментарии не обязательно должны располагаться в начале строки. Они довольно часто встречаются в той же строке, что и комментируемый код, поясняя его назначение. Например:

```
pizzas = pizza + 1; // добавить еще одну пиццу
```

Многострочные комментарии

Многострочные комментарии начинаются символами /* и сообщают парсеру JavaScript о том, что весь последующий текст вплоть до пары символов */ следует игнорировать. Многострочные комментарии полезны для более полного документирования кода.

```
/* Функция countToTen выполняет следующие действия:
  * Инициализирует переменную count нулевым значением
  * Запускает цикл с проверкой того, что текущее значение
        count меньше 10
  * Увеличивает значение count на 1
  * Добавляет текущее значение count вместе со следующим за ним
        символом разрыва строки в абзац с id='theCount'
  * Запускает следующую итерацию цикла
*/
```

Использование комментариев для предотвращения выполнения кода

Комментарии полезны не только для документирования программ, но и для изоляции фрагментов кода в процессе отладки с целью локализации ошибок. Допустим, нам захотелось узнать, что произойдет, если удалить из функции countToTen строку, обеспечивающую приращение значения переменной count. Для этого достаточно ввести в начале строки символы однострочного комментария или, как говорят, "закомментировать" эту строку.

```
function countToTen(){
var count = 0;
while (count < 10) {
// count++;
document.getElementById("theCount").innerHTML +=
count + "<br>";
}
}
```

Если вы запустите эту программу, то строка count++ уже не будет выполняться и программа будет бесконечно (или пока вы не закроете окно браузера) выводить нули.

Конструкция, которую мы только что использовали, называется *бесконечным циклом*. Выполнение видоизмененной версии программы не причинит вреда вашему компьютеру, но, вероятно, заставит процессор вашего компьютера работать на бешеной скорости до тех пор, пока вы не закроете окно браузера, в котором выполняется программа.

Глава 3

Работа с переменными

В этой главе...

- Создание и использование переменных
- Область видимости переменных
- Типы данных JavaScript
- Правила именования переменных
- Использование встроенных функций для работы с переменными

"Красота изменчива, уродство постоянно".
Дуглас Хортон

В этой главе вы узнаете о том, как создавать переменные, присваивать им значения, использовать функции, позволяющие определить тип переменных, преобразовывать данные из одного типа в другой и манипулировать данными.

Понятие переменной

Переменные — это символические имена в программе. Точно так же, как *x* может представлять некоторое пока еще неизвестное значение в алгебре или точку на пиратской карте, обозначающую место, где зарыты сокровища, переменные в программировании используются для представления всевозможных величин.

Можете считать переменные контейнерами, предназначенными для хранения данных. Этим контейнерам можно присваивать имена и впоследствии использовать их для извлечения и изменения хранимых данных.

В отсутствие переменных любая компьютерная программа может быть предназначена только для чего-то одного. Например, рассмотрим следующий пример, включающий всего лишь одну строку, в которой не используются переменные:

```
alert(3 + 7);
```

Эта программа суммирует числа 3 и 7 и выводит результат во всплывающем окне браузера.

Однако от такой программы нет почти никакого толка (если только вам не приходится регулярно вспоминать, чему равна сумма чисел 3 и 7). Используя переменные, можно создать универсальную программу, которая будет суммировать любые два числа и выводить результат, как в следующем примере.

```
var firstNumber = 3;
var secondNumber = 7;
var total = Number(firstNumber) + Number(secondNumber);
alert (total);
```

Сделав еще один шаг, можно расширить программу таким образом, чтобы пользователь мог задавать числа, сумму которых требуется найти.

```
var firstNumber = prompt("Введите первое число");
var secondNumber = prompt("Введите второе число");
var total = Number(firstNumber) + Number(secondNumber);
alert (total);
```

Проверьте сами, как работает эта программа. (О том, как работать с редактором кода, см. в главе 2.) Для этого выполните следующие действия.

1. **Откройте редактор кода и создайте базовый шаблон HTML-документа.**
2. **Между тегами `<body>` и `</body>` вставьте открывающий и закрывающий теги сценария (`<script>` и `</script>`).**
3. **Введите между тегами сценария приведенный выше JavaScript-код.**

   Теперь ваш документ должен приобрести следующий вид.

   ```
   <html>
   <head></head>
   <body>
     <script>
     var firstNumber = prompt("Введите первое число");
     var secondNumber = prompt("Введите второе число");
     var total = Number(firstNumber) + Number(secondNumber);
     alert (total);
     </script>
   </body>
   </html>
   ```

4. **Сохраните свой новый HTML-документ в файле с именем `addTwo.html`.**
5. **Откройте HTML-документ в браузере.**

   Программа предложит вам ввести первое число (рис. 3.1).

6. **Введите первое число.**

   Программа предложит вам ввести второе число.

7. **Введите второе число.**

   После того как вы предоставите второе число, результат сложения двух введенных чисел отобразится на экране.

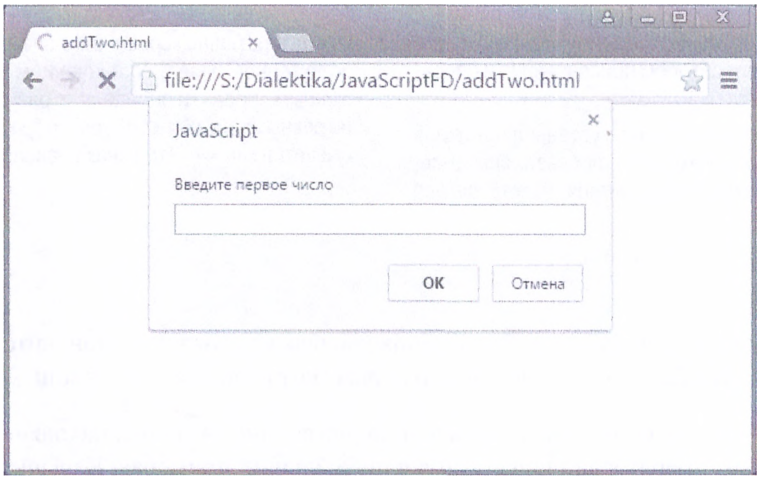


Рис. 3.1. Универсальная программа, выполняющая сложение двух чисел, предоставляемых пользователем

Объявление переменных

Объявление переменной — это технический термин, который используется для описания процесса первоначального создания переменной в программе. Вам также может встретиться эквивалентный ему термин *инициализация*. Создание переменной, объявление переменной, инициализация переменной — все эти выражения имеют один и тот же смысл.

В JavaScript переменные можно создавать одним из двух способов.

- ✓ **С использованием ключевого слова `var`.**

  ```
  var myName;
  ```

  Переменной, объявляемой с помощью ключевого слова `var`, можно присвоить начальное значение в момент ее создания (это называется *инициализацией переменной*).

  ```
  var myName = "Chris"
  ```

- ✓ **Без использования ключевого слова `var`.**

  ```
  myName = "Chris";
  ```

Когда "равно" не означает равенство

Инструкции наподобие этой, которые содержат знак "=", обычно читаются так :"Значение переменной `var` *равно* `"Chris"`". Однако в программировании такая буквальная интерпретация этого выражения не совсем корректна.

Рассмотрим инструкцию `var myName = "Chris";` более подробно.

Несмотря на то что символ, помещенный между именем переменной (`myName`) и ее значением (`"Chris"`), выглядит в точности как знак равенства и даже вводится с использованием той же клавиши, в программировании он называется не знаком равенства, а *оператором присваивания*.

Очень важно, чтобы вы понимали смысл различия между оператором присваивания и оператором "равно".

- ✓ Оператор присваивания устанавливает для переменной, находящейся слева, значение, равное значению выражения, находящегося справа:

```
var myName = "Chris";
```

- ✓ Оператор "равно" сравнивает значение, расположенное слева, со значением, расположенным справа, и определяет, равны они или не равны. В JavaScript оператор "равно" записывается как === (три знака равенства).

Переменная, созданная без использования ключевого слова `var`, становится *глобальной* переменной. (Глобальные переменные рассматриваются в следующем разделе.)

Обратите внимание на кавычки, окружающие значение, находящееся справа от оператора присваивания в предыдущих примерах. Кавычки указывают на то, что это значение должно интерпретироваться как текст, а не как число, ключевое слово JavaScript или другая переменная. Более подробно о том, когда и при каких обстоятельствах следует использовать кавычки, говорится в разделе, посвященном типам данных.

Глобальные и локальные области видимости

Как и где объявлять переменную, зависит от того, как именно и в каких местах программы эта переменная используется. Ключевую роль здесь играет понятие *области видимости переменной*. В JavaScript существует два типа области видимости.

- ✓ *Глобальные переменные* могут использоваться в любом месте программы.
- ✓ *Локальные переменные* (переменные функций) — это переменные, создаваемые в защищенных программах, которые располагаются в основной программе и называются *функциями*.

Трагическая история о недостающем ключевом слове `var`

В создании переменных без использования ключевого слова `var` нет никакой реальной необходимости. Более того, во многих случаях это чревато проблемами. Если вы опускаете ключевое слово `var`, то это выглядит так, будто вы о нем попросту забыли, поэтому никогда так не поступайте!

Приведем пример того, какого рода проблемы могут возникать из-за пропущенного ключевого слова `var`. В этом примере также будет продемонстрировано использование более сложных программных конструкций — функций, о которых будет более подробно рассказано в главе 7. Если говорить коротко, то функции позволяют разбивать программы на более мелкие части.

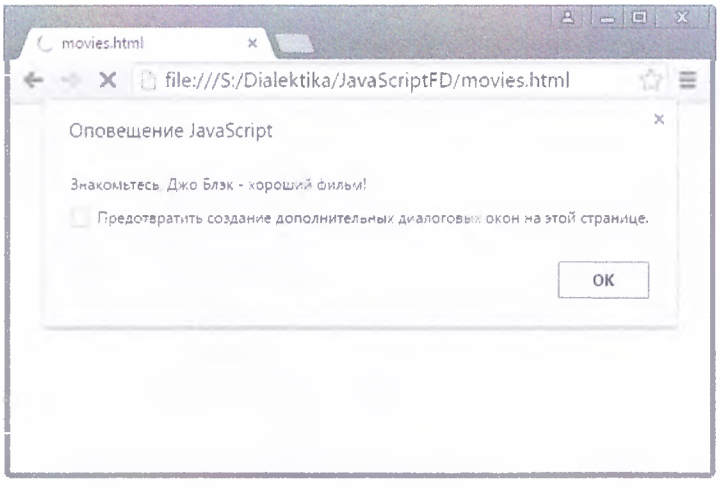
В этом примере программист хочет создать глобальную переменную `movie` и другую переменную, которая называется так же, но действительна только в функции `showBadMovie()`. В этом нет ничего предосудительного, и при обычных обстоятельствах переменная `movie`, находящаяся в функции, никоим образом не будет взаимодействовать с одноименной глобальной переменной.

В то же время, если вы забудете указать ключевое слово `var` при объявлении переменной в функции, то начнут происходить неприятные вещи.

```
var movie = "Крестный отец";

function showGoodMovie () {
  alert (movie + " - хороший фильм!");
}

function showBadMovie () {
  movie = "Знакомьтесь, Джо Блэк";
  alert (movie + " - плохой фильм!");
}
```

Обратите внимание на то, что в функции `showBadMovie()` перед переменной `movie` отсутствует ключевое слово `var`. Поэтому JavaScript предполагает, что вы хотите переопределить значение глобальной переменной, а не создать новую переменную. Результат этого является поистине катастрофическим!

```
showGoodMovie(); // выводится сообщение:
                 // "Крестный отец - хороший фильм!"

showBadMovie();  // выводится сообщение:
                 // "Знакомьтесь, Джо Блэк - плохой фильм!"

/* Ой! Фильм "Знакомьтесь, Джо Блэк" был представлен
   как плохой, но теперь это название присвоено в виде значения
   глобальной переменной movie. */
showGoodMovie(); // выводится сообщение:
                 // "Знакомьтесь, Джо Блэк - хороший фильм!"
```



Вообще говоря, лучше использовать локальные переменные, а не глобальные, поскольку ограничение области видимости снижает вероятность того, что значение данной переменной будет случайно заменено значением другой переменной с тем же именем.

Использование глобальных переменных может приводить к возникновению проблем в работе программы, которые вам будет трудно обнаружить и исправить. Мы рекомендуем полностью отказаться от практики объявления переменных без использования ключевого слова `var`. Если у вас возникает необходимость в создании глобальной переменной, ее также лучше всего объявлять с помощью ключевого слова `var`, и мы советуем всегда придерживаться этого правила.

Именование переменных

Имена переменных можгут начинаться с одного из следующих символов:

- ✓ буква в верхнем или нижнем регистре;
- ✓ символ подчеркивания (_);
- ✓ знак доллара ($).

Несмотря на то что использование символа подчеркивания и знака доллара в начале имен переменных считается допустимым, все же лучше, чтобы имя переменной начиналось с буквы. Непривычные символы часто затрудняют чтение кода и делают его менее понятным, особенно для новичков в JavaScript.

Вслед за первым символом имени переменной могут стоять любые буквы и цифры, причем имя может иметь любую длину. Имена переменных в JavaScript не могут содержать пробелы, символы операторов и знаки препинания (отличные от символа подчеркивания).

Не забывайте о том, что язык JavaScript чувствителен к регистру. Переменные с именами `myname`, `Myname` и `myName` — это разные переменные.

Фактически имена переменных являются идентификаторами. Лучше всего давать переменным точные и осмысленные названия. Иногда соглашения, касающиеся именования переменных, могут приводить к длинным именам, но обычно длинное имя, которое точно отражает назначение переменной, более полезно, чем ничего не значащее короткое имя.

Разумеется, чтобы не усложнять себе жизнь, не следует использовать слишком длинные имена переменных. Если для точной передачи смысла переменной потребуется имя длиной 20 символов, это нормально. Но если вы создаете переменные с такими именами, как `nameOfPersonWhoJustFilledOutTheFormOnMyWebsite`, то, вероятно, стоит использовать более короткие имена наподобие `personName`, чтобы упростить жизнь себе, а также другим людям, которые, возможно, будут читать ваш код.

Рекомендации по созданию хороших имен переменных

Несмотря на то что JavaScript предоставляет большую свободу при назначении имен переменным, будет лучше, если вы выработаете для себя определенные правила в отношении этого еще до того, как приступите к программированию. Например, должны ли имена переменных начинаться со строчной или с прописной буквы? Разделять ли отдельные слова в составных именах символом подчеркивания или писать их слитно, но использовать "верблюжий стиль", при котором каждое слово начинается с прописной буквы? По мере усложнения кода важность правильного выбора стратегии присвоения имен переменных становится очевидной.

К счастью, в выработке собственного стиля оформления программ вы не остаетесь одиноки. Существуют проверенные опытом правила именования переменнных, которых стараются придерживаться программисты на JavaScript. Вот эти правила.

- ✓ Не используйте слишком короткие имена! Имена, состоящие из одной буквы, равно как и имена, не несущие никакого смысла, — далеко не лучший выбор для использования в качестве имен переменных.
- ✓ Используйте составные имена, чтобы как можно точнее описать назначение переменных.
- ✓ В составных именах всегда располагайте прилагательные слева, например `var greenPython;`.

Выберите определенный стиль для составных имен и строго придерживайтесь его. Возможны два способа объединения слов для создания имени: использование "верблюжьего" стиля или символов подчеркивания. JavaScript — гибкий язык, и вы вправе использовать любой из этих методов, но обычно применяют первый из них.

Некоторые слова нельзя использовать в качестве имен переменных. Ниже приведен перечень зарезервированных слов, которые не могут выступать в качестве названий переменных, функций, методов, меток циклов и объектов.

abstract	else	instanceof	switch
boolean	enum	int	synchronized
break	export	interface	this
byte	extends	long	throw
case	false	native	throws
catch	final	new	transient
char	finally	null	true
class	float	package	try
const	for	private	typeof
continue	function	protected	var
debugger	goto	public	void
default	if	return	volatile
delete	implements	short	while
do	import	static	with
double	in	super	

Создание констант с помощью ключевого слова const

Иногда может возникать необходимость в переменных, значения которых не могут быть изменены. Подобные переменные, которые называют *константами*, объявляются с использованием ключевого слова `const`.

```
const heightOfTheEmpireStateBuilding = 1454;
const speedOfLight = 299792458;
const numberOfProblems = 99;
const meanNumberofBooksReadIn2014 = 12;
```

Константы подчиняются тем же правилам, что и другие переменные, но коль скоро вы создали переменную как константу, ее значение не может быть изменено на протяжении всей ее жизни (которая длится, пока выполняется сценарий).

Работа с типами данных

Тип данных переменной указывает на то, данные какого вида могут храниться в переменной и какие операции могут выполняться с использованием ее значения. Например, число 10, используемое в текстовом предложении, отличается от числа 10, используемого в операции сравнения. Типы данных — это способ, используемый в JavaScript для проведения различий между значениями, используемыми как текст, и значениями, которые должны обрабатываться как математические выражения.

Если вы проанализируете все типы данных, с которыми вам приходится сталкиваться в повседневной жизни, — это и круговые диаграммы, и рецепты, и короткие рассказы, и журнальные статьи, и многое другое, — то поймете, насколько все усложняется, когда речь идет о данных. Великодушные создатели JavaScript решили сделать все для того, чтобы облегчить вам жизнь. Они предусмотрели всего лишь пять основных типов данных.

Более того, JavaScript относится к категории так называемых *слабо типизированных языков*. Это означает, что от вас не требуется сообщать JavaScript или даже знать это самому, будет ли создаваемая вами переменная содержать слово, абзац, число или какой-либо другой тип данных.

Слабая типизация означает, что JavaScript не требует, чтобы вы сообщали ему о том, что есть число, а что есть слово. Он с поблажкой относится к этому и незаметно для вас выполняет "за кулисами" всю работу по определению типов данных, хранящихся в переменных.

JavaScript распознает пять основных или, как говорят, примитивных (элементарных) типов данных. В языке программирования C++ насчитывается по крайней мере 12 различных типов данных.

Числовой тип данных

В JavaScript числа хранятся в виде 64-разрядных значений с плавающей точкой. В переводе на обычный язык это означает, что их значения могут меняться в пределах от $\pm 5 \times 10^{-324}$ (т.е. 5 с предшествующими 323 нулями после десятичной точки) до

$\pm 1.7976931348623157 \times 10^{308}$ (переместите десятичную точку вправо на 308 позиций, и вы увидите это гигантское число). Любое число может быть записано как с десятичной точкой, так и без нее. В отличие от большинства других языков программирования, в JavaScript отсутствуют отдельные типы данных для целых чисел (положительных и отрицательных чисел без дробной части) и чисел с плавающей точкой (десятичных).

Насколько велико самое большое число, которое можно использовать в JavaScript? Вот как оно выглядит, будучи записанным не в научном, а в обычном формате.

```
179769313486231570000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000
```

Объявление числовой переменной состоит из следующих элементов:

- ✓ ключевое слово `var`;
- ✓ имя, назначенное переменной;
- ✓ оператор присваивания;
- ✓ число (или же выражение, которое сводится к числу);
- ✓ точка с запятой.

Вот некоторые примеры корректного объявления числовых переменных.

```
var numberOfDucks = 4;
var populationOfSpain = 47200000;
var howManyTacos = 8;
```

Функция `Number()`

В JavaScript предусмотрена встроенная функция `Number()`, предназначенная для преобразования чисел, которые могут участвовать в вычислениях. Чтобы использовать эту функцию, просто укажите значение (или содержащую его переменную), которое вы хотите преобразовать в число, в круглых скобках после имени `Number`.

Функция `Number()` имеет четыре типа выходных значений, которые перечислены ниже.

- ✓ Числа, отформатированные как текстовые строки, преобразуются в числа, которые могут участвовать в вычислениях:
  ```
  Number("42") // возвращает число 42
  ```
- ✓ В случае строк, которые не могут быть преобразованы в числа, возвращается значение NaN:
  ```
  Number("eggs") // возвращает значение NaN
  ```
- ✓ В случае булевого значения `true` возвращается число 1:
  ```
  Number(true) // возвращает число 1
  ```
- ✓ В случае булевого значения `false` возвращается число 0:
  ```
  Number(false) // возвращает число 0
  ```

Функция `parseInt()`

Для JavaScript все числа являются фактически числами с плавающей точкой. Однако с помощью функции `parseInt()` вы сможете взять только целую часть числа (без дробной части), отбросив все, что находится после десятичной точки.

```
parseInt(100.33); // возвращает 100
```

Функция `parseFloat()`

Функция `parseFloat()` позволяет явно указать JavaScript, что данное число должно трактоваться как число с плавающей точкой. Кроме того, ее можно использовать для преобразования строки в число, например:

```
parseFloat(100.33); //возвращает 100.33
parseFloat("10"); //возвращает 10
```

Примеры

Теперь вы уже достаточно осведомлены для того, чтобы самостоятельно поэкспериментировать с числами и числовыми функциями. Попробуйте ввести приведенные ниже выражения в консоли JavaScript браузера Chrome и посмотрите, к каким результатам они приводят.

Для открытия консоли JavaScript можно использовать комбинацию клавиш <Command+Option+J> (Mac) или <Ctrl+Shift+J> (Windows).

```
1 + 1
3 * 3
parseFloat("839");
parseInt("33.333333");
12 + "12"
"12" + 12
"12" * 2
```

Числовые переменные должны объявляться без использования кавычек: `"10"` — это не то же самое, что `10`. Первое из этих значений является строкой (строки рассматриваются в следующем разделе), и если вы случайно объявите переменную, инициализировав ее числом, заключенным в кавычки, то можете получить неожиданные результаты.

Если вы были внимательны, то могли заметить несколько необычное поведение переменных в предыдущих примерах. Например, если к `"12"` (строка) прибавить `12` (число), то результатом будет `"1212"` (строка). Но если вы умножите `"12"` (строка) на `2` (число), то получите `24` (число). Это как раз тот случай, когда JavaScript анализирует текст программы.

В первом примере (сложение), поскольку одно из слагаемых является строкой, JavaScript предполагает, что оба слагаемых рассматриваются вами как строки. Поэтому число преобразуется в строку, а символ + интерпретируется как *оператор конкатенации*.

Во втором примере (умножение) одним из значений, участвующих в операции, является число, а операция умножения двух строк не имеет смысла. Поэтому JavaScript преобразует строку в число, после чего выполняет операцию умножения. Но что случится, если вы попытаетесь перемножить две строки?

```
"sassafras" * "orange"
```

Результатом будет значение NaN (Not a Number — не число). Поскольку строки `"sassafras"` и `"orange"` не преобразуются в числа, JavaScript проводить бессмысленное вычисление.

Строковый тип данных

Строки могут состоять из следующих символов:

- ✓ буквы;
- ✓ цифры;
- ✓ знаки пунктуации (например, запятая или точка);
- ✓ специальные символы, которые могут быть записаны с помощью символа с предшествующим ему символом обратной косой черты.

Некоторые символы, например кавычки, имеют в JavaScript особый смысл или требуют использования особых сочетаний символов (например, символов табуляции или разрыва строки) для представления в строках. Мы называем их *специальными символами*. Специальные символы, которые можно использовать в строках JavaScript, приведены в табл. 3.1.

Таблица 3.1. Специальные символы JavaScript

Код	Описание
`\'`	Одинарная кавычка
`\"`	Двойная кавычка
`\\`	Обратная косая черта
`\n`	Перевод строки
`\r`	Возврат каретки
`\t`	Табуляция
`\b`	Возврат на один символ
`\f`	Подача формы

Строковая переменная создается путем заключения значения в одинарные или двойные кавычки:

```
var myString = "Привет, я строка.";
```

Не имеет никакого значения, какие кавычки вы для этого используете — одинарные или двойные, лишь бы открывающая и закрывающая кавычки были одного типа.

Окружив строку одинарными кавычками, можно беспрепятственно использовать двойные кавычки в составе самой строки. Подобным образом, заключив строку в двойные кавычки, вы без проблем сможете включать в нее одинарные кавычки.

Заключив строку в кавычки одного типа, вы не сможете использовать кавычки этого же типа в строке, поскольку парсер JavaScript, встретив первую из таких внутренних кавычек, решит, что вы имели в виду закончить ею строку, и сгенерирует ошибку.

Экранирование кавычек

Проблема включения кавычек в строку, окруженную кавычками того же типа, решается помещением символа обратной косой черты (\) перед внутренней кавычкой. Это называется *экранированием* кавычек.

Функции для работы со строками

В JavaScript имеется много полезных функций, предназначенных для обработки и преобразования строк.

Ниже перечислены наиболее часто используемые встроенные функции для работы со строками.

- ✓ `charAt()`. Возвращает символ, находящийся в указанной позиции в строке. Обратите внимание на то, что отсчет позиций начинается с 0.

```
var watzThisString = 'JavaScript - это здорово!';
console.log (watzThisString.charAt(3));
// возвращает "a"
```

- ✓ `concat()`. Присоединяет к строке одну или несколько строк и возвращает объединенную строку.

```
var watzThisString = 'JavaScript - это здорово!';
console.log (watzThisString.concat(' Нам нравится
        JavaScript!'));
// возвращает "JavaScript - это здорово! Нам нравится
JavaScript!"
```

- ✓ `indexOf()`. Выполняет поиск первого вхождения символа или подстроки в данной строке и возвращает номер соответствующей позиции.

```
var watzThisString = 'JavaScript - это здорово!';
console.log (watzThisString.indexOf('здорово'));
// возвращает 17
```

- ✓ `split()`. Разбивает строку на массив подстрок.

```
var watzThisString = 'JavaScript - это здорово!';
console.log (watzThisString.split('з'));
// возвращает ["JavaScript - это ", "дорово!"]
```

- ✓ `substr()`. Извлекает часть строки, начиная с указанной позиции, и возвращает указанное количество символов.

```
var watzThisString = 'JavaScript - это здорово!';
console.log (watzThisString.substr(2,5));
// возвращает "vaScr"
```

- ✓ `substring()`. Извлекает символы строки, заключенные между указанными позициями (символ, соответствующий концу диапазона, в результат не включается).

```
var watzThisString = 'JavaScript - это здорово!';
console.log (watzThisString.substring(2,5));
// возвращает "vaS"
```

- ✓ `toLowerCase()`. Переводит все символы строки в нижний регистр.

```
var watzThisString = 'JavaScript - это здорово!';
console.log (watzThisString.toLowerCase());
// возвращает "javascript - это здорово!"
```

- ✓ `toUpperCase()`. Переводит все символы строки в верхний регистр.

```
var watzThisString = 'JavaScript - это здорово!';
console.log (watzThisString.toUpperCase());
// возвращает "JAVASCRIPT - ЭТО ЗДОРОВО!"
```

Булев тип данных

Булевы переменные принимают одно из двух возможных значений: `true` или `false`.

Булевы переменные получили свое название по фамилии математика Джорджа Буля (1815–1864) — создателя алгебраической системы логики.

Булевы переменные часто используют для хранения результатов сравнения. С помощью функции `Boolean()` можно найти булев результат сравнения или преобразовать любое значение JavaScript в булево значение.

```
var isItGreater = Boolean (3 > 20);
alert (isItGreater); // returns false

var areTheySame = Boolean ("тигр" === "Тигр");
alert (areTheySame); // returns false
```

Результат преобразования некоторого значения JavaScript в булево значение с использованием функции `Boolean()` зависит от преобразуемого значения.

- ✓ В JavaScript следующие значения всегда дают при вычислении булево значение `false`:
  - `NaN`;
  - `undefined`;
  - `0` (нулевое численное значение);
  - `-0`;
  - `""` (пустая строка);
  - `false`;

- ✓ Все остальное, что не относится к перечисленным выше значениям, дает при вычислении булево значение `true`. Например:
  - `74;`
  - `"Eva";`
  - `"10";`
  - `"NaN".`

Цифровой символ `"0"` — это не то же самое, что число `0` (нуль). Если `0` при преобразовании в булево значение всегда дает `false`, то преобразование строки `"0"` всегда дает `true`.

Булевы значения в основном используются в условных выражениях. Приведенная ниже программа создает булеву переменную, а затем тестирует ее значение с использованием инструкции `if/else` (подробнее — в главе 6).

```
var b = true;
if (b == true) {
  alert ("Это true!");
  } else {
  alert ("Это false.");
  }
```

При записи булевых значений их не заключают в кавычки:

```
var myVar = true;
```

В то же время объявление `var myVar = "true"` создает строковую переменную.

Тип данных NaN

NaN — это сокращение от *Not a Number* (не число). Этот результат вы будете получать при попытке выполнить математические операции над строками или в тех случаях, когда вычисление приводит к ошибке или не может быть выполнено. Например, невозможно вычислить квадратный корень отрицательного числа. Результатом такой попытки будет NaN.

Чаще всего значение NaN возвращается при попытке выполнения математических операций над строками, которые не могут быть преобразованы в числа.

Тип данных `undefined`

Даже если вы создадите переменную JavaScript и не присвоите ей какое-либо значение, она всегда будет иметь значение по умолчанию. Таковым является значение `"undefined"`.

Глава 4

Массивы

В этой главе...

- Что такое массив
- Создание массивов
- Многомерные массивы
- Работа с элементами массива
- Использование функций и свойств массивов

"Я огромен. Во мне — множества".
Уолт Уитмен

Массивы — фундаментальная часть любого языка программирования. В этой главе вы узнаете о том, что собой представляют массивы, как их используют и что отличает массивы JavaScript от массивов в других языках программирования. Вы будете использовать массивы для создания и упорядочения списков, а также для добавления и удаления их элементов.

Создание списка

Предыдущие главы включали примеры работы с переменными, которые представляют отдельные единицы данных, например: `var myName = "Chris"`, `var firstNumber = "3"` и `var howManyTacos = 8`. В программировании (как и в жизни) часто бывает так, что родственные данные желательно хранить под одним именем. Например, рассмотрим следующие типы списков:

- ✓ список любимых артистов;
- ✓ список цитат, из которого каждый раз при запуске программы выбирается и отображается новая цитата;
- ✓ список лучших музыкальных альбомов года;
- ✓ список дней рождения членов вашей семьи и друзей;
- ✓ список покупок;
- ✓ список "что сделать";
- ✓ список новогодних пожеланий.

Используя переменные, способные хранить только одно значение (см. главу 3), вы должны будете создавать и по отдельности контролировать каждую из множества переменных, образующих список. Вот пример списка, созданного с использованием переменных, хранящих одиночные значения:

```
var artist1 = "Alphonse Mucha";
var artist2 = "Chiara Bautista";
var artist3 = "Claude Monet";
var artist4 = "Sandro Botticelli";
var artist5 = "Andy Warhol";
var artist6 = "Wassily Kadinski";
var artist7 = "Vincent Van Gough";
var artist8 = "Paul Klee";
var artist9 = "William Blake";
var artist10 = "Egon Schiele";
var artist11 = "Salvador Dali";
var artist12 = "Paul Cezanne";
var artist13 = "Diego Rivera";
var artist14 = "Pablo Picasso";
```

В краткосрочной перспективе такой подход мог бы сработать, но вы быстро столкнетесь с трудностями. Что если вам захочется отсортировать список в алфавитном порядке и надо будет соответствующим образом переместить имена в другие переменные? В этом случае вам, например, придется сначала извлечь из переменной `artist1` имя `Mucha` (и сохранить его во временной переменной), а затем сохранить в ней имя `Bautista`. Тогда место в переменной `artist2` освободится для имени `Blake`, но не стоит забывать о том, что имя `Mucha` по-прежнему находится во временном хранилище! Извлечение имени `Blake` из переменной `artist9` освобождает ее для того, чтобы ее можно было использовать в качестве временного хранилища для другого имени, и т.д. С увеличением размера списка эта задача еще более усложняется.

К счастью, JavaScript (как, впрочем, и любой другой из известных нам языков программирования) поддерживает создание переменных, способных содержать более одного значения. Эти переменные называются *массивами*. Массивы — это способ хранения группы родственных данных в единственной переменной. С помощью массивов можно создавать списки, содержащие смешанные строковые и числовые значения, булевы значения и объекты, функции и другие типы данных, даже другие массивы!

Основные сведения о массивах

Массив состоит из элементов. Элемент массива записывается в виде имени массива, за которым в квадратных скобках указывается индекс элемента в массиве. Каждое отдельное значение, входящее в массив, называется *элементом массива*. Для доступа к элементам массива используются числа — так называемые *индексы*. Ниже приведен пример, демонстрирующий доступ к элементам массива по индексу.

```
myArray[0] = "желтый шарик";
myArray[1] = "красный шарик";
myArray[2] = "синий шарик";
myArray[3] = "розовый шарик";
```

В этом примере элемент с индексом `0` имеет значение `"желтый шарик"`. Элемент с индексом `3` имеет значение `"розовый шарик"`. Как и в случае любой переменной, массиву можно присвоить любое имя, совместимое с правилами именования переменных JavaScript. Благодаря индексам можно использовать одно имя переменной для хранения почти неограниченного списка значений.

Исключительно для того, чтобы вы случайно не перестарались, сообщим, что количество элементов, которые могут храниться в массиве, в действительности ограничено, хотя и маловероятно, что вы превысите допустимый предел, который составляет 4 294 967 295 элементов.

Кроме правил именования (одинаковых для переменных любого типа; см. главу 3), при работе с массивами следует придерживаться еще некоторых правил:

- ✓ индексы отсчитываются от нуля;
- ✓ в массивах могут храниться данные любого типа.

Отсчет индексов в массивах ведется от нуля

У JavaScript нет пальцев, по которым можно было бы вести счет. А раз так, то нет нужды привязываться к нашей фанатичной человеческой привычке начинать считать с единицы. Индексом первого элемента массива в JavaScript всегда является `0` (рис. 4.1).

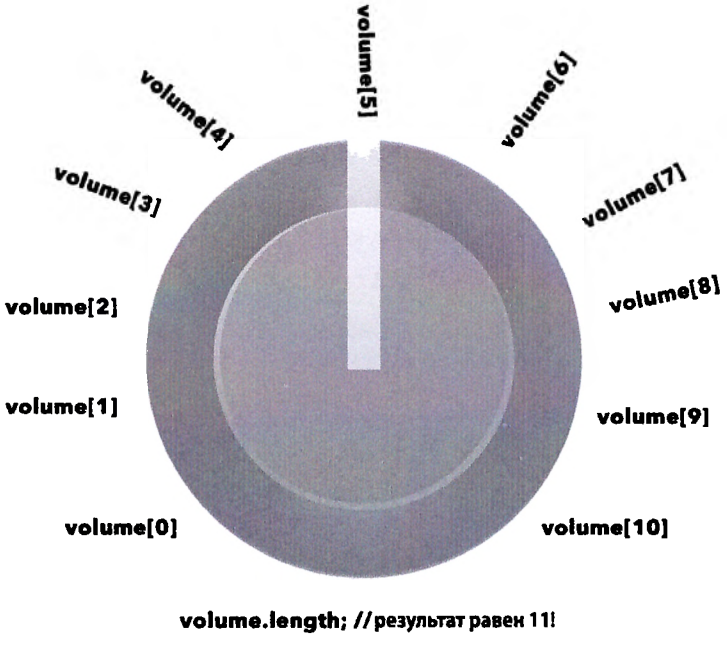


Рис. 4.1. При работе с массивами JavaScript ведет себя подобно регулятору громкости: отсчет ведется от нуля

Для вас это означает, что элемент `myArray[3]` — это на самом деле четвертый элемент массива.

Отсчет индексов от нуля часто приводит к путанице и является причиной ошибок, допускаемых начинающими программистами, но если к этому привыкнуть, то со временем все становится на свои места.

В массивах могут храниться данные любого типа

В любом элементе массива могут храниться данные любого типа (см. главу 3), в том числе и другие массивы. Кроме того, элементы массивов могут содержать функции и объекты JavaScript (подробнее об этом — в главах 7 и 8).

Имея возможность хранить в массиве данные любого типа, вы также можете хранить разные типы данных в разных элементах одного и того же массива (листинг 4.1).

Листинг 4.1. Хранение данных различного типа в одном массиве

```
item[0] = "apple";
item[1] = 4+8;
item[2] = 3;
item[3] = item[2] * item[1];
```

Создание массивов

JavaScript предлагает два способа создания массивов:

- ✓ ключевое слово `new`;
- ✓ литеральное определение массива.

Использование ключевого слова `new`

В способе, основанном на использовании ключевого слова `new`, массив создается и заполняется значениями с помощью конструкции `new Array()`:

```
var catNames = new Array("Larry", "Fuzzball", "Mr. Furly");
```

Возможно, вы уже сталкивались с этим методом в своей карьере программиста, и он вполне подходит для создания массивов.

Следует отметить, что многие программисты возражают против использования этого метода. Самая большая проблема, связанная с использованием ключевого слова `new`, состоит в том, что его часто забывают указать, и это может самым непредсказуемым образом сказаться на работе программы.

Литеральное определение массива

Гораздо более простой и безопасный способ создания массивов заключается в использовании так называемой литеральной нотации. Это выглядит примерно так:

```
var dogNames =["Shaggy", "Tennesee", "Dr. Spock"];
```

Вот и все, что вам нужно сделать. Больше от вас ничего не требуется. Использование квадратных скобок вместо ключевых слов снижает вероятность того, что вы о чем-то случайно забудете. Кроме того, литеральный метод определения массивов требует записи меньшего количества символов, чем в случае использования ключевого слова `new`. И если вы хотите, чтобы ваш JavaScript-код выглядел как можно более аккуратным, то любая мелочь важна!

Заполнение массивов значениями

Можно либо добавить значения в массив в момент его создания, либо сначала создать массив, а затем добавлять в него элементы по мере необходимости. Добавление элементов в массив происходит точно так же, как создание или изменение переменной, за исключением того, что необходимо указывать индекс создаваемого или изменяемого элемента. В листинге 4.2 приведен пример создания пустого массива с последующим добавлением в него элементов.

Листинг 4.2. Заполнение пустого массива

```
var peopleList =[];
peopleList[0] = "Chris Minnick";
peopleList[1] = "Eva Holland";
peopleList[2] = "Abraham Lincoln";
```

Вы не обязаны строго соблюдать последовательность заполнения массива элементами. В этом примере вполне допустимой была бы следующая инструкция:

```
peopleList[99] = "Tina Turner";
```

Такой способ добавления элемента в массив дополнительно сопровождается созданием пустых элементов для всех индексов между `peopleList[2]` и `peopleList[99]`.

Следовательно, если вы проверите свойство `length` массива `peopleList` после добавления в него элемента с индексом 99, то получите следующий интересный результат:

```
peopleList.length // возвращает 100
```

Несмотря на то что вы создали всего четыре элемента, JavaScript сообщит вам, что массив имеет длину 100, поскольку эта характеристика основана на самом высоком значении индекса занятого элемента, а не на количестве фактически созданных элементов.

Многомерные массивы

Разрешается не только хранить массивы в массивах, но и помещать массивы в массивы, хранящиеся в других массивах. Этот процесс может продолжаться и далее.

Массив, который содержит другой массив, называется *многомерным*. Для записи многомерного массива нужно просто добавить необходимое количество парных квадратных скобок после имени переменной, например:

```
var listOfLists[0][0];
```

Возможно, вам будет трудно представить себе, что такое многомерные массивы, когда вы только начнете работать с ними. Структуру многомерного массива иллюстрирует пример, приведенный на рис. 4.2, на котором представлено разбиение лучших музыкальных альбомов по жанрам: рок, кантри, панк.

Рис. 4.2. Графическое представление многомерного массива

Многомерный массив можно представлять себе в виде иерархической структуры списков, как показано ниже.

1. Country
   - **1.1.** 1. Johnny Cash:Live at Folsom Prison
   - **1.2.** Patsy Cline:Sentimentally Yours
   - **1.3.** Hank Williams:I'm Blue Inside
2. Rock
   - **2.1.** T-Rex:Slider
   - **2.2.** Nirvana:Nevermind
   - **2.3.** Lou Reed:Transformer
3. Punk
   - **3.1.** Flipper:Generic
   - **3.2.** The Dead Milkmen:Big Lizard in my Backyard
   - **3.3.** Patti Smith:Easter

А вот как выглядит код, который создает массив, представленный на рис. 4.2.

```
var bestAlbumsByGenre = [];
bestAlbumsByGenre[0] = "Country";
bestAlbumsByGenre[0][0] = "Johnny Cash:Live at Folsom
        Prison"
bestAlbumsByGenre[0][1] = "Patsy Cline:Sentimentally
        Yours";
bestAlbumsByGenre[0][2] = "Hank Williams:I'm Blue Inside";
bestAlbumsByGenre[1] = "Rock";
bestAlbumsByGenre[1][0] = "T-Rex:Slider";
bestAlbumsByGenre[1][1] = "Nirvana:Nevermind";
bestAlbumsByGenre[1][2] = "Lou Reed:Tranformer";
bestAlbumsByGenre[2] = "Punk";
bestAlbumsByGenre[2][0] = "Flipper:Generic";
bestAlbumsByGenre[2][1] = "The Dead Milkmen:Big Lizard in
        my Backyard";
bestAlbumsByGenre[2][2] = "Patti Smith:Easter";
```

Доступ к элементам массива

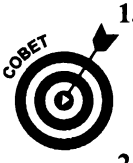
Доступ к элементам массива осуществляется также с использованием квадратных скобок и индекса. Например, чтобы получить доступ к третьему элементу массива `myArray`, вы должны использовать следующую запись:

```
myArray[2];
```

Для доступа к элементам многомерного массива добавляются дополнительные квадратные скобки:

```
bestAlbumsByGenre[0][1];
    // возвращает "Patsy Cline:Sentimentally Yours";
```

Чтобы протестировать присвоение и получение значений элементов многомерного массива, выполните следующие действия.



1. **Откройте браузер Chrome и его консоль JavaScript.**

   Для открытия консоли JavaScript можно использовать меню Chrome либо нажать комбинацию клавиш <Command+Option+J> (Mac) или <Ctrl+Shift+J> (Windows).

2. **Создайте массив `valuesOfNumber`, введя в консоли следующую инструкцию и нажав клавишу <Return> или <Enter>:**

   ```
   var valuesOfNumber = [2,4,1.5,80];
   ```

3. **Чтобы получить значение элемента, введите имя массива, указав вслед за ним индекс интересующего вас элемента в квадратных скобках, например:**

   ```
   valuesOfNumber[0];
   valuesOfNumber[3];
   valuesOfNumber[2];
   ```

4. **Попытайтесь использовать индекс несуществующего элемента массива:**

```
valuesOfNumber[4];
```

Обратите внимание на то, что значение этого элемента не определено.

5. **Введите следующую команду для создания новой переменной, в которой хранится сумма имеющихся чисел.**

```
var totalValue = valuesOfNumber[0] +
valuesOfNumber[1] + valuesOfNumber[2] +
valuesOfNumber[3];
```

6. **Наконец, получите значение `valuesOfNumber`, введя следующую команду:**

```
totalValue;
```

Перемещение по элементам массива в цикле

Нетрудно догадаться, что с ростом числа элементов в массиве вводить каждый раз имя массива и индекс элемента становится все более утомительно. К счастью, для работы со всеми элементами массива существуют гораздо более легкие способы. Самый распространенный из них — использование конструкции, которая называется *циклом*. (Более подробно о циклах речь пойдет в главе 6.)

Свойства массивов

О массивах можно получать некоторую информацию, используя *свойства массивов*. Один из способов доступа к свойствам связан с использованием так называемой *точечной нотации*. Эта нотация предполагает, что имя свойства указывается после имени массива и отделяется от него точкой. (Более подробно о свойствах вы узнаете в главе 8.) Свойства массивов JavaScript перечислены в табл. 4.1.

Таблица 4.1. Свойства массивов JavaScript

Свойство	Возвращаемое значение
`prototype`	Позволяет добавлять свойства и методы в объект `Array`
`constructor`	Ссылка на функцию, создавшую прототип объекта `Array`
`length`	Возвращает или устанавливает количество элементов в массиве

Чаще всего используется свойство `length`. До этого вы уже имели возможность увидеть, как оно работает. Это свойство позволяет узнать количество элементов в массиве, включая и те, которым значения еще не присвоены.

```
var myArray = [];
myArray[2000];
myArray.length; // возвращает 2001
```

Кроме того, вы можете использовать это свойство для усечения массива.

```
myArray.length; // возвращает 2001
myArray.length = 10;
myArray.length; // возвращает 10
```

Методы для работы с массивами

Методы (также называемые функциями) для работы с массивами предоставляют удобные способы манипулирования их элементами. Перечень этих методов вместе с описанием их назначения или возвращаемых значений приведен в табл. 4.2.

Использование методов для работы с массивами

Синтаксис методов для работы с массивами зависит от метода, который вы хотите использовать. Однако доступ к функциональности каждого из них осуществляется аналогично доступу к свойствам массива — с использованием точечной нотации.

Таблица 4.2. Методы для работы с массивами в JavaScript

Метод	Возвращаемое значение
`concat()`	Новый массив, содержащий текущий массив и дополненный другими массивами и элементами
`every()`	Возвращает значение `true`, если каждый элемент массива удовлетворяет условию, предоставляемому тестовой функцией
`filter()`	Новый массив, который содержит все элементы текущего массива, удовлетворяющие условию, предоставляемому заданной тестовой функцией
`forEach()`	Выполняет заданную функцию для каждого элемента массива
`indexOf()`	Индекс первого вхождения заданного значения в массиве. Если значение не обнаружено, то возвращается `-1`
`join()`	Объединяет все элементы массива в одну строку
`lastIndexOf()`	Индекс последнего вхождения заданного значения в массиве. Если значение не обнаружено, возвращается `-1`
`map()`	Новый массив, полученный путем преобразования каждого элемента текущего массива с помощью заданной функции
`pop()`	Удаляет последний элемент из массива
`push()`	Добавляет новые элементы в конец массива
`reduce()`	Сводит два значения массива в одно, применяя к обоим заданную функцию (слева направо)
`reduceRight()`	Сводит два значения массива в одно, применяя к обоим заданную функцию (справа налево)
`reverse(0`	Обращает порядок следования элементов в массиве
`shift()`	Удаляет первый элемент из массива и возвращает его, изменяя длину массива
`slice()`	Выбирает часть массива и возвращает его в виде нового массива
`some()`	Возвращает значение `true`, если один или несколько элементов массива удовлетворяет условию, предоставляемому тестовой функцией
`sort()`	Возвращает отсортированный массив (по умолчанию сортировка выполняется в алфавитном порядке и в порядке возрастания)
`splice()`	Возвращает новый массив, состоящий из элементов, которые были добавлены или удалены из данного массива
`toString()`	Преобразует массив в строку
`unShift()`	Возвращает новый массив другой длины, к которому добавлены один или несколько элементов

Полную справку по методам работы с массивами JavaScript можно найти по такому адресу:

`https://docs.webplatform.ofg/wiki/javascript/Array#Methods`

В листинге 4.3 приведены примеры использования некоторых наиболее употребительных методов JavaSript.

Листинг 4.3. Пример использования наиболее употребительных методов для работы с массивами JavaScript

```
<html>
<head>
  <title>Методы для работы с массивами</title>
</head>
<body>
  <script>
  var animals = ["тигр" , "медведь"];
  var fruit = ["дыня" , "апельсин"];
  var dishes = ["тарелка" , "бокал" , "чашка"];

  var whereIsTheTiger = animals.indexOf("тигр");
  var fruitsAndAnimals = fruit.concat(animals);

  document.write (animals + "<br>");
  document.write ("Индекс тигра: "
          + whereIsTheTiger + "<br>");
  document.write (fruitsAndAnimals + "<br>");
  whereIsTheTiger = animals.indexOf("тигр");
  document.write ("Индекс тигра: "
          + whereIsTheTiger + "<br>");
  </script>
</body>
</html>
```

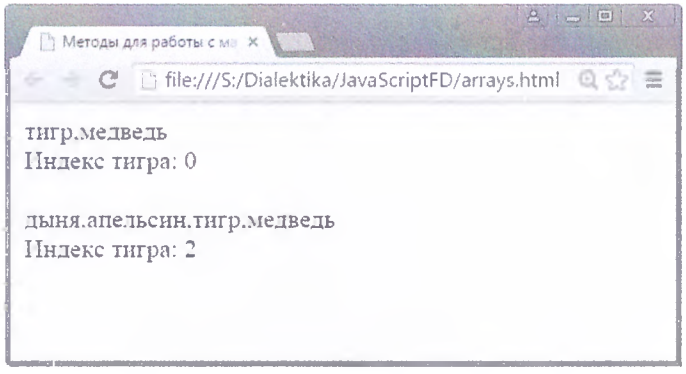


Рис. 4.3. Пример использования наиболее употребительных методов для работы с массивами JavaScript

Глава 5

Операторы, выражения, инструкции

В этой главе...

- Чтение и запись выражений JavaScript
- Изменение значений с помощью оператора присваивания
- Логическое ветвление с помощью операторов сравнения
- Выполнение вычислений с помощью арифметических операторов
- Знакомство с поразрядными операторами
- Строковые операторы

"Алло, оператор! Можете соединить меня с номером 9?"
The White Stripes

Операторы, выражения и инструкции JavaScript — это основные строительные кирпичики программ. С их помощью вы манипулируете значениями, сравниваете их между собой, проводите вычисления и решаете массу других важных задач.

В этой главе вы познакомитесь с тем, как работают операторы, выражения и инструкции, и узнаете о том, как использовать их с максимальной пользой для себя.

Выражения

Выражение — это часть кода, которая преобразуется в значение (может быть вычислена с получением значения). Выражение может существовать само по себе, либо его значение может присваиваться переменной. Ниже приведен пример двух таких выражений.

```
1 + 1
a = 1;
```

Выражения могут быть как короткими и простыми, так и очень сложными.

Отдельные элементы данных (`1` и `a` в предыдущих примерах), участвующие в выражениях, называются *операндами*.

Знакомство с операторами

То, что воздействует на выражения и заставляет их работать, называется *оператором*. Воздействуя на данные, операторы производят различные результаты. В предыдущих примерах операторами служат + и =.

Приоритет операторов

В одном выражении часто содержится несколько операторов. Рассмотрим следующий пример:

```
a = 1 + 2 * 3 / 4;
```

В зависимости от очередности выполнения отдельных вычислений мы можем получить один из следующих конечных результатов.

```
a = 1.75;
a = 2.5;
a = 2.25;
```

В действительности правильным результатом является `2.5`. Но откуда мы это знаем? Вообще говоря, кто-то мог бы выполнить сначала деление (3/4), кто-то — сложение (1+2), а кто-то — умножение (2*3).

Ясно, что для нахождения правильного ответа должен существовать какой-то способ, и такой способ действительно существует! А решает все *приоритет операторов*. Приоритет операторов определяет очередность их выполнения в выражении.

Все операторы делятся на группы с различными уровнями приоритета от 0 до 19 (табл. 5.1).

Таблица 5.1. Приоритет операторов

Оператор	Описание	Ассоциативность	Приоритет	Пример
`(...)`	Группирование	Неприменимо	0 — наивысший приоритет	`(1 + 3)`
`.`	Доступ к свойству	Слева направо	1	`myCar.color`
`[...]`	Доступ к массиву	Слева направо	1	`thingsToDo[4]`
`new...()`	Создание объекта (со списком аргументов)	Неприменимо	1	`new Car("red")`
`function...()`	Вызов функции	Слева направо	2	`function addNumbers(1,2)`
`new...`	Создание объекта (без списка аргументов)	Справа налево	2	`new Car`
`...++`	Постфиксный инкремент	Неприменимо	3	`number++`
`...--`	Постфиксный декремент	Неприменимо	3	`number--`
`!...`	Логическое НЕ	Справа налево	4	`!myVal`
`~...`	Поразрядное НЕ	Справа налево	4	`~myVal`
`-...`	Отрицание	Справа налево	4	`-aNumber`
`++...`	Префиксный инкремент	Справа налево	4	`++aNumber`
`--...`	Префиксный декремент	Справа налево	4	`--aNumber`

Продолжение табл. 5.1

Оператор	Описание	Ассоциативность	Приоритет	Пример
`typeof...`	Информация о типе	Справа налево	4	`typeof myVar`
`void...`	Вычисляет выражение и возвращает `undefined`	Справа налево	4	`void(0)`
`delete...`	Удаление	Справа налево	4	`delete object.property`
`...*...`	Умножение	Слева направо	5	`result = 3 * 7`
`.../...`	Деление	Слева направо	5	`result = 3 / 7`
`...%...`	Остаток	Слева направо	5	`result = 7 % 3`
`...+...`	Сложение	Слева направо	6	`result = 3 + 7`
`...-...`	Вычитание	Слева направо	6	`result = 3 - 7`
`...<<...`	Поразрядный сдвиг влево	Слева направо	7	`result = 3 << 7`
`...>>...`	Поразрядный сдвиг вправо	Слева направо	7	`result = 3 >> 7`
`...>>>...`	Поразрядный сдвиг вправо без знака	Слева направо	7	`result = 3 >>> 7`
`...<...`	Меньше чем	Слева направо	8	`a < b`
`...<=...`	Меньше чем или равно	Слева направо	8	`a <= b`
`...>...`	Больше чем	Слева направо	8	`a > b`
`...>=...`	Больше чем или равно	Слева направо	8	`a >= b`
`...in...`	Проверка наличия свойства	Слева направо	8	`value in values`
`...instanceof...`	Проверка принадлежности к типу	Слева направо	8	`myCar instanceof car`
`...==...`	Проверка равенства	Слева направо	9	`3 == "3" //true`
`...!=...`	Проверка неравенства	Слева направо	9	`3 !== "3" //false`
`...===...`	Проверка строгого равенства	Слева направо	9	`3 ==="3" //false`
`...!==...`	Проверка строгого неравенства	Слева направо	9	`3 != "3" //true`
`...&...`	Поразрядное И	Слева направо	10	`result = a & b`
`...^...`	Поразрядное исключающее ИЛИ	Слева направо	11	`result = a ^ b`
`...|...`	Поразрядное ИЛИ	Слева направо	12	`result = a | b`
`...&&...`	Логическое И	Слева направо	13	`a && b`
`...||...`	Логическое ИЛИ	Слева направо	14	`a || b`

Окончание табл. 5.1

Оператор	Описание	Ассоциативность	Приоритет	Пример
`...?...:...`	Тернарный условный оператор	Справа налево	15	`a ? 3 : 7`
`...=...`	Присваивание	Справа налево	16	`a = 3`
`...+=...`	Присваивание	Справа налево	16	`a += 3`
`...-=...`	Присваивание	Справа налево	16	`a -= 3`
`...*=...`	Присваивание	Справа налево	16	`a *= 3`
`.../=...`	Присваивание	Справа налево	16	`a /= 3`
`...%=...`	Присваивание	Справа налево	16	`a %= 3`
`...<<=...`	Присваивание	Справа налево	16	`a <<= 3`
`...>>=...`	Присваивание	Справа налево	16	`a >>= 3`
`...>>>...`	Присваивание	Справа налево	16	`a >>> 3`
`...&=...`	Присваивание	Справа налево	16	`a &= 3`
`...^=...`	Присваивание	Справа налево	16	`a ^= 3`
`...|=...`	Присваивание	Справа налево	16	`a |= 3`
`yield...`	Остановка и возобновление функции-генератора	Справа налево	17	`yield[`*выражение*`]`
	Вычисляет оба операнда и возвращает значение второго	Слева направо	18	`a + b, c + d`

Оператор с наименьшим значением уровня приоритета обладает наивысшим приоритетом. Поначалу это может вас несколько смущать, но стоит вам задуматься над тем, что первый из стоящих в очереди (в нашем случае ему соответствует уровень приоритета 0) будет также первым, кому достанется великолепный сэндвич или чашечка дымящегося кофе, то в голове у вас сразу же сложится правильная картина.

Если выражение содержит два или несколько операторов с одинаковым приоритетом, то очередность их выполнения определяется их *ассоциативностью* — характеристикой оператора, определяющей направление вычислений: слева направо или справа налево.

Использование скобок

В выражениях оператором с наивысшим приоритетом являются круглые скобки. В большинстве случаев правила, определяющие приоритет операторов, можно игнорировать, группируя операции в подвыражения с помощью скобок. Например, выражение с несколькими операторами из предыдущего примера можно было бы записать одним из следующих способов, каждый из которых делает намерения программиста очевидными.

```
a = (1 + 2) * (3 / 4); // результат: 2.25
a = (1 + (2 * 3)) / 4; // результат: 1.75
a = ((1 + 2) *3) / 4;  // результат: 2.25
a = 1 + ((2 * 3) / 4); // результат: 2.5
```

Скобки в выражении вынуждают интерпретатор JavaScript вычислять сначала их содержимое, продвигаясь от внутренних скобок к внешним, и лишь затем выполнять операции вне скобок.

Сверившись с табл. 5.1, нетрудно заключить, что фактическая очередность выполнения операций в рассматриваемом примере должна быть такой:

```
a = 1 + ((2 * 3) / 4);
```

Инструкция в таком виде явно учитывает приоритет выполнения операций. Сначала выполняется умножение, затем деление и только после этого — сложение.

Типы операторов

Операторы JavaScript делятся на типы. В этом разделе обсуждаются наиболее часто используемые типы операторов.

Операторы присваивания

Оператор присваивания записывает значение операнда, находящегося справа от него, в операнд, находящийся слева:

```
a = 5;
```

После выполнения этого выражения переменная `a` будет содержать значение `5`. Операторы присваивания могут объединяться в цепочки для того, чтобы присвоить одно и то же значение сразу нескольким переменным:

```
a = b = c = 5;
```

Поскольку ассоциативность оператора присваивания — справа налево (см. табл. 5.1), сначала значение будет присвоено переменной `c`, затем значение `c` будет присвоено переменной `b`, после чего значение `b` будет присвоено переменной `a`. Результат выполнения этого выражения состоит в том, что каждая из переменных `a`, `b` и `c` будет иметь значение `5`.

Как вы думаете, какое значение будет иметь переменная `a` после выполнения приведенных ниже двух выражений?

```
var b = 1;
var a = b += c = 5;
```

Чтобы найти ответ на этот вопрос, откройте консоль JavaScript в Chrome и поочередно введите эти строки, каждый раз нажимая клавишу <Return> или <Enter>. В результате вы получите `6`.

Полный список операторов присваивания приведен в разделе "Объединение операторов".

Операторы сравнения

Операторы сравнения проверяют равенство или неравенство операндов и возвращают в качестве результата значение `true` или `false`.

Полный список операторов сравнения в JavaScript приведен в табл. 5.2.

Таблица 5.2. Операторы сравнения в JavaScript

Оператор	Описание	Пример
==	Равно	`3 == "3" //true`
!=	Не равно	`3 != "3" //false`
===	Строго равно	`3 === "3" //false`
!==	Строго не равно	`3 !== "3" //true`
>	Больше	`7 > 1 //true`
>=	Больше или равно	`7 > = 7 //true`
<	Меньше	`7 < 10 //true`
<=	Меньше или равно	`2 <=2 //true`

Арифметические операторы

Арифметические операторы выполняют математические операции над операндами и возвращают результат. Полный список арифметических операторов приведен в табл. 5.3.

Таблица 5.3. Арифметические операторы

Оператор	Описание	Пример
+	Сложение	`a = 1 + 1`
-	Вычитание	`a = 10 - 1`
*	Умножение	`a = 2 * 2`
/	Деление	`a = 8 / 2`
%	Взятие остатка	`a = 5 % 2`

++	Инкремент	`a = ++b`
		`a = b++`
		`a = ++`

--	Декремент	`a = --b`
		`a = b--`
		`a--`

Работа арифметических операторов продемонстрирована в листинге 5.1.

Листинг 5.1. Использование арифметических операторов

```
<html>
<head>
  <title>Арифметические операторы</title>
</head>
<body>
  <h1>Отгадывание даты рождения</h1>
  <p>
  <ul>
    <li>Введите число 7</li>
    <li>Умножьте его на месяц вашего рождения</li>
    <li>Вычтите 1</li>
    <li>Умножьте результат на 13</li>
    <li>Добавьте число вашего дня рождения</li>
    <li>Добавьте 3</li>
    <li>Умножьте результат на 11</li>
    <li>Вычтите месяц вашего рождения</li>
    <li>Вычтите число вашего дня рождения</li>
    <li>Разделите результат на 10</li>
    <li>Добавьте 11</li>
    <li>Разделите результат на 100</li>
  </ul>
  </p>
  <script>
    var numberSeven = Number(prompt('Введите число 7'));
    var birthMonth = Number(prompt('Введите месяц вашего рождения'));
    var calculation = numberSeven * birthMonth;
    calculation = calculation - 1;
    calculation = calculation * 13;
    var birthDay = Number(prompt('Введите число вашего дня рождения'));
    calculation = calculation + birthDay;
    calculation = calculation + 3;
    calculation = calculation * 11
    calculation = calculation - birthMonth;
    calculation = calculation - birthDay;
    calculation = calculation / 10;
    calculation = calculation + 11;
    calculation = calculation / 100;
    document.write("Вы родились " + calculation);
  </script>
</body>
</html>
```

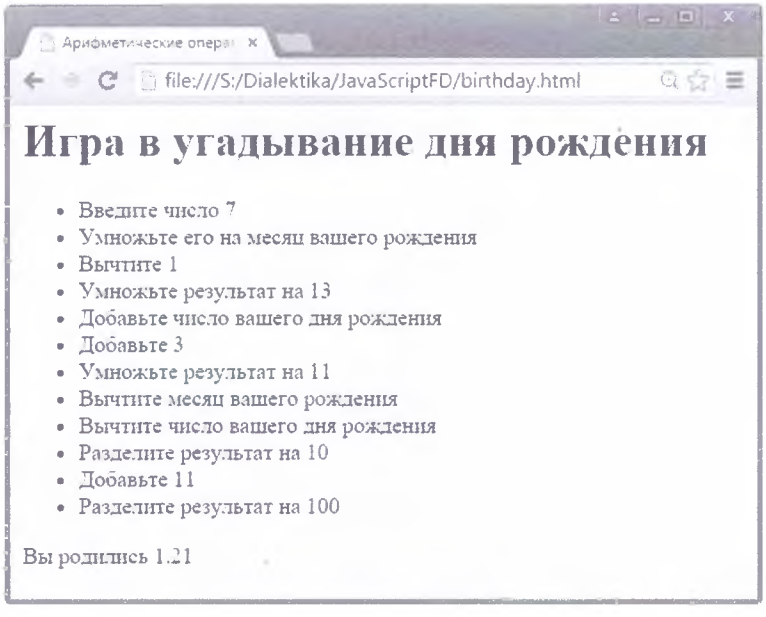


Рис. 5.1. Игра в угадывание дня рождения[1]

Строковый оператор

Строковый оператор выполняет операции с использованием двух строк. В случае строк оператор "+" становится оператором конкатенации, который объединяет две строки в одну. Обратите внимание на то, что при объединении строк с помощью оператора конкатенации никакого добавления пробелов не происходит. Именно поэтому нередко встречаются инструкции наподобие приведенной ниже, в которых строки, не содержащие ничего, кроме пробела, добавляются в конце или в начале строк (но вне их кавычек) для образования связного предложения.

```
var greeting = "Привет, " + firstName + ". Я" + " " + mood +
               " тебя видеть.";
```

Поразрядные операторы

Поразрядные (побитовые) операторы обрабатывают операнды как 32-разрядные (32-битовые) целые двоичные числа со знаком, представленные в дополнительном коде (дополнение до 2). Поясним, что все это означает, начиная с термина *двоичный*.

Двоичные числа — это строки, состоящие из нулей и единиц, в которых значение каждой единицы определяется ее позицией в данной строке. Вот так, например, выглядит число 1, записанное в формате 32-разрядного целого двоичного числа:

```
00000000000000000000000000000001
```

[1] Здесь дата выводится в формате *месяц.день*. — *Примеч. ред.*

Крайняя справа позиция имеет значение 1. Каждый сдвиг влево на одну позицию означает удвоение значения, соответствующего предыдущей позиции, находящейся справа. Поэтому число 5 имеет следующее двоичное представление:

```
00000000000000000000000000000101
```

Термин *целые числа со знаком* означает, что в этой форме могут быть представлены как положительные, так и отрицательные целые числа.

Термин *дополнение до 2* означает, что для получения противоположного по знаку числа необходимо обратить (инвертировать) значения всех битов (разрядов) данного числа и к полученному результату прибавить 1. Таким образом, числу -5 соответствует следующее двоичное представление:

```
11111111111111111111111111111010
```

Поразрядные операторы сначала преобразуют то, что мы считаем обычными числами, в 32-разрядные двоичные числа, а после выполнения операции совершают обратное преобразование.

Поначалу новичкам трудно разобраться в том, как работают поразрядные операторы. Они используются сравнительно редко, но не рассказать вам о них было бы непростительно с нашей стороны.

Поразрядные операторы JavaScript перечислены в табл. 5.4.

Таблица 5.4. Поразрядные операторы

Оператор	Использование	Описание
Поразрядное И (AND)	`a & b`	Возвращает 1 в тех позициях результата, в которых биты каждого из операндов равны 1
Поразрядное ИЛИ (OR)	`a | b`	Возвращает 1 в тех позициях результата, в которых бит хотя бы одного из операндов равен 1
Поразрядное исключающее ИЛИ (XOR)	`a ^ b`	Возвращает 1 в тех позициях результата, в которых бит только одного из операндов равен 1
Поразрядное НЕ (NOT)	`~a`	Заменяет каждый бит операнда на противоположный
Левый сдвиг	`a << b`	Сдвигает двоичное представление `a` на `b` разрядов влево, заполняя освобождающиеся позиции нулями
Правый сдвиг с переносом знака[2]	`a >> b`	Сдвигает двоичное представление `a` на `b` (<32) разрядов вправо, отбрасывая уходящие биты
Правый сдвиг с заполнением нулями	`a >>> b`	Сдвигает двоичное представление `a` на `b` (<32) разрядов вправо, отбрасывая уходящие биты и заполняя освобождающиеся биты нулями

[2] При этом освобождающиеся биты заполняются предыдущим значением старшего (знакового) бита. В связи с этим интересно отметить, что сдвиг вправо значения –1 на любое число разрядов вновь дает значение –1. — *Примеч. ред.*

На рис. 5.2 представлен пример, демонстрирующий работу каждого из поразрядных операторов в консоли JavaScript браузера Chrome.

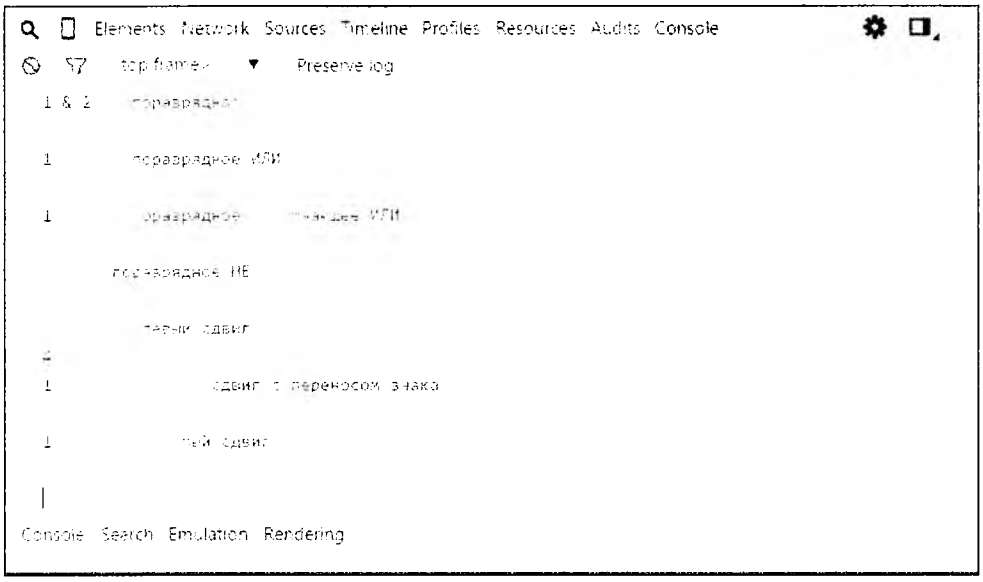


Рис. 5.2. Поразрядные операторы JavaScript

Логические операторы

Логические операторы вычисляют логическое выражение и определяют его истинность или ложность. В JavaScript есть три логических оператора (табл. 5.5).

Таблица 5.5. Логические операторы

Оператор	Название	Описание
`&&`	Логическое И (AND)	Возвращает первый операнд, если он равен `false`, иначе — второй операнд
`||`	Логическое ИЛИ (OR)	Возвращает первый операнд, если он равен `true`, иначе — второй операнд
`!`	Логическое НЕ (NOT)	Принимает лишь один операнд. Возвращает `false`, если операнд может быть преобразован в `true`, иначе возвращает `true`

С помощью оператора `||` можно задавать значения по умолчанию для переменных. Например, в приведенном ниже выражении для переменной `myVar` будет установлено значение `x` при условии, что приведение `x` к булеву значению не дает `false` (например, если переменная `x` не определена). В противном случае для переменной будет установлено значение по умолчанию, равное `0`.

```
var myVar = x || 0;
```

Специальные операторы

Специальные операторы JavaScript — это пестрая смесь символов и слов, которые выполняют различные важные функции.

Условный оператор

Условный оператор (также известный как *тернарный оператор*) использует три операнда. Он вычисляет логическое выражение, а затем возвращает значение, зависящее от того, какой результат был получен: `true` или `false`. Условный оператор — единственный, который требует трех операндов.

```
var isItBiggerThanTen = (value > 10) ? "больше 10" :
                                       "не больше 10";
```

Оператор запятая

Оператор запятая (`,`) вычисляет два операнда и возвращает значение второго из них. Чаще всего он используется для выполнения нескольких присваиваний или других операций внутри циклов. Он также может использоваться для сокращенной записи инструкций инициализации переменных:

```
var a = 10, b = 0;
```

Поскольку оператор запятая имеет наинизший приоритет среди всех операторов, его операнды всегда вычисляются по отдельности.

Оператор `delete`

Оператор `delete` удаляет свойство из объекта или элемент из массива.

Длина массива после удаления из него элемента с помощью оператора `delete` остается прежней. Удаленный элемент принимает значение `undefined`.

```
var animals = ["dog","cat","bird","octopus"];
console.log (animals[3]); // возвращает "octopus"
delete animals[3];
console.log (animals[3]); // возвращает "undefined"
```

Оператор `in`

Оператор `in` возвращает `true`, если указанное значение имеется в массиве или объекте.

```
var animals = ["dog","cat","bird","octopus"];
if (3 in animals) {
  console.log ("Это значение есть в массиве");
}
```

В этом примере, если в массиве `animals` имеется элемент с индексом `3`, в консоли JavaScript отобразится строка "Это значение есть в массиве".

Оператор instanceof

Оператор instanceof возвращает true, если указанный объект относится к указанному типу.

```
var myString = new String();
if (myString instanceof String) {
  console.log("Да, это строка!");
}
```

Оператор new

Оператор new создает экземпляр объекта. Как вы увидите в главе 8, в JavaScript имеется несколько встроенных объектных типов, и вы можете определять собственные типы. В следующем примере Date() — встроенный объект JavaScript, тогда как Pet() и Flower() — пример экземпляров объектов, которые программист может создавать для решения нестандартных задач в программе.

```
var today = new Date();
var bird = new Pet();
var daisy = new Flower();
```

Оператор this

Оператор this возвращает текущий объект. Он часто используется для получения свойств внутри объекта.

Более подробно оператор this рассматривается в главе 8.

Оператор typeof

Оператор typeof возвращает строку, содержащую тип операнда.

```
var businessName = "Minimax";
console.log typeof businessName; // возвращает "string"
```

Оператор void

Оператор void приводит к вычислению выражения операнда без возвращения значения. Чаще всего оператор void можно встретить в HTML-документах, когда нужна ссылка, но создатель ссылки хочет переопределить ее поведение по умолчанию, используя JavaScript.

```
<a href="javascript:void(0);">Это ссылка, но она
         ничего не делает</a>
```

Объединение операторов

Операцию присваивания переменной результата вычисления выражения можно записать в краткой форме, объединяя оператор присваивания с другими операторами. Например, следующие два выражения дают один и тот же результат.

```
a = a + 10;
a += 10;
```

Все допустимые сочетания оператора присваивания с другими операторами приведены в табл. 5.6.

Таблица 5.6. Допустимые сочетания оператора присваивания с другими операторами

Операция	Сокращенная запись	Стандартная запись
Присваивание	`x = y`	`x = y`
Сложение с присваиванием	`x += y`	`x = x + y`
Вычитание с присваиванием	`x -= y`	`x = x - y`
Умножение с присваиванием	`x *= y`	`x = x * y`
Деление с присваиванием	`x /= y`	`x = x / y`
Взятие остатка с присваиванием	`x %= y`	`x = x % y`
Левый сдвиг с присваиванием	`x <<= y`	`x = x << y`
Правый сдвиг с присваиванием	`x >>= y`	`x = x >> y`
Правый сдвиг с заполнением нулями с присваиванием	`x >>>= y`	`x = x >>> y`
Поразрядное И с присваиванием	`x &= y`	`x = x & y`
Поразрядное исключающее ИЛИ с присваиванием	`x ^= y`	`x = x ^ y`
Поразрядное ИЛИ с присваиванием	`x |= y`	`x = x | y`

Глава 6

Циклы и ветвление кода

В этой главе...

- Ветвление кода с помощью конструкции `if...else`
- Различные типы циклов
- Использование циклов для многократного повторения инструкций
- Перечисление элементов массива в цикле

"Не сложно принимать решения, когда осознаешь свои ценности".
Рой Дисней

В предыдущих главах обсуждался и демонстрировался, как правило, линейный код JavaScript. Однако нередко (а в действительности очень часто) наступает момент, когда в программе требуется организовать принятие какого-либо решения относительно дальнейших действий или изменить прямолинейную логику программы таким образом, чтобы один и тот же набор инструкций выполнялся с различными значениями переменных. Эта глава посвящена операторам цикла и ветвления.

Ветвление кода

Инструкции циклов и ветвления кода называют *управляющими инструкциями*, поскольку они позволяют управлять последовательностью выполнения программ JavaScript. *Инструкции ветвления* используются для создания различных путей выполнения программы в зависимости от логики условий. *Циклы* — это простейший способ группирования JavaScript-инструкций в программе.

При разработке логики JavaScript-программ часто наступает момент, когда для определения дальнейшего пути выполнения требуется сделать определенный выбор. На рис. 6.1 приведен иллюстративный JavaScript-код, демонстрирующий осуществление выбора в реальной ситуации с использованием ветвления кода.

`if...else`

Инструкции `if` и `else` работают совместно, вычисляя логическое выражение и выполняя различные инструкции в зависимости от результата. Инструкции `if` часто используются без инструкции `else`, тогда как последняя должна использоваться только совместно с инструкцией `if`.

Рис. 6.1. Выбор дальнейшего пути с помощью ветвления

Вот как выглядит базовый синтаксис инструкции `if`.

```
if (условие) {
...
}
```

В данном случае *условие* — это любое выражение, вычисление которого дает булево значение (`true` или `false`). Если результат вычисления выражения равен `true`, то выполняются инструкции, заключенные в фигурные скобки. Если результат равен `false`, то эти инструкции пропускаются.

Инструкция `else` пригодится в тех случаях, когда требуется выполнить какие-либо другие действия, даже если результат вычисления условия равен `false`.

```
var age = 17;
if (age < 18){
  document.write ("Продажа алкоголя несовершеннолетним
                   запрещена.");
} else {
  document.write ("Чем это закончится?");
}
```

Во многих других языках программирования имеется также ключевое слово `elseif`, которое может неоднократно использоваться в конструкции `if...else` до тех пор, пока вычисление условия не даст значение `true`. В JavaScript это ключевое слово отсутствует.

Однако функциональность, обеспечиваемую ключевым словом `elseif`, можно получить, записав подряд `if` и `else` с пробелом между ними.

```
if (time < 12){
  document.write ("Доброе утро!");
```

```
} else if (time < 17){
  document.write ("Добрый день!");
} else if (time < 20){
  document.write ("Добрый вечер!");
} else {
  document.write ("Доброй ночи!");
}
```

Сокращенная запись инструкции `if...else`

Вам будет полезно познакомиться с двумя вариантами сокращенной записи инструкции `if...else`. Первый из них — это использование тернарного оператора. Инструкцию в этой форме несколько труднее читать, чем стандартную инструкцию `if...else`:

```
var whatToSay = (time < 12 ?
        "Доброе утро!" : "Привет!");
```

В данном случае переменная `whatToSay` получает значение "Доброе утро!", если `time` меньше 12, и "Привет!" в противном случае.

В другом варианте сокращенной записи инструкции `if...else` предполагается использование оператора "логическое И" (`&&`). Вспомните, что оператор "логическое И" вычисляет лишь второй операнд, если вычисление первого дает `true`. Программисты называют это *неполным вычислением*, поскольку, если в логической операции И первый операнд возвращает значение `false`, то общий результат будет заведомо равен `false`, так что вычислять второй операнд нет никакой необходимости.

```
time < 12 && document.write
          ("Доброе утро!");
```

В предыдущем примере оператор `&&` сначала проверяет, что `time` меньше 12. Если условие выполняется, то в HTML-документ будет записана строка "Доброе утро!". В противном случае, в силу побочного эффекта, связанного с неполным вычислением, ничего не произойдет.

Этот метод используется не очень часто, главным образом потому, что смысл инструкции в такой форме записи может быть не до конца понятен неопытным программистам. Но если вам встретится нечто подобное, то вы теперь знаете, как работают подобные инструкции.

Обратите внимание на использование символов разрыва строки и пробела в предыдущих примерах. Каждый программист волен самостоятельно выбирать наиболее удобный с его точки зрения стиль форматирования инструкций `if...else`. Вы могли видеть их записанными с использованием большего или меньшего количества строк или без пробелов между ключевыми словами и скобками. Все эти способы равноценны. Однако в любом случае следует всегда ставить во главу угла удобочитаемость кода, а не его компактность.

Инструкция `switch`

Инструкция `switch` (переключатель) позволяет выбрать, какая из нескольких инструкций должна быть выполнена, исходя из возможных значений заданного выражения. Каждое из таких значений называется *вариантом* (case). На обычном языке работу этой инструкции можно описать примерно так:

> "В том случае, если ожидается 6 гостей, заказать три пиццы. Если ожидается 12 гостей, то заказать 6 пицц. Если же ожидается более 20 гостей, то сойти с ума".

Инструкция `switch` имеет следующий синтаксис.

```
switch (выражение) {
  case значение1:
  // Инструкции
  break;
  case значение2:
  // Инструкции
  break;
  case значение3:
  // Инструкции
  break;
  default:
  // Инструкции
  break;
}
```

Обратите внимание на инструкцию `break`, которая указывается вслед за каждой группой инструкций, соответствующей определенному варианту. Каждая такая инструкция `break` обеспечивает прекращение выполнения инструкции `switch` и выход за ее пределы. В случае отсутствия инструкции `break` инструкция `switch` продолжит свою работу и выполнит операторы следующего варианта, независимо от того, совпадает ли указанное в нем значение со значением выражения, указанного в начале инструкции `switch`.

Отсутствие инструкций `break` в инструкции `switch` чревато проблемами, о чем вам никогда не следует забывать. Если после выполнения инструкций варианта, удовлетворяющего условию инструкции `switch`, окажется, что инструкция `break` отсутствует, то начнется выполнение инструкций следующего варианта, что может приводить к непредсказуемым последствиям. Порожденные этим проблемы нелегко обнаружить, поскольку обычно они приводят не к ошибкам, а к некорректным результатам.

Если ни для одного из предложений `case` совпадений со значением выражения не найдено, то инструкция `switch` осуществляет поиск варианта `default` и выполняет содержащиеся в нем инструкции.

Единственный случай, когда инструкция `break` может быть безболезненно опущена, относится к варианту `default`, указанному последним среди всех остальных (как и должно быть), поскольку вслед за инструкцией `switch` выполнение программы продолжается в любом случае.

Пример использования инструкции `switch` приведен в листинге 6.1.

Листинг 6.1. Использование инструкции `switch` для персонализации приветствия

```
var languagePreference = "испанский";
switch (languagePreference){
  case "английский":
    console.log("Hello!");
    break;
  case "испанский":
```

```
    console.log("Hola!");
    break;
  case "немецкий":
    console.log("Guten Tag!");
    break;
  case "французский":
    console.log("Bon Jour!");
    break;
  default:
    console.log("Извините, я не знаю " +
            languagePreference + " язык!");
}
```

Циклы

Циклы предназначены для многократного выполнения одного и того же набора инструкций. В JavaScript возможны циклы нескольких видов:

- ✓ `for`
- ✓ `for...in`
- ✓ `do...while`
- ✓ `while`

for

Инструкция `for` создает цикл с использованием трех выражений.

- ✓ **Инициализация.** Инициализация переменной, обычно — счетчика.
- ✓ **Условие.** Булево выражение, которое вычисляется на каждой итерации цикла.
- ✓ **Завершающее выражение.** Выражение, которое вычисляется по завершении каждой итерации цикла.

Несмотря на то что любое из этих выражений может быть опущено, в инструкцию почти всегда включают все три выражения. Обычно цикл `for` используется для выполнения кода заданное количество раз.

Ниже приведен пример использования цикла `for`.

```
for (var x = 1; x < 10; x++){
  console.log(x);
}
```

Рассмотрим выполнение этого цикла более подробно.

1. **Переменная, в данном случае `x`, инициализируется значением 1.**
2. **Осуществляется проверка того, что `x` меньше `10`.**

   Если это так, то выполняются инструкции тела цикла (в данном случае — инструкция `console.log`).
3. **Значение `x` увеличивается на 1 с помощью оператора инкремента (++).**

4. **Вновь осуществляется проверка того, что `x` меньше `10`.**

   Если это так, то выполняются инструкции тела цикла.

5. **Проверка повторяется до тех пор, пока результат вычисления выражения условия не станет ложным.**

Результат выполнения этого примера в консоли JavaScript браузера Chrome показан на рис. 6.2.

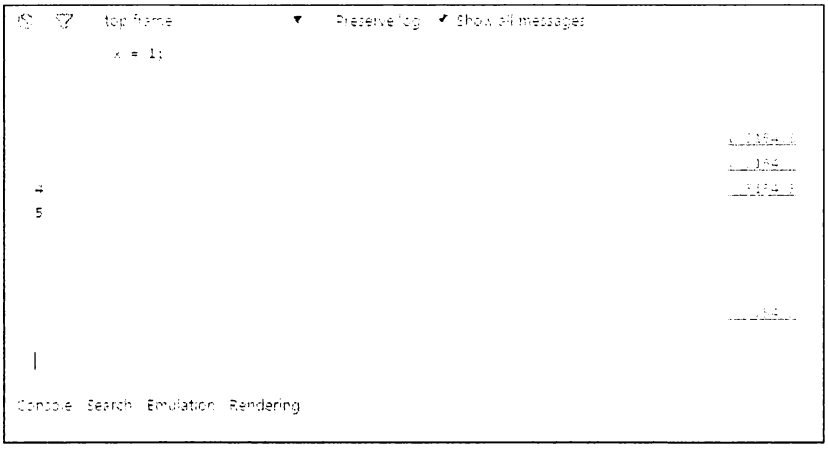


Рис. 6.2. Цикл, в котором значение счетчика изменяется от 1 до 9

Перечисление элементов массива в цикле

Циклы можно использовать для перечисления содержимого массива, сравнивая значение счетчика со значением свойства `length` массива. При этом нельзя забывать о том, что в JavaScript отсчет индексов начинается с нуля и что значение `array.length` будет всегда на единицу больше наивысшего значения индекса элемента в массиве. Именно поэтому в листинге 6.2 из значения этого свойства вычитается единица.

Листинг 6.2. Вывод содержимого массива в цикле

```
<html>
<head>
  <title>Другие телефонные коды</title>
</head>
<body>
  <script>
    var areaCodes = ["770", "404", "718", "202", "901",
          "305", "312", "313", "215", "803"];
    for (x=0; x < areaCodes.length - 1; x++){
      document.write("Другой телефонный код:" + areaCodes[x]
          + "<br>");
    }
  </script>
</body>
</html>
```

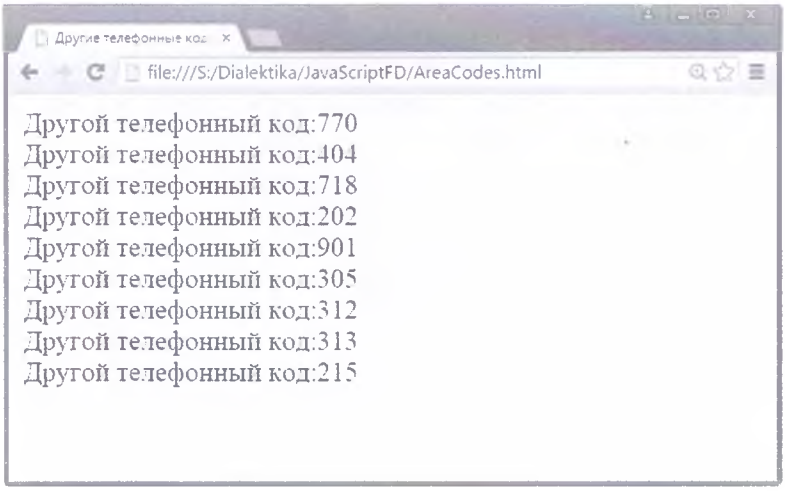


Рис. 6.3. Вывод содержимого элементов массива с помощью цикла `for`

for...in

Цикл `for...in` перебирает все свойства объекта. Его также можно использовать для перебора всех элементов массива.

Цикл `for...in` имеет одну интересную особенность. Он совершенно не учитывает, в каком порядке хранятся свойства объекта или элементы массива. По этой причине, а также потому, что цикл `for...in` работает медленнее, для работы с массивами лучше использовать стандартный цикл `for`.

Объекты — это контейнеры данных, имеющие свойства (которые определяют, что собой представляет данный объект) и методы (которые определяют, что может делать объект). В браузерах имеются наборы встроенных объектов, которые программисты могут использовать для того, чтобы управлять поведением браузера. Из них наиболее общим типом объекта является `Document`. В частности, с помощью метода `write()` этого объекта вы можете приказать браузеру вставить заданное значение в HTML-документ.

В объекте `Document` имеются также свойства, которые он использует для отслеживания информации в текущем документе и передачи ее программисту. Например, коллекция `Document.img` содержит информацию о всех тегах `img` текущего HTML-документа.

В листинге 6.3 цикл `for...in` используется для вывода всех свойств объекта `Document`.

Листинг 6.3. Перечисление свойств объекта `Document` с помощью цикла `for...in`

```
<html>
<head>
  <title>Свойства документа</title>
  <style>
    .columns {
```

```
      -webkit-column-count: 6; // Chrome, Safari, Opera
      -moz-column-count: 6; // Firefox
      column-count: 6;
    }
  </style>
</head>
<body>

  <div class="columns">

  <script>
    for (var prop in document){
      document.write (prop + "<br>");
  }
  </script>

  </div>

</body>
</html>
```

Результаты выполнения листинга 6.3 представлены на рис. 6.4.

Вы также можете использовать цикл `for...in` для вывода значений свойств объекта, а не только их имен. В листинге 6.4 приведен пример программы, которая выводит текущие значения каждого из свойств объекта `Document`.

Листинг 6.4. Вывод имен и значений свойств объекта `Document` с помощью цикла `for...in`

```
<html>
<head>
  <title>Имена и значения свойств документа</title>
  <style>
    .columns {
      -webkit-column-count: 6; // Chrome, Safari, Opera
      -moz-column-count: 6; // Firefox
      column-count: 6;
    }
  </style>
</head>
<body>

  <div class="columns">

  <script>
    for (var prop in document){
      document.write (prop + ": " + document[prop] +
          "<br>");
  }
  </script>

  </div>

</body>
</html>
```

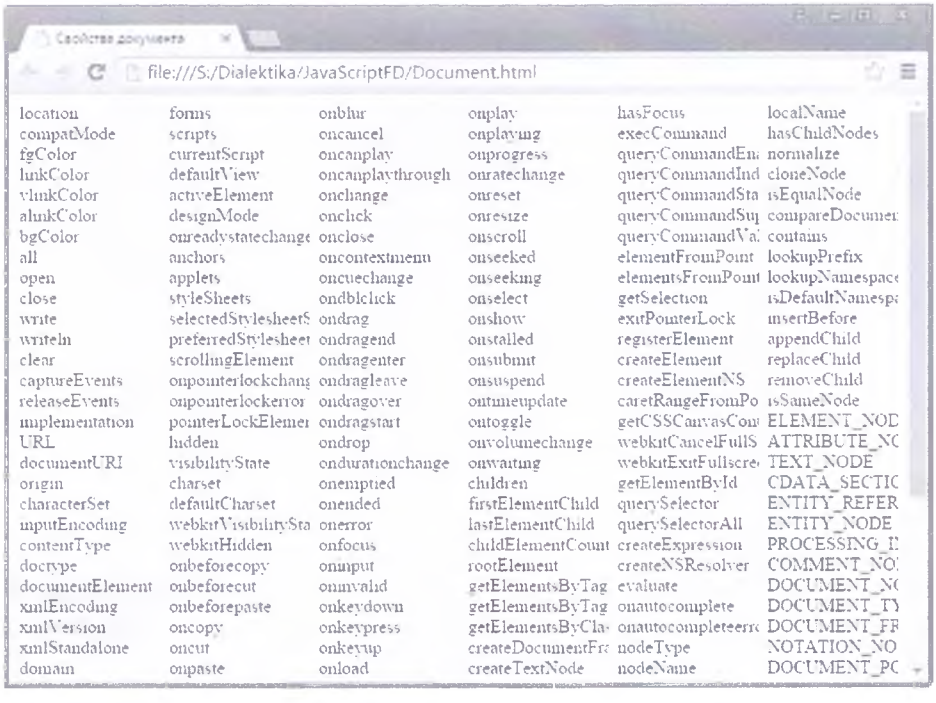


Рис. 6.4. Вывод списка всех свойств объекта `Document` с помощью цикла `for...in`

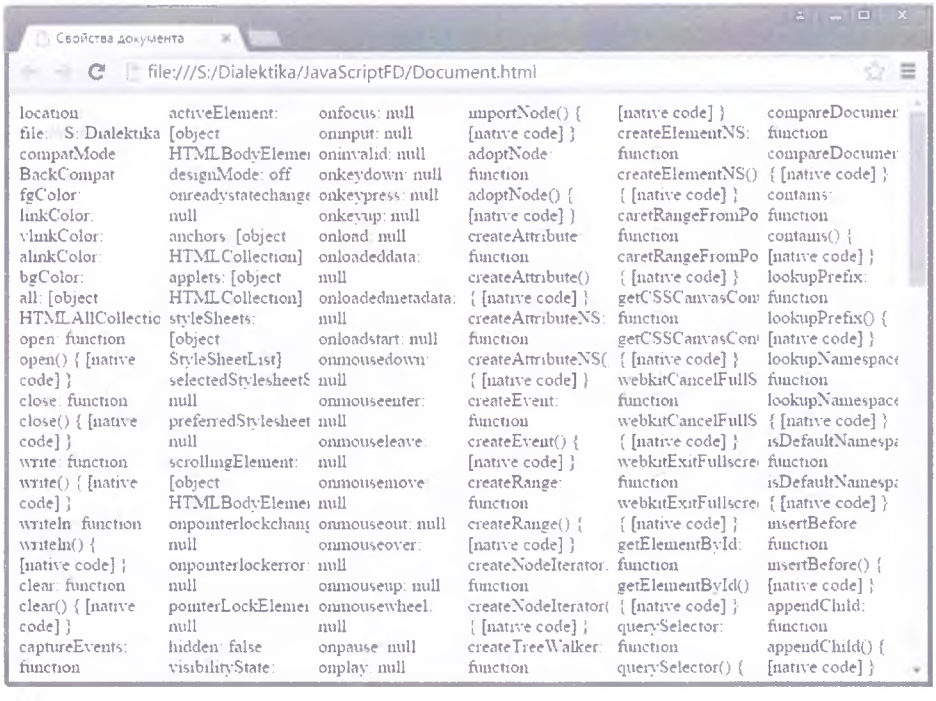


Рис. 6.5. Результаты вывода имен и значений свойств объекта `Document` с помощью цикла `for...in`

Цикл while

Инструкция while создает цикл, который выполняется до тех пор, пока вычисление выражения, устанавливающего условие выполнения цикла, дает истинное значение. В листинге 6.5 приведен пример веб-страницы, содержащей цикл while.

Листинг 6.5. Использование цикла while

```
<html>
<head>
  <title>Игра в слова</title>
</head>
<body>
  <script>
    var guessedWord = prompt("Какое слово я задумал?");
    while (guessedWord != "сэндвич") { // пока не названо слово
                                       // "сэндвич"
      prompt("Нет. Это не оно. Попробуйте еще раз.");
    }
    alert("Поздравляю! Это именно то слово!"); // выполняется после
                                               // выхода из цикла
  </script>
</body>
</html>
```

Цикл do...while

Цикл do...while во многом работает точно так же, как и цикл while, за исключением того, что в данном случае инструкции предшествуют проверке условия. Как следствие, данный вид цикла всегда выполняется по крайней мере один раз.

Использование цикла do...while продемонстрировано в листинге 6.6.

Листинг 6.6. Использование цикла do...while

```
<html>
<head>
  <title>Давайте посчитаем</title>
</head>
<body>
  <script>
    var i = 0;
    do {
      i++;
      document.write(I + "<br>");
    } while (i<10);
  </script>
</body>
</html>
```

break и continue

Инструкции `break` и `continue` прерывают выполнение цикла. Мы уже рассматривали инструкцию `break` в контексте инструкции `switch`, где она использовалась для продолжения программы после выполнения подходящего варианта.

Инструкция `break` в циклах делает то же самое. Она приводит к немедленному выходу из тела цикла, независимо от того, выполняются или не выполняются условия для продолжения выполнения цикла.

Например, представленная в листинге 6.7 игра в слова работает так же, как и та, которая приводилась в листинге 6.5, но работа цикла немедленно прекращается, если значение не было введено.

Листинг 6.7. Использование инструкции `break` в цикле `while`

```
<html>
<head>
  <title>Игра в слова</title>
</head>
<body>
  <script>
    var guessedWord = prompt("Какое слово я задумал?");
    while (guessedWord != "сэндвич") {
      if(guessword == "" {break;} // немедленно выйти из цикла, если
                                  // пользователь не ввел значение
      prompt("Нет. Это не оно. Попробуйте еще раз.");
    }
    alert("Поздравляю! Это именно то слово!");
  </script>
</body>
</html>
```

Когда встречается инструкция `continue`, выполнение текущей итерации прекращается, инструкции цикла, следующие за инструкцией `continue`, игнорируются, и начинается выполнение следующей итерации цикла.

В листинге 6.8 представлена программа, которая ведет счет от 1 до 20, но выводит лишь четные числа. Обратите внимание на то, что для определения четности числа в программе используется оператор деления по модулю (`%`).

Листинг 6.8. Счет и использование инструкции `continue` для отображения четных чисел

```
<html>
<head>
  <title>Счет и отображение четных чисел</title>
</head>
<body>
  <script>
    for (var i = 0; i <= 20; i++){
      if (i%2 != 0){
        continue;
      }
```

```
      document.write (i + " - четное число.<br>");
    }
  </script>
</body>
</html>
```

Если инструкция `continue` используется таким способом, то она может заменять собой инструкцию `else`.

Результат выполнения кода, представленного в листинге 6.8, показан на рис. 6.6.

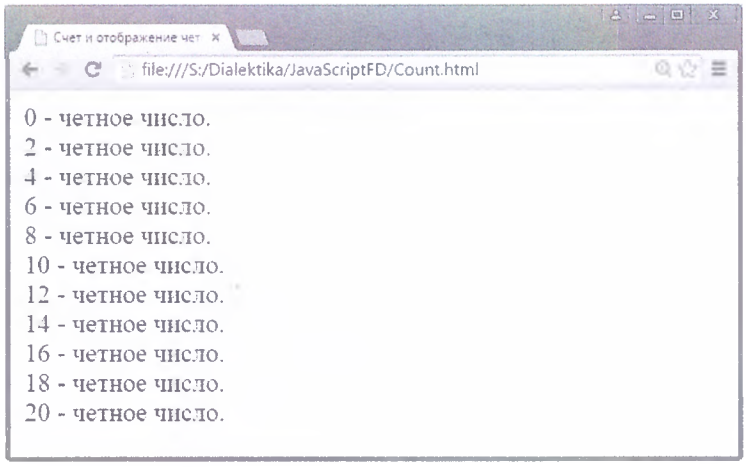


Рис. 6.6. Счет и использование инструкции `continue` для отображения четных чисел

Инструкции `break` и `continue` могут быть не только полезными, но и небезопасными. Они короткие, и их можно легко пропустить при чтении кода. Поэтому некоторые программисты считают их использование в цикле небезопасной практикой. Более подробно о всех сложностях этой проблемы см. дискуссию по следующему адресу:

`http://programmers.stackexchange.com/questions/58237/arebreak-and-continue-bad-programming-practices`

Часть II

Организация программ на JavaScript

В этой части...

- ✓ Работа с функциями
- ✓ Создание и использование объектов

Глава 7

Приобретаем функциональность

В этой главе...

- Создание функций
- Документирование функций
- Передача аргументов
- Возврат значений
- Организация программ с помощью функций

"Для меня писать — естественная функция.
Без этого я бы заболел и умер.
Это такая же моя часть, как печень или кишечник,
и примерно столь же привлекательная".

Чарльз Буковски

Функции помогают снизить повторяемость кода за счет превращения часто встречающихся участков кода в независимые единицы, допускающие многократное использование. В этой главе вы напишете некоторые функции, которые превратят решение трудоемких задач в настоящее развлечение.

Роль функций

Функции — это мини-программы в составе ваших программ. Функции предназначены для выполнения тех конкретных задач в рамках основной программы, потребность в которых может возникать неоднократно.

Если вы успели прочитать какую-либо из предыдущих глав, то уже видели, как работают функции. Ниже приведен пример простой функции, в результате выполнения которой к концу строки добавляется символ `t`.

```
function addt(aString) {
  aString += "t";
  return aString;
}
```

Чтобы проверить работу этой функции, выполните следующие действия.

1. **Откройте консоль JavaScript в браузере Chrome.**
2. **Введите код функции в консоли.**

Можете ввести весь текст в одной строке или же нажимать клавиши <Shift+ +Return> или <Shift+Enter> для создания нескольких строк, прежде чем выполнить код.

3. **После ввода завершающей фигурной скобки нажмите клавишу <Return> или <Enter>.**

   В консоли должен отобразиться текст `undefined`.

4. **Введите следующую команду, а затем нажмите клавишу <Return> или <Enter>, чтобы выполнить функцию:**

```
addt("Я программирую на JavaScrip");
```

Результат выполнения этой функции представлен на рис. 7.1.

Функции — фундаментальные элементы программирования на JavaScript, и их использование регламентируется множеством правил, которые вы как JavaScript-программист должны знать. Не расстраивайтесь, если вам не удастся запомнить сразу все детали работы функций. Некоторые абстрактные концепции станут для вас более понятными лишь после того, как вы накопите практический опыт работы с функциями, и, возможно, вам даже понадобится еще раз прочитать эту главу. В конечном счете для вас все прояснится, так что смело двигайтесь вперед!

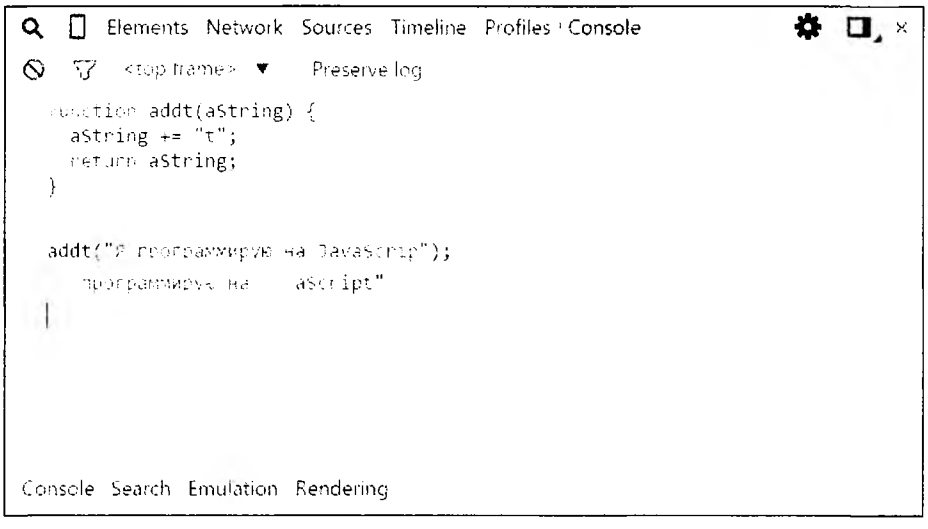


Рис. 7.1. Выполнение вашей первой функции в консоли JavaScript

Терминология функций

Обсуждая функции, программисты используют терминологию, которую вам необходимо знать для того, чтобы понимать, о чем они говорят. Мы интенсивно используем эту терминологию не только в данной главе, но и на протяжении всей книги. Ниже кратко описаны профессиональные термины, с которыми вы будете сталкиваться в процессе работы с функциями.

Определение функции

Объявление функции в коде еще не означает, что она выполняется. Она просто создается и становится доступной для выполнения в более поздний момент времени. Создание функции с той целью, чтобы ее можно было использовать позднее, называется *определением* функции.

Функция объявляется и определяется в программе или на веб-странице только один раз. Если же вы определите одну и ту же функцию более одного раза, то JavaScript не сообщит об ошибке. В подобных случаях используется та версия функция, которая была определена последней.

Примеры объявления функции:

```
var myFunction = new Function() {
};
```

или

```
function myFunction(){
};
```

Заголовок функции

Заголовок функции — это часть определения функции, которая содержит ключевое слово `function`, имя функции и круглые скобки, например:

```
function myFunction()
```

Тело функции

Тело функции состоит из инструкций, заключенных в фигурные скобки.

```
{
  // тело функции
}
```

Вызов функции

Использование функции осуществляется путем ее *вызова*. Вызов функции приводит к выполнению инструкций, составляющих ее тело. Например:

```
myFunction();
```

Определение параметров и передача аргументов

Параметры — это имена, которые вы присваиваете элементам данных, предоставляемым функции при ее вызове. *Аргументы* — это значения, которые вы предоставляете функции. Предоставление аргументов функции при ее вызове (в соответствии с определенными для функции параметрами) программисты называют *передачей аргументов*.

Для определения параметров функции используется следующий синтаксис:

```
function myFunction(параметр) {
```

Для вызова функции с передачей ей аргумента используется следующий синтаксис:

```
myFunction(myArgument)
```

Возврат значения

Функции могут не только принимать входные данные из внешнего мира, но и передавать обратно значения по завершении своего выполнения. Этот процесс называется *возвратом значения*.

Для возврата значения используется ключевое слово `return`, например:

```
return myValue;
```

Преимущества использования функций

В листинге 7.1 представлена программа, которая суммирует числа. Она отлично работает и делает именно то, что предполагается, с помощью цикла `for...in` (см. главу 7).

Листинг 7.1. Программа для сложения чисел с помощью цикла `for...in`

```
<html>
<head>
  <title>Получение суммы чисел</title>
</head>
<body>
  <script>
    var myNumbers = [2,4,2,7];
    var total = 0;
    for (oneNumber in myNumbers){
      total = total + myNumbers[oneNumber];
    }
    document.write(total);
  </script>
</body>
</html>
```

Однако, если бы у нас было несколько наборов чисел, подлежащих сложению, то потребовалось бы написать отдельные циклы для каждого из них.

В листинге 7.2 программа, представленная в листинге 7.1, преобразуется в функцию, и эта функция используется впоследствии для нахождения суммы элементов, относящихся к различным массивам.

Листинг 7.2. Функция для сложения чисел, образующих массив

```
<html>
<head>
  <title>Получение суммы чисел</title>
```

```
</head>
<body>
  <script>
    /**
    *Сложение элементов массива
    *@param {Array.<number>} numbersToAdd
    *@return {Number} sum
    */
    function addNumbers(numbersToAdd) {
      var sum = 0;
      for (oneNumber in numbersToAdd) {
        sum = sum + numbersToAdd[oneNumber];
      }
      return sum;
    }

    var myNumbers = [2,4,2,7];
    var myNumbers2 = [3333,222,111];
    var myNumbers3 = [777,555,777,555];
    var sum1 = addNumbers(myNumbers);
    var sum2 = addNumbers(myNumbers2);
    var sum3 = addNumbers(myNumbers3);

    document.write(sum1 + "<br>");
    document.write(sum2 + "<br>");
    document.write(sum3 + "<br>");

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

Блочные комментарии, предшествующие функции в листинге 7.2, следуют формату, специфицированному системой документирования JSDoc. Комментируя функцию с использованием этого формата, вы не только облегчаете чтение текста программы, но и делаете возможным использование этих комментариев для документирования своих программ. О системе документирования рассказано во врезке "Документирование JavaScript с помощью JSDoc". Более подробно о системе JSDoc можно прочитать на сайте `http://usejsdoc.org`.

Функции обеспечивают значительную экономию времени, усилий и объема кода программы. Поначалу на написание полезной функции у вас может уйти больше времени, чем на написание JavaScript-кода, расположенного вне функций, но в долгосрочной перспективе ваши программы будут лучше организованы, и если написание функций войдет у вас в привычку, то во многих случаях вы избавите себя от ненужной головной боли.

Документирование JavaScript с помощью JSDoc

Считается хорошей практикой всегда документировать JavaScript-код с помощью какой-либо стандартной системы. Наиболее широко используемой системой документирования JavaScript, которая стала стандартом де-факто, является система JSDoc.

Язык JSDoc — это простой язык разметки, которая может внедряться в файлы JavaScript. Текущая, 3-я версия JSDoc основана на системе JavaDoc, применяющейся для документирования кода, написанного на языке Java.

Аннотировав свои JavaScript-файлы с помощью JSDoc, вы сможете использовать генератор документации, например jsdoc-toolkit, для создания HTML-файлов, документирующих код.

Разметка JSDoc окружается специальными тегами блочных (многострочных) комментариев. Единственное различие между разметкой JSDoc и обычными блочными комментариями JavaScript состоит в том, что в первом случае разметка начинается символами `/**` и заканчивается символами `*/`, тогда как в последнем случае вслед за начальной косой черты указывается лишь один символ "звездочка". Дополнительная "звездочка" не препятствует использованию разметки JSDoc в качестве обычного блочного комментария, но при необходимости делает ее частью генерируемой документации.

На приведенном ниже рисунке представлена часть кода фреймворка AngularJS, аннотированного с использованием системы JSDoc.

Различные части и разделы программы документируются с использованием тегов JSDoc. Ниже перечислены наиболее популярные из них.

Тег JSDoc	Описание
@author	Имя разработчика
@constructor	Маркирует функцию как конструктор
@deprecated	Маркирует метод устаревшим и не рекомендуемым
@exception	Описывает исключение, генерируемое методом; синоним тега @throws

Окончание таблицы

Тег JSDoc	Описание
@exports	Указывает член, экспортируемый модулем
@param	Описывает параметр метода
@private	Обозначает, что метод закрытый
@return	Описывает возвращаемое значение; синоним @returns
@returns	Описывает возвращаемое значение; синоним @return
@see	Описывает связь с другим объектом
@this	Задает тип объекта, на который указывает ключевое слово `this` в функции
@throws	Описывает исключение, генерируемое методом
@version	Версия библиотеки

Написание функций

Объявление функции должно записываться в определенном порядке. Оно состоит из следующих элементов, указанных в порядке их следования:

- ✓ ключевое слово `function`;
- ✓ имя функции;
- ✓ круглые скобки, в которых может быть указан один или несколько параметров;
- ✓ пара фигурных скобок, содержащих инструкции.

Иногда единственным назначением функции является вывод сообщения в отображаемой на экране веб-странице. Хорошим примером этого может служить функция, отображающая текущую дату. Приведенная ниже функция выводит текущую дату в окне браузера.

```
function getTheDate(){
  var rightNow = newDate();
  document.write(rightNow.toDateString());
}
```

Чтобы проверить работу этой функции, выполните следующие действия.

1. **Откройте консоль JavaScript в браузере Chrome.**
2. **Введите код функции в консоли.**

   Для создания отдельных строк кода без их выполнения нажимайте клавиши <Shift+Return> (или <Shift+Enter>) после ввода каждой строки.
3. **После ввода закрывающей фигурной скобки нажмите клавишу <Return> (или <Enter>).**

Обратите внимание на то, что при этом ничего не произойдет, просто в консоли отобразится слово undefined, информирующее вас о том, что функция воспринята, но она не вернула никакого значения.

4. **Вызовите функцию, введя ее имя (`getTheDate`), а вслед за ним — пару круглых скобок и точку с запятой:**

```
getTheDate();
```

Функция выведет текущую дату и время в окне браузера, вслед за которыми на консоли отобразится значение undefined, поскольку функция не имеет возвращаемого значения и предназначена лишь для вывода даты и времени в окне браузера.

По умолчанию возвращаемым значением функций является undefined, поэтому с технической точки зрения отображаемое консолью значение undefined является возвращаемым значением.

Возврат значений

В примере, приведенном в предыдущем разделе, мы создали функцию, которая всего лишь отображает строку в окне браузера. После выполнения единственной инструкции `document.write()` больше нечему выполняться, поэтому программа покидает функцию и переходит к инструкции, следующей за вызовом функции.

Закончив свою работу, большинство функций возвращают значение (отличное от undefined). Это значение можно использовать в остальной части программы. В листинге 7.3 приведен пример функции, возвращающей значение. Затем это значение присваивается переменной и выводится в окне консоли.

Листинг 7.3. Возврат значения функцией

```
function getHello(){
  return "Привет!";
}

var helloText = getHello();
console.log (helloText);
```

Как правило, инструкция `return` является последней инструкцией функции. Ее выполнение означает выход из функции. Вы можете использовать эту инструкцию для возврата функцией любого литерального значения (например, строки “Привет!” или числа 3), значения переменной или выражения, а также объекта, массива и даже другой функции (см. листинг 7.4).

Листинг 7.4. Возврат результата вычисления выражения

```
function getCircumference(){
  var radius = 12;
  return 2 * Math.PI * radius;
}

console.log (getCircumference());
```

Передача и использование аргументов

Для того чтобы функция могла выполнять одни и те же действия с различными входными данными, у программиста должна быть возможность передавать эти данные функции. Такая возможность уже была продемонстрирована в листинге 7.2, где для передачи параметров использовались скобки после имени функции в ее объявлении.

Поначалу различие между параметрами и аргументами может не быть для вас очевидным. Вот в чем суть этого различия.

- ✓ Параметры — это имена, которые вы указываете в определении функции.
- ✓ Аргументы — это значения, которые вы передаете функции.

В следующем примере мы определяем для функции `myTacos` два параметра.

```
function myTacos(meat,produce){
...
}
```

При вызове функции данные (аргументы) указываются там, где в определении функции размещаются параметры. Очень важно, чтобы аргументы, передаваемые функции, были указаны в том порядке, в каком соответствующие им параметры следуют в определении функции.

```
myTacos("beef","onions");
```

Значения, переданные функции, становятся значениями локальных переменных в функции и принимают имена определенных для нее параметров.

В листинге 7.5 в функцию закладывается возможность вывода значений обоих аргументов на консоль. Передача аргументов — это все равно что использование инструкции `var` в функции, но только значения поступают извне функции.

Листинг 7.5. Ссылка на аргументы в функции через имена параметров

```
function myTacos(meat,produce){
  console.log(meat); // вывести "beef"
  console.log(produce); // вывести "onions"
}

myTacos("beef","onions");
```

В определении функции разрешается задавать до 255 параметров. Однако вероятность того, что у кого-то возникнет реальная потребность в функции с таким большим количеством параметров, чрезвычайно мала! Если вдруг обнаруживается, что вам нужна функция, число параметров в которой очень велико, то всегда имеет смысл подыскать лучший способ организации программы в интересах сохранения ясности и удобочитаемости кода.

Передача аргументов по значению

Если для передачи аргумента используется переменная, относящаяся к одному из примитивных (элементарных) типов данных, то говорят, что аргумент передается *по значению*. Это означает, что новая переменная, созданная для хранения значения в функции, полностью отделена от переменной, используемой для передачи аргумента, и что бы ни происходило с переданным значением после того, как оно окажется в функции, переменная снаружи функции не почувствует никаких изменений.

В JavaScript элементарными считаются следующие типы данных: `string`, `number`, `boolean`, `undefined` и `null`.

Проиллюстрируем это на конкретном примере. В листинге 7.6 создаются две переменные, которым присваиваются значения, после чего эти переменные передаются функции. В данном случае имена параметров функции совпадают с именами переменных, используемых для передачи аргументов. Несмотря на то что значения переменных в функции подвергаются изменению, значения исходных переменных остаются прежними.

Листинг 7.6. Демонстрация передачи аргументов по значению

```
<html>
<head>
  <title>Передача аргументов по значению</title>
</head>
<body>
  <script>
    /**
    * Инкрементирование двух значений
    * @param {number} number1
    * @param {number} number2
    */
    function addToMyNumbers(number1,number2){
      number1++;
      number2++;
      console.log("Значение number 1 в функции: " + number1);
      console.log("Значение number 2 в функции: " + number2);
    }

    var number1 = 3;
    var number2 = 12;

    addToMyNumbers(number1,number2); // передача двух аргументов

    console.log("Исходное значение number1: " + number1);
    console.log("Исходное значение number2: " + number2);
  </script>
</body>
</html>
```

На рис. 7.2 показан результат работы этой программы в консоли JavaScript.

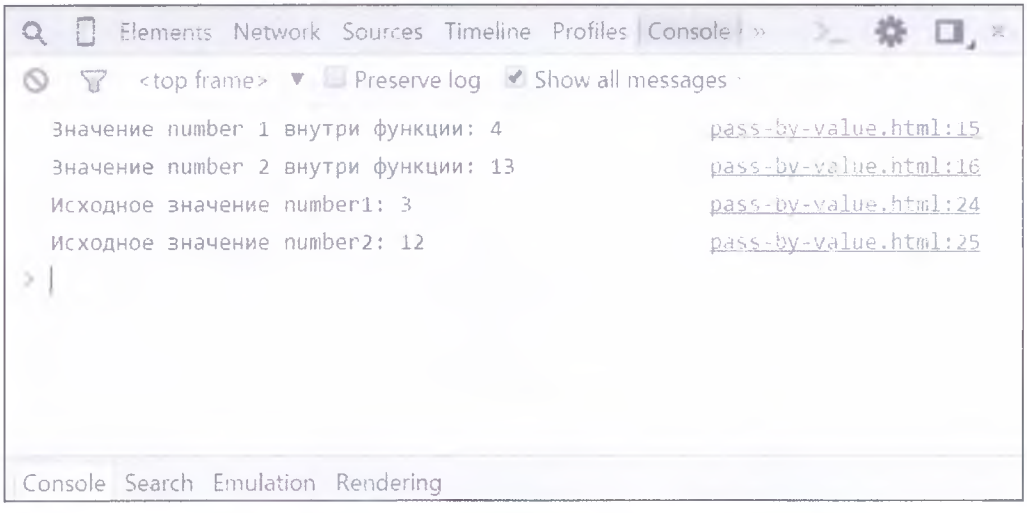


Рис. 7.2. Изменение переменных в функции не сказывается на внешних переменных

Передача аргументов по ссылке

В то время как переменные JavaScript, принадлежащие к элементарным типам (строки, числа, булевы значения, `undefined` и `null`), передаются функциям по значению, объекты JavaScript передаются *по ссылке*. Это означает, что в тех случаях, когда функции в качестве аргумента передается объект, любые изменения этого объекта, внесенные в функции, влекут за собой изменения значений вне функции. Вопросы передачи аргументов по ссылке выходят за рамки данной главы и будут рассмотрены в главе 8.

Вызов функции с неполным числом аргументов

Количество аргументов, задаваемых при вызове функции, не обязано совпадать с количеством параметров, указанных в определении этой функции. Если в определении функции содержатся три параметра, но вы вызываете ее, задавая всего лишь два аргумента, то третий параметр создаст в функции переменную, имеющую значение `undefined`.

Аргументы по умолчанию

Если желательно, чтобы значения аргументов по умолчанию были отличными от `undefined`, то установите эти значения по своему усмотрению. Наиболее широко поддерживаемый и общепринятый способ реализации этого заключается в проверке значений аргументов в функции и установке значений по умолчанию, если типом данных аргумента является `undefined`.

Например, в листинге 7.7 функция принимает один параметр. В теле функции проверяется, равно ли значение аргумента `undefined`. Если это так, то устанавливается значение по умолчанию.

Листинг 7.7. Задание значений по умолчанию для аргументов

```
function welcome(yourName){
  if (typeof yourName === 'undefined'){
    yourName = "друг";
  }
```

В следующей версии JavaScript, которая называется ECMAScript 6, будет предоставлена возможность устанавливать для параметров значения по умолчанию в заголовке функции (листинг 7.8).

Листинг 7.8. Задание значений по умолчанию для аргументов в заголовке функции

```
function welcome(yourName = "друг") {
  document.write("Привет," + yourName);
}
```

К моменту публикации книги еще не все браузеры поддерживали такую возможность, и поэтому данный метод установки значений аргументов по умолчанию может работать не для всех пользователей вашей программы. Таким образом, в настоящее время лучше ограничиться методом, совместимым с прежними версиями JavaScript, который использован в листинге 7.7.

Вызов функции с количеством аргументов, превышающим количество параметров

Если при вызове функции количество аргументов превышает количество ее параметров, то для дополнительных аргументов локальные переменные не создаются, поскольку функции неизвестно, как к ним обращаться.

И все же существует один трюк, используя который вы сможете получить значения аргументов даже в том случае, если соответствующие им параметры отсутствуют. Это делается с помощью объекта `arguments`.

Получение значений аргументов с помощью объекта `arguments`

Если вы не знаете заранее, сколько аргументов будет передано в функцию, то для получения значений аргументов воспользуйтесь объектом `arguments`, который JavaScript автоматически встраивает в функции.

Объект `arguments` содержит массив всех аргументов, переданных функции. Осуществив в цикле перебор всех элементов массива (с помощью цикла `for` или `for... in`), вы сможете использовать все аргументы, даже если их количество может меняться от вызова к вызову.

В листинге 7.9 демонстрируется применение объекта `arguments` для вывода приветственного сообщения, которое может адресоваться как лицам с одним вторым именем, так и лицам с двумя вторыми именами.

Листинг 7.9. Использование объекта `arguments` для определения функции, выводящей приветственное сообщение

```
<html>
<head>
  <title>Приветственное сообщение</title>
</head>
<body>
  <script>
    /**
    *Гибкое приветственное сообщение
    */
    function flexibleWelcome(){
      var welcome = "Добро пожаловать,";
      for (i = 0; I < arguments.length; i++) {
        welcome = welcome + arguments[i] + "";
      }
      return welcome;
      }
    document.write(flexibleWelcome("Christopher" ,
          "James" , "Phoenix" , "Minnick") + "<br>");
    document. write(flexibleWelcome("Eva" , "Ann" ,
          "Holland") + "<br>");
  </script>
</body>
</html>
```

Область видимости функции

Переменные, созданные в функции путем передачи ей аргументов или использования ключевого слова `var`, доступны только в функции. Программисты называют это свойство JavaScript *областью видимости функции*. После выхода из функции созданные в ней переменные уничтожаются.

В то же время переменная, созданная в функции без использования ключевого слова `var`, становится *глобальной* переменной. Доступ к такой переменной и ее изменение возможны в любом месте программы.

Случайное создание глобальной переменной является распространенным источником ошибок и дефектов в JavaScript, поэтому следует всегда внимательно следить за ограничением области видимости переменных и не допускать создания глобальных переменных, если только в этом нет острой необходимости.

Анонимные функции

В заголовке функции имя не является обязательной частью, так что вы можете создавать функции, не имеющие имени. Поначалу это может казаться странным, поскольку такая функция подобна собаке без клички, которую невозможно позвать! Однако *анонимные* функции могут назначаться переменным при их создании, что оставляет вам те же возможности, как и в том случае, когда имя функции указывается в ее заголовке.

```
var doTheThing = function(thingToDo) {
  document.write("Сделать следующее: " + thingToDo);
}
```

Различия между анонимными и именованными функциями

Между созданием именованной функции и присвоением анонимной функции переменной имеются важные и иногда полезные различия. Первое из них состоит в том, что анонимные функции, присвоенные переменной, существуют и могут вызываться лишь после выполнения операции присваивания. Доступ же к именованным функциям возможен в любом месте программы.

Суть второго различия заключается в том, что в случае анонимных функций значение переменной может быть в любой момент изменено, и ей может быть присвоена другая функция. Это делает механизм анонимных функций более гибким по сравнению с именованными функциями.

Самовыполняющиеся анонимные функции

Другим полезным применением анонимных функций являются *самовыполняющиеся* функции. Самовыполняющаяся функция — это функция, которая выполняется сразу же после ее создания.

Чтобы превратить обычную анонимную функцию в самовыполняющуюся, достаточно заключить ее в круглые скобки, а затем добавить пару круглых скобок и точку с запятой после них.

Преимуществом использования самовыполняющихся анонимных функций является то, что переменные, которые вы создаете в них, уничтожаются при выходе из функции. Благодаря этому вы избегаете конфликта имен переменных и предотвращаете утечку памяти, т.е. сохранение в памяти переменных, которые больше не используются. Пример определения и использования самовыполняющейся анонимной функции приведен в листинге 7.10.

Листинг 7.10. Использование самовыполняющейся анонимной функции

```
var myVariable = "Я нахожусь вне функции.";
(function() {
  var myVariable = "Я нахожусь в этой анонимной функции";
  document.write(myVariable);
})();
document.write(myVariable);
```

Разработчики веб-приложений регулярно используют анонимные функции для создания самых разнообразных эффектов на веб-страницах. Более подробно об анонимных функциях можно прочитать в главах 15 и 16.

Сделайте это снова с помощью рекурсии

Функции можно вызывать извне функций или в других функциях. Возможен даже вызов функции из самой себя. На языке программистов такой вызов называется *рекурсией.*

Во многих ситуациях рекурсия эквивалентна циклу с той лишь особенностью, что повторяются инструкции, образующие тело функции. В листинге 7.11 приведен пример простой рекурсивной функции. Однако в этой рекурсивной функции имеется одна большая проблема. Видите ли вы ее?

Листинг 7.11. Рекурсивная функция, содержащая фатальную ошибку

```
function squareItUp(startingNumber) {
  var square = startingNumber * startingNumber;
  console.log(square);
  squareItUp(square);
}
```

Вы поняли, в чем суть проблемы? Выполнение этой функции никогда не завершится. Она будет бесконечно долго умножать числа, пока вы ее не остановите.

Выполнение этой функции, вероятнее всего, приведет к аварийной остановке браузера. Разумеется, никакой вред браузеру нанесен не будет, но этого будет достаточно для того, чтобы вы внимательно просмотрели код и обнаружили в нем проблему.

В листинге 7.12 представлен другой вариант функции `squareItUp()`, усовершенствованный за счет предоставления так называемого *базового случая.* Базовый случай — это условие, выполнение которого означает прерывание рекурсивного процесса и прекращение работы функции. Базовый случай должен предусматриваться для каждой рекурсивной функции.

Листинг 7.12. Рекурсивная функция, вычисляющая квадраты чисел до тех пор, пока число не станет большим 1 000 000

```
function squareItUp(startingNumber) {
  square = startingNumber * startingNumber;
  if (square > 1000000) {
    console.log(square);
  } else {
    squareItUp(square);
  }
}
```

Вот это уже лучше! Но проблемы остались и у этой функции. Что если кто-то передаст функции отрицательное число, нуль или единицу? Любое из этих значений

породит бесконечный цикл. Чтобы обезопасить себя в подобных ситуациях, необходимо ввести условие прерывания рекурсии. В листинге 7.13 дополнительно введена проверка того, что аргумент не меньше и не равен 1 и что он представляет собой число. В любом из этих случаев выполнение функции немедленно прекращается.

Листинг 7.13. Рекурсивная функция с заданными условиями для прекращения выполнения и базовым случаем

```
function squareItUp(startingNumber) {

  // Условия прекращения выполнения, неверный ввод
  if ((typeof startingNumber != 'number') ||
          (startingNumber <= 1)) {
    return - 1; // выйти из функции
  }

  square = staringNumber * startingNumber;

  //Базовый случай
  if (square > 1000000) {
      console.log(square); // Вывести конечный результат
    } else { // Если базовое условие не выполняется,
             // то выполнить снова
      squareItUp(square);
  }
}
```

Функции, объявленные в других функциях

Функции могут объявляться в других функциях. В листинге 7.14 продемонстрировано, как работает эта техника и как она влияет на область видимости переменных, созданных в функциях.

Листинг 7.14. Объявление функций в функциях

```
function turnIntoAMartian(myName) {

  function recallName(myName) {
    var martianName = myName + " Марсианин";
  }
  recallName(myName);
  console.log(martianName); // вернуть undefined

}
```

Предыдущий пример показывает, как вложение функции в другую функцию создает другой уровень области видимости. К переменным, созданным во внутренней функции, невозможен непосредственный доступ извне. Для получения их значений требуется использование инструкции `return`, как показано в листинге 7.15.

Листинг 7.15. Возврат значений из внутренней функции

```
function turnIntoAMartian(myName) {

  function recallName(myName) {
    var martianName = myName + " Марсианин";
    return martianName;
  }
  var martianName = recallName(myName);
  console.log(martianName);
}
```

Глава 8

Создание и использование объектов

В этой главе...

- Понятие объекта
- Свойства и методы объектов
- Создание объектов
- Точечная нотация
- Работа с объектами

"С вещью без имени нам нечего делать".

Морис Бланшо

В этой главе показано, зачем нужны объекты, как их используют и почему их применение способствует улучшению стиля программирования и качества программ.

Объект моих желаний

Кроме пяти элементарных типов данных (см. главу 3), в JavaScript имеется тип данных под названием *объект*. Объекты JavaScript инкапсулируют данные и функциональность в повторно используемых компонентах.

Чтобы понять, что такое объекты и как они работают, полезно сравнить объекты JavaScript с физическими объектами из реальной жизни. Возьмем, к примеру, гитару.

Мы можем описать гитару, указав, что она собой представляет и что с ее помощью можно делать. Вот некоторые факты о гитаре, которые будут использованы в этом примере:

- ✓ она имеет шесть струн;
- ✓ она черно-белая:
- ✓ она электрическая;
- ✓ она имеет цельный корпус.

А вот кое-что из того, что можно делать с помощью гитары:

- ✓ заставлять струны звучать;
- ✓ увеличивать громкость;

- ✓ уменьшать громкость;
- ✓ натягивать струны;
- ✓ регулировать тембр;
- ✓ ослаблять струны.

Если бы гитара была объектом JavaScript, а не реальным объектом, то действия, которые с ней можно совершать, назывались бы *методами*, а элементы, которые образуют гитару, например струны или корпус, назывались бы ее *свойствами*.

Методы и свойства объекта записываются одним и тем же способом — в виде пар "имя–значение", в которых имя и значение разделены двоеточием. Если значением свойства является функция, то оно называется методом.

В действительности все, что находится внутри объекта, является свойством. Просто свойство, значением которого является функция, мы называем иначе — методом.

В листинге 8.1 показано, как могла бы выглядеть гитара, представленная в виде объекта JavaScript.

Листинг 8.1. Гитара как объект JavaScript

```
var guitar = {
  bodyColor: "black",
  scratchPlateColor: "white",
  numberOfStrings: 6,
  brand: "Yamaha",
  bodyType: "solid",
  strum: function() {...},
  tune: function() {...}
```

Создание объектов

В JavaScript объекты можно создавать двумя способами:

- ✓ посредством объектного литерала;
- ✓ с помощью метода, называемого конструктором.

Какой способ лучше использовать, зависит от конкретных обстоятельств. В следующем разделе вы узнаете обо всех "за" и "против" каждого из этих подходов, а также о том, при каких условиях один из них предпочтительнее другого.

Определение объектов с помощью объектных литералов

Создание объекта с помощью объектного литерала начинается с определения обычной переменной посредством ключевого слова `var`, за которым следует оператор присваивания:

```
var person =
```

В правой части этой инструкции записывается пара фигурных скобок, в которые заключаются пары "имя–значение", разделенные запятыми:

```
var person = {eyes: 2, feet: 2, hands: 2, eyeColor: "blue"};
```

Если в момент создания объекта вы еще не знаете, какие свойства он будет иметь, или впоследствии вашей программе понадобится, чтобы он имел дополнительные свойства, то создайте объект с каким-то начальным набором свойств или вообще без таковых, а нужные свойства добавьте позднее.

```
var person = {};
person.eyes = 2;
person.hair = "brown";
```

Так называемая *точечная нотация*, которую вы здесь видите, вам должна быть знакома, поскольку в предыдущих примерах для вывода текста неоднократно применялись методы `document.write()` и `console.log()`. Точка между именем объекта и свойством указывает на принадлежность свойства данному объекту. Более подробно точечная нотация обсуждается в разделе "Получение и установка значений свойств объектов".

Важной особенностью объектов является то, что, как и в случае массивов, значения их свойств могут относиться к разным типам данных.

Не столь уж большим секретом, о котором вам надо знать, является то, что массивы и функции JavaScript — это объекты, и что элементарные типы, такие как числа, строки и булевы значения, также могут использоваться как объекты. Для вас это означает, что все они имеют свойства, и этим свойствам можно присваивать значения, как и в случае любого другого объекта.

Определение объектов с помощью конструктора `Object()`

Второй способ создания объектов связан с использованием конструктора `Object()`. При этом сначала используется выражение `new Object()`, а затем определяются и инициализируются свойства полученного объекта. Пример использования конструктора `Object()` приведен в листинге 8.2.

Листинг 8.2. Использование конструктора `Object()`

```
var person = new Object();
person.feet = 2;
person.name = "Sandy";
person.hair = "black";
```

Создание объектов с помощью конструктора `Object()` вполне приемлемо, однако считается, что такой подход менее предпочтителен по сравнению с созданием объектов с помощью объектных литералов. Основанием для этого служат следующие соображения:

- ✓ использование конструктора требует ввода большего количества символов по сравнению с объектными литералами;
- ✓ использование конструктора несколько снижает производительность кода в браузере;
- ✓ результирующий код читать труднее, чем в случае использования объектных литералов.

Получение и установка свойств объектов

После создания объекта и определения его свойств необходимо иметь возможность получать и устанавливать значения этих свойств. Возможны два способа доступа к свойствам объектов: с использованием точечной или скобочной нотации.

Точечная нотация

В соответствии с точечной нотацией после имени объекта ставится точка, а за ней имя свойства, значение которого вы хотите получить или установить.

Например, чтобы создать новое свойство `firstName` объекта `person` или изменить значение уже существующего свойства, используйте инструкцию наподобие следующей:

```
person.firstName = "Glenn";
```

Если до этого свойство `firstName` отсутствовало в объекте, то данная инструкция создаст его. Если же это свойство уже существует, то оно будет обновлено новым значением.

Для извлечения значения свойства с помощью точечной нотации используется в точности тот же синтаксис, но объект вместе с именем свойства (так называемый *получатель свойства*) перемещается в правую часть инструкции. Например, если требуется конкатенировать значения `person.firstName` и `person.lastName` и присвоить результат новой переменной `fullName`, то можно использовать следующую инструкцию:

```
var fullname = person.firstName + person.lastName;
```

Если же вы хотите записать значение `person.firstName` в HTML-документ, то достаточно использовать получатель свойства так, как если бы это была обычная переменная:

```
document.write (person.firstName);
```

Как правило, для установки и получения значений свойств используют точечную нотацию, поскольку она требует меньшего объема ввода и легче читается.

Скобочная нотация

Как вы, наверное, уже догадались, скобочная нотация предполагает использование скобок, в данном случае квадратных. Чтобы установить с ее помощью значение свойства, следует записать имя свойства, заключенное в кавычки, в квадратных скобках:

```
person["firstName"] = "Iggy";
```

Скобочная нотация предлагает возможности, которые точечная нотация не в состоянии обеспечить. Что наиболее важно, в квадратных скобках можно использовать переменные в тех случаях, когда при написании программы имя свойства не известно заранее.

В следующем примере делается точно то же, что и в предыдущем, но вместо строкового литерала мы помещаем в квадратные скобки переменную. Используя эту технику, вы можете создать единственную инструкцию, которая сможет функционировать во множестве различных ситуаций, например в циклах или функциях.

```
var personProperty = "firstName";

person[personProperty] = "Iggy";
```

В листинге 8.3 приведена простая программа, которая создает объект `chair`, а затем перебирает в цикле все его свойства и запрашивает для каждого из них ввод значения пользователем. По завершении ввода всех свойств вызывается функция `writeChairReceipt()`, которая выводит название каждого свойства и его значение, заданное пользователем.

Листинг 8.3. Сценарий конфигурирования объекта `Chair`

```
<html>
<head>
  <title>Что это? Конфигуратор объекта Chair</title>
</head>
<body>
  <script>
    var myChair = {
      "cushionMaterial" : "",
      "numberOfLegs" : "",
      "legHeight" : ""
    };

    function configureChair() {
      var userValue;
      for (var property in myChair) {
        if (myChair.hasOwnProperty(property)) {
          userValue = prompt("Введите значение свойства " +
            property);
          myChair[property] = userValue;
        }
      }
    }
```

```
    function writeChairReceipt() {
      document.write("<h2>Объект chair будет иметь следующую
              конфигурацию:</h2>");
      for (var property in myChair) {
        if (myChair.hasOwnProperty(property)) {
          document.write(property + ": " + myChair[property]
            + "<br>");
        }
      }
    }

    configureChair();
    writeChairReceipt();
  </script>
</body>
</html>
```

Удаление свойств

Для удаления свойств из объектов используется оператор `delete`. В листинге 8.4 показано, как это работает.

Листинг 8.4. Использование оператора `delete`

```
var myObject = {
  var1 : "the value",
  var2 : "another value",
  var3 : "yet another"
};

// удалить свойство var2 из объекта myObject
delete myObject.var2;

// попытка записи значения var2
document.write(myObject.var2); // ошибка
```

Работа с методами

Методы — это свойства, значениями которых являются функции. Метод определяется точно так же, как и любая функция. Единственное различие состоит в том, что метод присваивается свойству объекта. В листинге 8.5 продемонстрировано создание объекта с несколькими свойствами, одним из которых является метод.

Листинг 8.5. Создание метода

```
var sandwich = {
meat:"",
cheese:"",
bread:"",
condiment:"",
makeSandwich: function (meat,cheese,bread,condiment) {
```

```
  sandwich.meat = meat;
  sandwich.cheese = cheese;
  sandwich.bread = bread;
  sandwich.condiment = condiment;
  var mySandwich = sandwich.bread + ", " + sandwich.meat +
           ", " + sandwich.cheese + ", " + sandwich.
           condiment;
  return mySandwich;
}}
```

Можно использовать метод `makeSandwich` объекта `sandwich`, прибегнув к точечной нотации, как это обычно делается при доступе к свойству, предоставляя параметры в круглых скобках, записанных после имени метода (листинг 8.6).

Листинг 8.6. Вызов метода

```
<html>
<head>
  <title>Сделайте мне сэндвич</title>
</head>
<body>
  <script>

    var sandwich = {
    meat:"",
    cheese:"",
    bread:"",
    condiment:"",
    makeSandwich: function (meat,cheese,bread,condiment) {
      sandwich.meat = meat;
      sandwich.cheese = cheese;
      sandwich.bread = bread;
      sandwich.condiment = condiment;
      var mySandwich = sandwich.bread +
        ", " + sandwich.meat + ", " +
        sandwich.cheese + ", " +
        sandwich.condiment;
      return mySandwich;
      }
    }

    var sandwichOrder =
            sandwich.makeSandwich("ham","cheddar","wheat","
            spicy mustard");
    document.write (sandwichOrder);

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

Использование ключевого слова this

Ключевое слово this — это краткая запись ссылки на объект, содержащий данный метод. Например, в листинге 8.7 вместо имени объекта sandwich везде используется ключевое слово this. Когда функция makeSandwich() вызывается как метод объекта sandwich, JavaScript понимает, что this означает объект sandwich.

Листинг 8.7. Использование ключевого слова this в методе

```
<html>
<head>
  <title>Сделайте мне сэндвич</title>
</head>
<body>
  <script>

    var sandwich = {
      meat:"",
      cheese:"",
      bread:"",
      condiment:"",
      makeSandwich: function(meat,cheese,bread,condiment){
        this.meat = meat;
        this.cheese = cheese;
        this.bread = bread;
        this.condiment = condiment;
        var mySandwich = this.bread + ", " + this.meat + ",
            " + this.cheese + ", " + this.condiment;
        return mySandwich;
        }
      }

      var sandwichOrder =
            sandwich.makeSandwich("ham","cheddar","wheat","
            spicy mustard");
      document.write (sandwichOrder);

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

Используя указатель this вместо имени конкретного объекта, мы получим точно тот же результат, что и прежде.

Указатель this особенно полезен тогда, когда у вас есть функция, которая может применяться множеством различных объектов. В таком случае ключевое слово this не связывается с каким-то одним объектом и будет указывать на тот объект, который вызывает данный метод.

В следующих разделах вы познакомитесь с функциями-конструкторами и наследованием, причем в обоих случаях ключевое слово this играет важную роль.

Объектно-ориентированный способ разбогатеть: наследование

Ваши возможности не ограничены созданием лишь конкретных объектов, которые моделируют, например, гитару, автомобиль или сэндвич. Вся красота объектов состоит в том, что с их помощью можно создавать объектные типы, которые впоследствии могут быть использованы для создания других объектов.

В предыдущих разделах объекты конструировались непосредственно на основе объектного типа `Object`.

Особенно отчетливо это утверждение демонстрируется примерами в разделе "Создание объектов", в которых объекты создавались с помощью конструктора:

```
var person = new Object();
```

В этой строке создается объект типа `Object`. Вновь созданный объект `person` содержит все свойства и методы объектного типа `Object`, заданные по умолчанию, но они связаны с новым именем. Далее вы сможете самостоятельно добавить в объект `person` методы и свойства, конкретизирующие данный объект.

```
var person = new Object();
person.eyes = 2;
person.ears = 2;
person.arms = 2;
person.hands = 2;
person.feet = 2;
person.legs = 2;
person.species = "Homo sapiens";
```

Теперь объект `person` обладает конкретными свойствами. Представьте, что вы хотите создать новый объект, описывающий конкретного человека, например кантри-музыканта Вилли Нельсона. Вы могли бы просто создать новый объект `willieNelson` и наделить его теми же свойствами, которые имеются у объекта `person`, и дополнительно свойствами, делающими Вилли Нельсона уникальной личностью.

```
var willieNelson = new Object();
willieNelson.eyes = 2;
willieNelson.ears = 2;
willieNelson.arms = 2;
willieNelson.hands = 2;
willieNelson.feet = 2;
willieNelson.legs = 2;
willieNelson.species = "Homo sapiens";
willieNelson.occupation = "musician";
willieNelson.hometown = "Austin";
willieNelson.hair = "Long";
willieNelson.genre = "country";
```

Однако такой способ определения объекта `willieNelson` слишком расточителен. Он требует выполнения большого объема работы, и к тому же из этого вовсе не

следует, что он связан с тем же типом объектов, что и объект person. Можно лишь сказать, что у них есть некоторые общие свойства.

Решение состоит в том, чтобы создать новый тип объекта под названием Person, а затем создать объект willieNelson, принадлежащий к этому типу.

Обратите внимание на то, что, говоря о типе объекта, мы всегда используем прописную первую букву в имени объектного типа. Это не требование, но почти повсеместно используемое соглашение. Поэтому, например, мы записываем инструкции в следующем виде:

```
var person = new Object();
```

или

```
var willieNelson = new Person();
```

Создание объектов с помощью конструкторов

Чтобы создать новый тип объекта, вы создаете новую функцию конструктора. Конструкторы объектов создаются так же, как и любая другая функция JavaScript, но для присвоения свойств новому объекту используется ключевое слово this. Новый объект наследует свойства объектного типа.

Вот так выглядит конструктор объектов типа Person.

```
function Person(){
  this.eyes = 2;
  this.ears = 2;
  this.arms = 2;
  this.hands = 2;
  this.feet = 2;
  this.legs = 2;
  this.species = "Homo sapiens";
}
```

Теперь для создания нового объекта достаточно назначить функцию новой переменной, например:

```
var willieNelson = new Person();
```

Объект willieNelson наследует свойства объектного типа Person. И хотя вы не предпринимали никаких действий для создания свойств объекта willieNelson, он содержит все свойства объекта Person.

Чтобы проверить, как все это работает, выполните в браузере код, приведенный в листинге 8.8.

Листинг 8.8. Тестирование наследования

```
<html>
<head>
  <title>Демонстрация наследования</title>
</head>
```

```
<body>
  <script>

    function Person(){
      this.eyes = 2;
      this.ears = 2;
      this.arms = 2;
      this.hands = 2;
      this.feet = 2;
      this.legs = 2;
      this.species = "Homo sapiens";
    }
    var willieNelson = new Person();
    alert("Willie Nelson has " + willieNelson.feet + "
            feet!");
  </script>
</body>
</html>
```

Результат выполнения листинга 8.8 представлен на рис. 8.1.

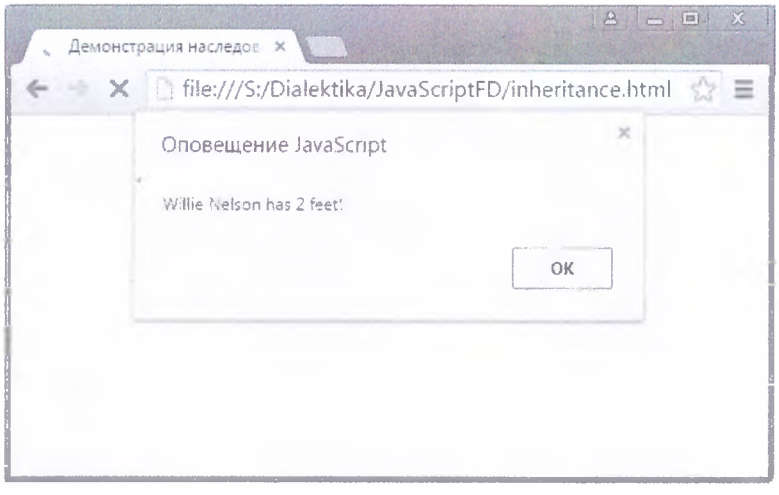


Рис. 8.1. Выполнение вашей первой функции в консоли JavaScript

Видоизменение объектного типа

Предположим, у вас есть объектный тип `Person`, который служит прототипом для нескольких объектов. В один прекрасный момент вы осознаете, что объект `person`, а также все другие объекты, которые являются его потомками, нуждаются в нескольких дополнительных свойствах.

Для изменения объектного прототипа используют свойство `prototype`, которое наследуется каждым объектом от объекта `Object`. В листинге 8.9 показано, как это работает.

Листинг 8.9. Видоизменение прототипа объекта

```
function Person(){
  this.eyes = 2;
  this.ears = 2;
  this.arms = 2;
  this.hands = 2;
  this.feet = 2;
  this.legs = 2;
  this.species = "Homo sapiens";
}

var willieNelson = new Person();
var johnnyCash = new Person();
var patsyCline = new Person();

// Объект Person нуждается в дополнительных свойствах!
Person.prototype.knees = 2;
Person.prototype.toes = 10;
Person.prototype.elbows = 2;

// Проверка наличия новых свойств у существующих объектов
document.write (patsyCline.toes); // выводит 10
```

Создание объектов с помощью метода `Object.create()`

Еще один способ создания одних объектов на основе других заключается в использовании метода `Object.create()`. Преимуществом такого подхода является то, что он не требует написания функции-конструктора и лишь копирует свойства заданного объекта в другой объект. Если один объект наследует от другого, то объект, от которого наследуются свойства, называется *прототипом*.

В листинге 8.10 показано, как метод `Object.create()` может быть использован для создания объекта `willieNelson` на основе прототипа.

Листинг 8.10. Использование метода `Object.create()` для создания объекта на основе прототипа

```
// создать обобщенный тип Person
var Person = {
  eyes: 2,
  arms: 2,
  feet: 2
}

// создать объект willieNelson на основе Person
var willieNelson = Object.create(Person);
// проверка наличия унаследованных свойств
document.write (willieNelson.feet); // выводит 2
```

Часть III

JavaScript в Интернете

В этой части...

- ✓ Управление браузером с помощью объекта `Window`
- ✓ Манипулирование документами посредством DOM
- ✓ Работа с событиями в JavaScript
- ✓ Интеграция ввода и вывода
- ✓ Работа с CSS и графикой

Глава 9

Управление браузером с помощью объекта Window

В этой главе...

- Что такое BOM (Browser Object Model)
- Открытие и закрытие окон
- Получение свойств
- Изменение размеров окон

"Создавая теории, всегда держите распахнутым окно, чтобы было куда выбрасывать теории, оказавшиеся ненужными".

Бела Лугоши

Объектная модель браузера (Browser Object Model — BOM) позволяет JavaScript взаимодействовать с функциональностью браузера. Используя BOM, можно создавать окна и изменять их размеры, отображать окна сообщений и изменять текущую веб-страницу, открытую в браузере.

В этой главе вы узнаете о том, какие операции можно выполнять с окном браузера и как использовать эти возможности для улучшения JavaScript-программ.

Браузерная среда

Браузер — это сложная совокупность программных компонентов, обеспечивающая просмотр веб-страниц. Когда все идет нормально, браузер безукоризненно выполняет свои функции, поддерживая комфортные рабочие условия для пользователя. Однако, как всем нам хорошо известно, время от времени браузеры "тормозят", а иногда их выполнение завершается аварийной остановкой. Чтобы понимать, почему так происходит, и знать, как сделать работу браузера более эффективной, важно иметь отчетливое представление о том, для чего предназначены те или иные его компоненты и каковы механизмы их взаимодействия.

Пользовательский интерфейс

Та часть браузера, с которой вы взаимодействуете, когда вводите URL-адрес, щелкаете на кнопке перехода на начальную страницу, создаете и используете закладки или изменяете настройки, называется пользовательским интерфейсом.

Пользовательский интерфейс браузера включает меню, панели инструментов, боковую панель, полосы прокрутки и кнопки, расположенные снаружи основного окна содержимого, в которое загружаются веб-страницы (рис. 9.1).

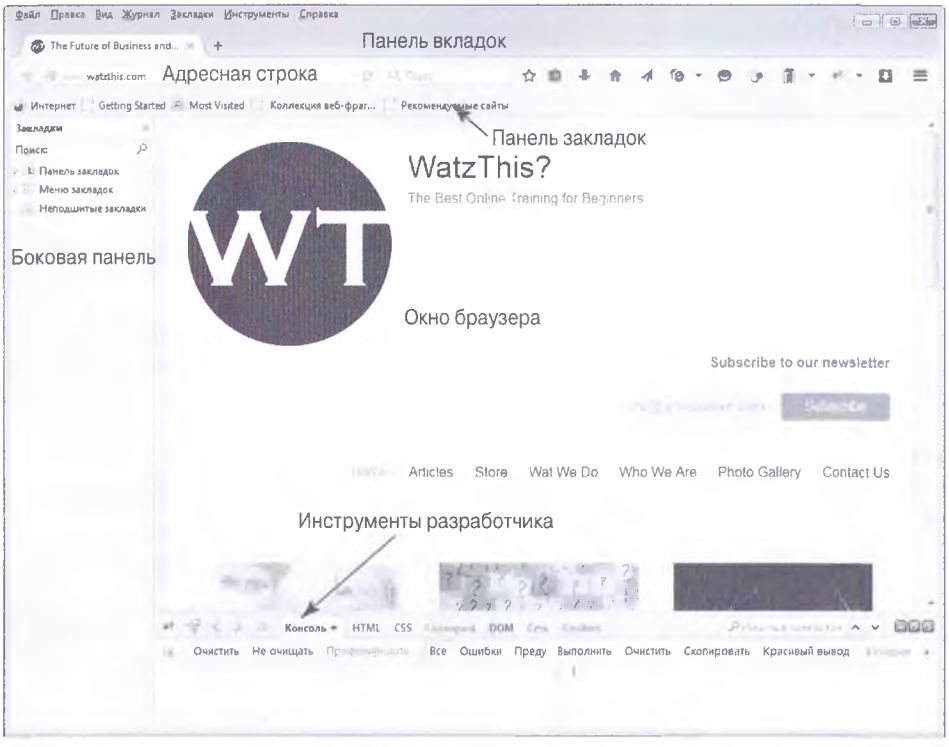


Рис. 9.1. Пользовательский интерфейс браузера

Загрузчик

Загрузчик — это часть браузера, которая обменивается данными с сервером и загружает веб-страницы, сценарии, стили CSS, графику и другие компоненты веб-страницы. Чаще всего именно стадия загрузки является основной виновницей того, что полного открытия страницы пользователю приходится дожидаться в течение некоторого времени.

HTML-страница — это та часть веб-страницы, которая должна загружаться в первую очередь, поскольку она содержит ссылки, внедренные сценарии и стили, подлежащие обработке до отображения готовой страницы на экране.

На рис. 9.2 показана вкладка Network (Сеть) средства Инструменты разработчика браузера Chrome. В ней отображаются все процессы, происходящие во время загрузки

веб-страницы, вместе с временной шкалой, позволяющей судить о том, сколько времени занимает загрузка каждой части.

По завершении загрузки HTML-документа браузер открывает несколько соединений с сервером для загрузки остальных частей веб-страницы с максимально возможной скоростью. Вообще говоря, части веб-страницы, связанные с HTML-документом (известные под названием *ресурсы*), загружаются в том порядке, в каком они появляются в HTML-документе. Например, сценарий, подключенный в верхней части страницы в элементе `head`, будет загружен раньше, чем сценарий, подключенный в нижней части страницы.

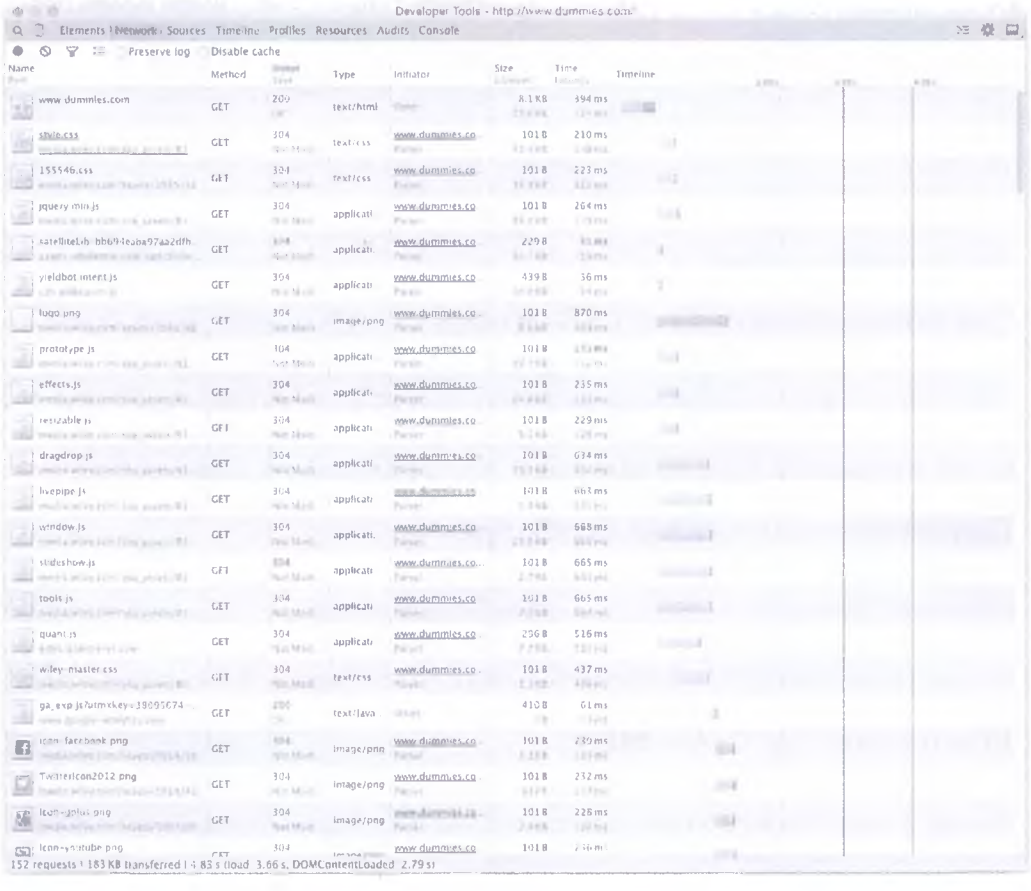


Рис. 9.2. Загрузка веб-страницы в браузер

Последовательность загрузки ресурсов критически влияет на эффективность и скорость отображения веб-страниц для пользователя. Чтобы страница отображалась корректно, должны быть загружены и проанализированы примененные к ней CSS-стили. Поэтому CSS-стили всегда должны загружаться в элементе `head`, располагающемся в верхней части веб-страницы.

Иногда на процесс отображения веб-страниц влияет также JavaScript, однако чаще всего это сказывается лишь на функциональности. Если сценарий воздействует на визуализацию веб-страницы, то он должен загружаться в заголовке документа (после CSS-стилей). Сценарии, не оказывающие критического влияния на отображение веб-страниц, должны подключаться в самом конце элемента `body` (непосредственно перед тегом `</body>`), чтобы загрузка их ресурсов не блокировала браузер, тем самым задерживая отображение другой информации.

Синтаксический анализ HTML-документа

По окончании процесса загрузки веб-страницы компонент браузера, ответственный за выполнение синтаксического анализа HTML-документа (синтаксический анализатор, или парсер), приступает к разбору HTML-разметки и создания модели веб-страницы, получившей название *DOM* (Document Object Model — объектная модель документа). Эту модель, подробному рассмотрению которой посвящена глава 10, можно уподобить карте веб-страницы. Программисты на JavaScript используют эту карту для получения доступа к различным частям веб-страницы и манипулирования ими.

Завершив синтаксический анализ HTML-кода, браузер начинает загрузку остальных частей веб-страницы.

Синтаксический анализ CSS-стилей

Сразу же после загрузки CSS-стилей веб-страницы браузер выполняет их синтаксический анализ и определяет, какие из них применяются к HTML-документу. Это сложный процесс, включающий многократные проходы по документу, целью которого является корректное применение стилей с учетом их взаимного влияния.

Синтаксический анализ JavaScript

Следующий этап отображения веб-страницы — синтаксический анализ сценариев JavaScript. Анализатор JavaScript компилирует и выполняет сценарии в том порядке, в каком они появляются в документе. Если JavaScript-код добавляет или удаляет элементы, текст или стили в HTML DOM, то браузер соответственно обновляет страницу.

Компоновка и визуализация

Наконец, как только все ресурсы веб-страницы загружены и проанализированы, браузер определяет, в каком виде должна отобразиться страница, после чего отображает ее. Если только вами не было задано ожидание завершения какого-либо сценария, включенного ранее в документ, компоновка и визуализация сценариев выполняются в том порядке, в каком они следуют в документе.

Вообще говоря, чем быстрее отобразится страница для пользователя, тем лучше, даже если на первоначальном этапе функциональность страницы будет неполной. Такая стратегия (носящая название *отложенная загрузка*) намеренно используется в современных браузерах для улучшения пользовательского восприятия веб-страниц. Если вам приходилось сталкиваться

с ситуациями, когда веб-страница открыта, но ввод данных в форму становится возможным лишь спустя некоторое время, то вы уже знаете, что такое отложенная загрузка.

Взаимодействие с BOM

JavaScript-программисты могут получать информацию о браузере пользователя и управлять различными аспектами пользовательского опыта посредством интерфейса прикладного программирования (Application Programming Interface — API), получившего название *объектная модель браузера* (Browser Object Model — BOM).

Какого-либо утвержденного стандарта объектной модели браузера не существует. В различных браузерах эта модель реализуется по-разному. Однако существуют некоторые общепринятые стандарты того, как JavaScript может взаимодействовать с браузерами.

Объект `Navigator`

Объект `Navigator` предоставляет JavaScript доступ к информации о браузере пользователя. Свое имя он получил по названию первого браузера, в котором был реализован, — Netscape Navigator. Объект `Navigator` не встроен в JavaScript, но доступ к нему из JavaScript возможен как к браузерному средству. Почти все производители браузеров, описывая этот высокоуровневый браузерный объект в документации, используют одну и ту же терминологию.

Объект `Navigator` обеспечивает доступ к следующей полезной информации:

- ✓ название браузера;
- ✓ версия браузера;
- ✓ физическое расположение компьютера, на котором установлен браузер (если пользователь разрешил доступ к геолокационной информации);
- ✓ язык, используемый в браузере;
- ✓ тип компьютера, на котором установлен браузер.

В табл. 9.1 приведены все свойства браузерного объекта `Navigator`.

Таблица 9.1. Свойства объекта `Navigator`

Свойство	Использование
`appCodeName`	Получает кодовое имя браузера
`appName`	Получает имя браузера
`appVersion`	Получает информацию о версии браузера
`cookieEnabled`	Сообщает, разрешено ли использование cookie-файлов в браузере
`geolocation`	Может использоваться для определения физического местоположения компьютера пользователя
`language`	Получает язык браузера
`onLine`	Идентифицирует состояние подключения браузера к сети

Окончание табл. 9.1

Свойство	Использование
`platform`	Получает платформу, для которой скомпилирован браузер
`product`	Получает имя движка браузера
`userAgent`	Получает данные, передаваемые пользовательским агентом браузера на серверы

Для получения значений свойств браузера используется тот же синтаксис, что и для получения значений свойств любого объекта, а именно: точечная или скобочная нотация. В листинге 9.1 приведен код, который, будучи открытым в браузере, отображает текущие свойства объекта `Navigator` и их значения.

Листинг 9.1. Свойства объекта `Navigator` и их значения

```
<head>
<style>
  .columns {
    -webkit-column-count: 6; /* Chrome, Safari, Opera */
    -moz-column-count: 6; /* Firefox */
    column-count: 6;
    }
  </style>
</head>
<body>
  <div class="columns">
    <script>
      for (var prop in navigator){
        document.write (prop + ": " + navigator[prop] +
          "<br>");
      }
    </script>
  </div>
</body>
</html>
```

Результат работы этой программы в браузере Chrome представлен на рис. 9.3.

Если вы запустите код из листинга 9.1 в браузере, то заметите одну интересную вещь: создается впечатление, будто значение свойства `appName` не соответствует действительности. Например, данные, представленные на рис. 9.3, были получены с использованием браузера Google Chrome, и тем не менее для свойства `appName` выведено значение `Netscape`.

Это вводящее в заблуждение значение является пережитком тех давних дней, когда свойства объекта `Navigator` применялись программистами для того, чтобы определить, использует ли пользователь конкретную модель браузера, поддерживающую определенные свойства.

Когда появились новые браузеры, такие как Chrome или Firefox, в них для свойства `appName` было сохранено значение из браузера Netscape, что должно было подтверждать их совместимость с сайтами, проверяющими это свойство.

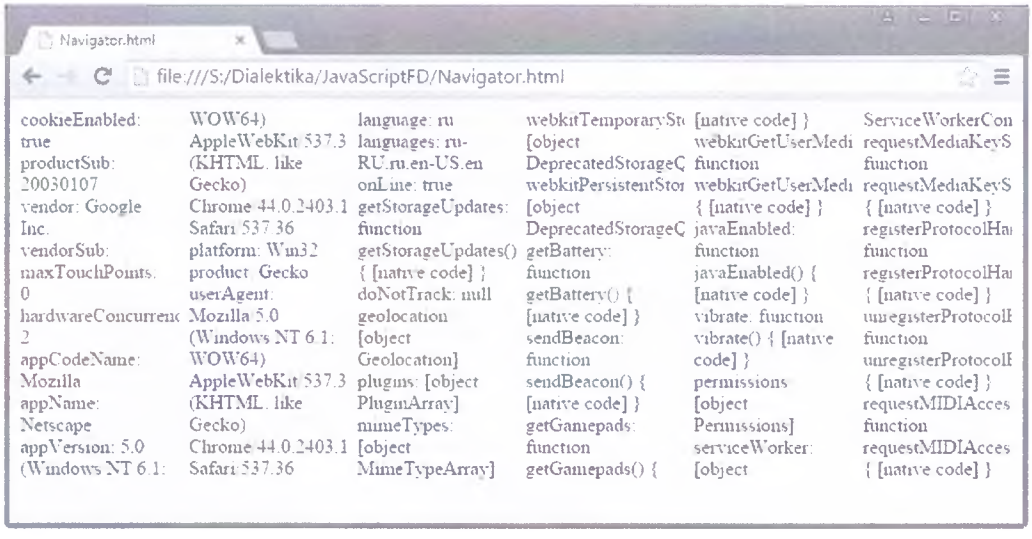


Рис. 9.3. Вывод всех свойств объекта `Navigator` и их значений

В наши дни проверка типа браузера не рекомендуется ввиду ее неэффективности, и вместо просмотра значения свойства `appName` выполняется проверка того, поддерживает ли браузер конкретную функциональную возможность. Наиболее распространенным способом обнаружения возможностей браузера является поиск в DOM объектов, связанных с функциональностью, которую предполагается использовать. Например, если вы хотите узнать, поддерживает ли браузер элемент HTML5 `audio`, то воспользуйтесь следующим тестом.

```
var test_audio = document.createElement("audio");
if (test_audio.play) {
  console.log ("Браузер поддерживает элемент HTML5 audio");
  } else {
  console.log ("Браузер не поддерживает элемент HTML5 audio");
  }
```

Объект `Window`

Основная область браузера называется *окном* (window). Это та область, в которую загружаются HTML-документы (и связанные с ними ресурсы). Каждая вкладка браузера представляется в JavaScript экземпляром объекта `Window`. Свойства объекта `Window` перечислены в табл. 9.2.

Таблица 9.2. Свойства объекта `Window`

Свойство	Использование
`closed`	Булево значение, указывающее на то, закрыто или открыто окно
`defaultStatus`	Получает или устанавливает текст, выводимый по умолчанию в строке состояния

Окончание табл. 9.2

Свойство	Использование
`document`	Ссылается на объект `Document` окна
`frameElement`	Получает элемент, такой как `<frame>` или `<object>`, в который внедрено окно
`frames`	Перечисляет все вспомогательные фреймы в текущем окне
`history`	Получает историю просмотра для текущего окна в браузере пользователя
`innerHeight`	Получает внутреннюю высоту окна
`innerWidth`	Получает внутреннюю ширину окна
`length`	Получает количество фреймов в окне
`location`	Получает объект `Location` для окна
`name`	Получает или устанавливает имя окна
`navigator`	Получает объект `Navigator` для окна
`opener`	Получает объект `Window`, создавший текущее окно
`outerHeight`	Получает наружную высоту окна, включая полосы прокрутки и панели инструментов
`pageXOffset`	Получает количество пикселей, прокрученных в горизонтальном направлении в окне
`pageYOffset`	Получает количество пикселей, прокрученных в вертикальном направлении в окне
`parent`	Ссылается на родительский объект текущего окна
`screen`	Ссылается на объект `Screen` окна
`screenLeft`	Получает расстояние в пикселях по горизонтали от левого края основного экрана до левого края текущего окна
`screenTop`	Получает расстояние в пикселях по вертикали от верхнего края основного экрана до верхнего края текущего окна
`screenX`	Получает горизонтальную координату относительно экрана
`screenY`	Получает вертикальную координату относительно экрана
`self`	Ссылается на текущее окно
`top`	Ссылается на самое верхнее окно браузера

Вот некоторые из наиболее популярных применений свойств объекта `window`:

- ✓ открытие в окне браузера страницы, расположенной в другом месте;
- ✓ определение размера окна браузера;
- ✓ возврат к ранее открывавшейся странице (например, как после щелчка на кнопке **Назад**).

Открытие веб-страницы с использованием свойства `window.location`

Запрос значения свойства `window.location` возвращает URL-адрес текущей страницы. Установка нового URL-адреса в этом свойстве приводит к загрузке веб-страницы, расположенной по указанному адресу, в то же окно.

В листинге 9.2 приведен пример веб-страницы со сценарием, который запрашивает ввод адреса пользователем и загружает указанную страницу в текущее окно браузера.

Листинг 9.2. Сценарий для загрузки веб-страницы в окно браузера с использованием свойства `window.location`

```
<html>
<head>
  <script>
    function loadNewPage (url){
      window.location = url;
    }
  </script>
</head>
<body>
  <script>
    var newURL =
        prompt("Пожалуйста, введите адрес веб-страницы!");
    loadNewPage(newURL);
  </script>
</body>
</html>
```

Результат выполнения кода из листинга 9.2 в браузере Chrome выглядит так, как показано на рис. 9.4.

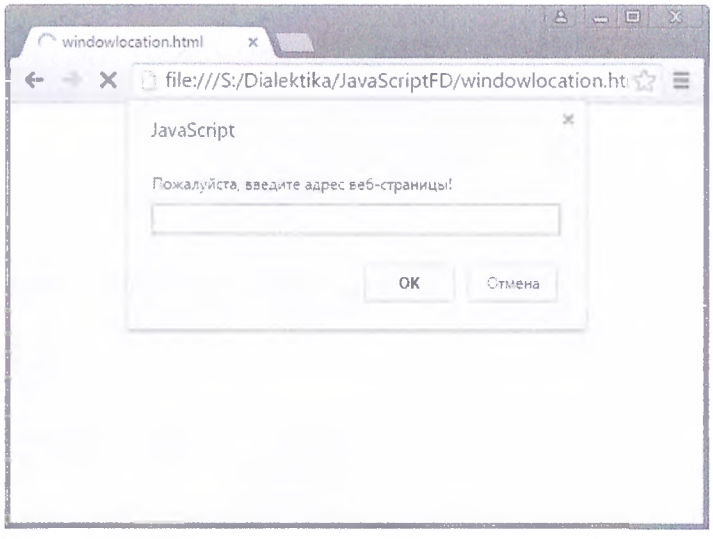


Рис. 9.4. Открытие страницы с помощью свойства `window.location`

Определение размера окна браузера

Когда проектируется веб-сайт или веб-приложение, которые должны корректно работать на устройствах различного типа (так называемый *адаптивный дизайн*), знание размера окна браузера, особенно его ширины, приобретает критическое значение.

Информацию о размерах текущего окна браузера в пикселях можно получить с помощью свойств `window.innerWidth` и `window.innerHeight`.

Также возможен и довольно распространен другой способ определения размеров окна браузера — с помощью стилей CSS. Однако в обработке полос прокрутки с помощью CSS и JavaScript имеются некоторые различия, в силу которых один из этих подходов может оказаться для вас более приемлемым.

Рассмотрим простой пример адаптивного дизайна, основанный на использовании JavaScript. Выполните программу, приведенную в листинге 9.3, в своем браузере. Если ширина окна вашего браузера меньше 500 пикселей, отобразится одно сообщение. Если же она превышает 500 пикселей, то отобразится другое сообщение.

Листинг 9.3. Изменение веб-страницы в зависимости от ширины окна

```
<html>
<head>
  <title>Адаптация к значению window.innerWidth</title>
</head>
<body>
  <script>
    var currentWidth = window.innerWidth;
    if (currentWidth > 500) {
      document.write("<h1>Ваше окно большое.</h1>");
    } else {
      document.write("<h1>Ваше окно небольшое.</h1>");
    }
  </script>
</body>
</html>
```

Чтобы протестировать пример адаптивного дизайна, приведенный в листинге 9.3, выполните следующие действия.

1. **Откройте в своем браузере HTML-документ, который содержит код, приведенный в листинге 9.3.**

   Если ширина окна браузера превышает 500 пикселей, то вы увидите сообщение о том, что ваше окно большое.

2. **Перетащите нижний правый угол окна, чтобы сделать его настолько узким, насколько это возможно, как показано на рис. 9.5.**

3. **Щелкните на кнопке обновления страницы или нажмите клавиши <Command+R> (Mac) или <Ctrl+R> (Windows).**

   Обратите внимание на то, что теперь сообщение на странице говорит о том, что окно имеет небольшой размер.

Создание кнопки Назад с помощью свойств `location` и `history`

Свойство `history` объекта `window` — это предназначенная только для чтения ссылка на объект `history`, который хранит информацию о страницах, ранее посещенных пользователем в текущем окне браузера. Обычное применение объекта `history` — активизация кнопок, возвращающих пользователя к ранее просмотренным страницам.

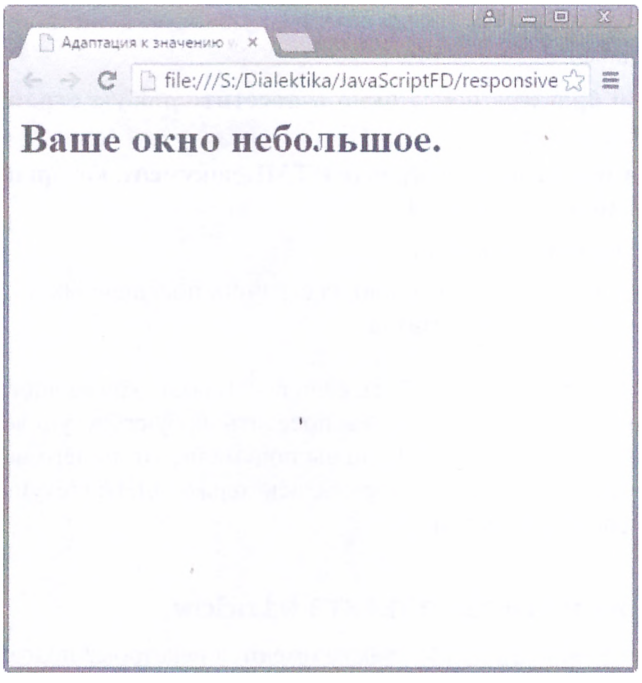


Рис. 9.5. Отображение разных сообщений в зависимости от ширины окна

Листинг 9.4. Реализация кнопки Назад в веб-приложении

```
<html>
<head>
  <title>Создание кнопки Назад</title>
  <script>
    function takeMeBack () {
      window.location(window.history.go(-1));
    }
    function getHistoryLength () {
      var l = window.history.length;
      return l;
    }
  </script>
</head>
<body>
  <script>
    var historyLength = getHistoryLength();
    document.write ("<p>Добро пожаловать! количество
             страниц, посещенных вами в этом окне: " +
            historyLength + ".</p> ");
  </script>
  <br>
  <a href="javascript:void(0);" onclick="takeMeBack();">
            Назад</a>
</body>
</html>
```

Чтобы использовать кнопку Назад, создаваемую кодом в листинге 9.4, выполните следующие действия.

1. **Откройте в браузере новое окно и посетите любую страницу, например `www.watzthis.com`.**
2. **Находясь в том же окне, откройте HTML-документ, который содержит код, приведенный в листинге 9.4.**
3. **Щелкните на кнопке Назад.**

   Браузер вернет вас на последнюю из страниц, посещенных до того, как вы открыли страницу с кнопкой Назад.

Догадайтесь, что произойдет, если вы откроете код из листинга 9.4 в новой вкладке браузера, прежде чем посетить любую другую веб-страницу, находясь на этой вкладке? Если вы подумали, что ничего не произойдет, то вы правы! Если в окне отображалась только ОДНА (текущая) страница, то и возвращаться некуда.

Использование методов объекта `window`

Кроме свойств, у объекта `window` также имеются некоторые полезные методы, которые программисты на JavaScript должны знать и использовать. Полный список этих методов приведен в табл. 9.3.

Метод — это лишь другое название для функции, содержащейся в объекте.

Таблица 9.3. Методы объекта `Window`

Метод	Использование
`alert()`	Отображает окно оповещения, которое содержит текст сообщения и кнопку OK
`atob()`	Декодирует строку, закодированную с помощью системы шифрования base64
`blur()`	Делает текущее окно неактивным
`clearInterval()`	Отменяет повторное выполнение кода, заданного методом `setInterval()`
`clearTimeout()`	Отменяет выполнение кода, заданного методом `setTimeout()`
`close()`	Закрывает текущее окно
`confirm()`	Отображает диалоговое окно, которое содержит необязательное сообщение и две кнопки — OK и Cancel (Отмена)
`createPopup()`	Создает всплывающее окно
`focus()`	Делает текущее окно активным
`moveBy()`	Перемещает текущее окно на заданную величину смещения
`moveTo(0`	Перемещает текущее окно в заданную позицию
`open()`	Открывает новое окно

Окончание табл. 9.3

Метод	Использование
`print()`	Выводит содержимое текущего окна
`prompt()`	Отображает окно запроса, ожидающее пользовательского ввода
`resizeBy()`	Изменяет размеры окна на заданное количество пикселей
`resizeTo()`	Изменяет размеры окна до заданных значений ширины и высоты
`scrollBy()`	Прокручивает документ на указанное количество пикселей
`scrollTo()`	Прокручивает документ до указанных координат
`setInterval()`	Периодически вызывает функцию или выполняет выражение через определенные промежутки времени (заданные в миллисекундах)
`setTimeout()`	Однократно вызывает функцию или выполняет выражение по истечении определенного промежутка времени (заданного в миллисекундах)
`stop()`	Прекращает загрузку текущего окна

Глава 10

Манипулирование документами с помощью DOM

В этой главе...

- Знакомство с DOM (объектная модель документа)
- Работа с узлами
- Перемещение по дереву узлов
- Выбор элементов

"Загадочны не вещи. Загадка — ваши глаза".
Элизабет Боуэн

Знание DOM дает возможность манипулировать текстом или HTML-разметкой на веб-странице. Используя DOM, можно создавать анимацию, обновлять данные без перезагрузки веб-страниц, перемещать объекты в браузере, а также делать много других полезных вещей.

Что такое DOM

DOM (Document Object Model — объектная модель документа) — это интерфейс, посредством которого JavaScript работает с HTML-документами в окнах браузера. Данная модель допускает графическое представление в виде перевернутого дерева, где каждая часть документа изображается в виде ветви, исходящей из той части документа, в которой она содержится.

Пример разметки веб-страницы представлен в листинге 10.1, а ее DOM-представление — на рис. 10.1.

Листинг 10.1. HTML-документ

```
<html>
<head>
  <title>Bob's Appliances </title>
</head>
<body>
  <header>
    <img src="logo.gif" width="100" height="100" alt="Site
          Logo">
```

```
  </header>
  <div>
    <h1>Welcome to Bob's</h1>
    <p>The home of quality appliances</p>
  </div>
  <footer>
    copyright &copy; Bob
  </footer>
</body>
</html>
```

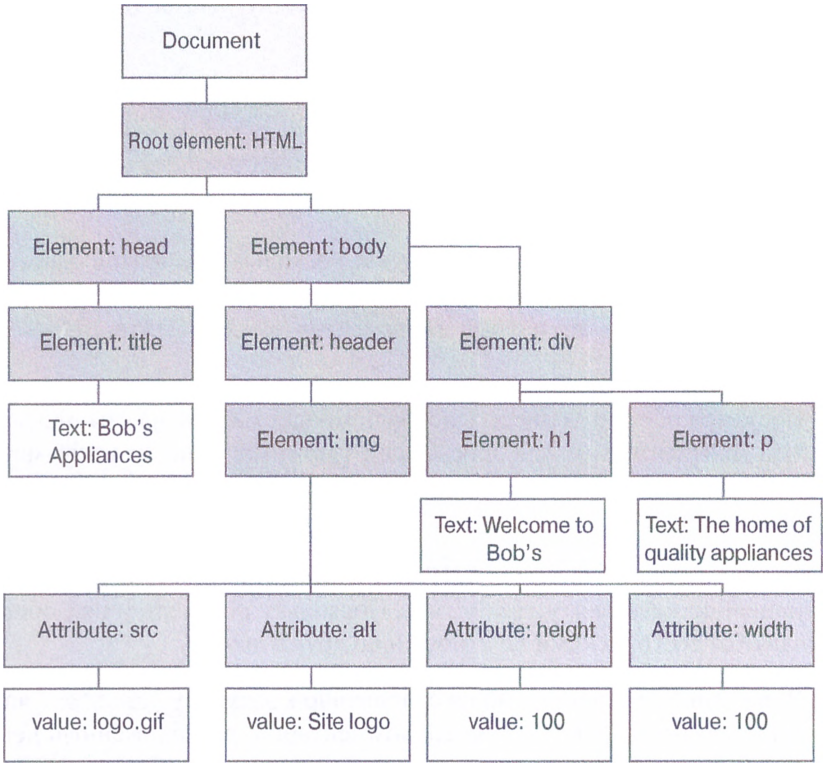


Рис. 10.1. DOM-представление документа из листинга 10.1

Иерархическое дерево DOM состоит из отдельных компонентов, называемых *узлами*. Главный узел, от которого ответвляются другие узлы, называется *узлом документа*. Непосредственно под узлом документа располагается *узел корневого элемента*. Для HTML-документа корневым узлом является HTML. За корневым узлом следуют другие элементы, атрибуты, а также элементы содержимого документа, каждый из которых представлен узлом дерева, происходящим от другого узла.

В DOM имеются узлы нескольких типов.

- ✓ **Узел документа.** Представляет весь HTML-документ.
- ✓ **Узлы элементов.** HTML-элементы.

- ✓ **Узлы атрибутов.** Атрибуты, ассоциированные с элементами.
- ✓ **Текстовые узлы.** Текстовое содержимое элементов.
- ✓ **Узлы комментариев.** Имеющиеся в документе HTML-комментарии.

Отношения между узлами

Иерархическая система отношений между узлами DOM-дерева напоминает отношения между родственниками в семье. Это сходство настолько велико, что даже технические термины, описывающие соотношения между узлами, получили свои названия от отношений, существующих в семье.

- ✓ У каждого узла, за исключением корневого, имеется один *родительский узел*.
- ✓ Каждый узел может иметь любое количество *дочерних узлов*.
- ✓ Узлы, имеющие общего родителя, называют *сестринскими*.

Поскольку HTML-документы часто содержат несколько элементов одного и того же типа, DOM позволяет обращаться к различным элементам в узле, используя их индексы. Например, `p[0]` — это первый из элементов `<p>` документа, `p[1]` — второй элемент `<p>` и т.д.

Несмотря на то что список узлов напоминает массив, на самом деле это не так. Действительно, для просмотра содержимого списка узлов можно использовать циклы, однако методы для работы с массивами к спискам узлов неприменимы.

В примере, приведенном в листинге 10.2, все три элемента `<p>` являются дочерними по отношению к элементу `<section>`. Поскольку родитель у них общий, эти элементы являются сестринскими по отношению друг к другу.

В листинге 10.2 дочерними по отношению к элементу `<section>` являются также комментарии. Последний комментарий, расположенный непосредственно перед закрывающим тегом `<section>`, называется *последним дочерним элементом* элемента `<section>`.

Зная отношения, существующие между узлами документа, вы сможете использовать DOM-дерево для нахождения любого элемента.

Листинг 10.2. Отношения между родительскими, дочерними и сестринскими узлами в HTML-документе

```
<html>
<head>
  <title>HTML-семейство</title>
</head>
<body>
  <section> <!-- гордый родитель трех элементов p,
```

```
        являющийся дочерним элементом элемента body -->
    <p>Первый</p> <!-- 1-й дочерний элемент элемента section,
        являющийся сестринским по отношению к двум
        элементам p -->
    <p>Второй</p> <!-- 2-й дочерний элемент элемента section,
        являющийся сестринским по отношению к двум
        элементам p -->
    <p>Третий</p> <!-- 3-й дочерний элемент элемента section,
        являющийся сестринским по отношению к двум
        элементам p--> -->
  </section>
</body>
</html>
```

В листинге 10.3 приведен HTML-документ, который содержит сценарий, отображающий все дочерние узлы элемента `<section>`.

Листинг 10.3. Отображение дочерних узлов элемента `section`

```
<html>
<head>
  <title>HTML-семейство</title>
</head>
<body>
  <section> <!-- гордый родитель трех элементов p,
        являющийся дочерним элементом элемента body -->
    <p>Первый</p> <!-- 1-й дочерний элемент элемента section,
        являющийся сестринским по отношению к двум
        элементам p -->
    <p>Второй</p> <!-- 2-й дочерний элемент элемента section,
        являющийся сестринским по отношению к двум
        элементам p -->
    <p>Третий</p> <!-- 3-й дочерний элемент элемента section,
        являющийся сестринским по отношению к двум
        элементам p -->
  </section>
  <h1>Узлы в элементе section</h1>
  <script>
    var myNodelist =
        document.body.childNodes[1].childNodes;
    for (i = 0; i < myNodelist.length; i++){
      document.write (myNodelist[i] + "<br>");
    }
  </script>
</body>
</html>
```

На рис. 10.2 показано, как выглядит результат работы кода из листинга 10.3 в браузере. Обратите внимание на то, что первым дочерним узлом элемента `section` является текстовый узел. Если вы внимательно присмотритесь к HTML-разметке, приведенной в листинге 10.3, то заметите, что между открывающим тегом `<section>` и комментарием имеется пробел. Но и этого единственного пробела оказалось достаточно для того,

чтобы создать узел в DOM-дереве. Вы всегда должны учитывать факторы подобного рода при навигации по дереву DOM с использованием отношений между узлами.

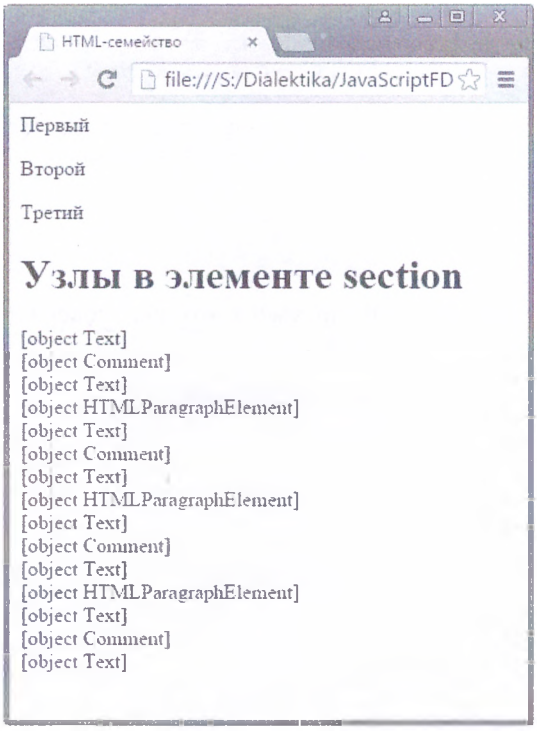


Рис. 10.2. Результат выполнения кода, приведенного в листинге 10.3, в браузере

DOM предоставляет также ряд ключевых слов для навигации по узлам с использованием их позиций по отношению к сестринским или родительским узлам. Соответствующие свойства перечислены ниже.

- ✓ `firstChild`. Ссылка на первый дочерний элемент узла.
- ✓ `lastChild`. Ссылка на последний дочерний элемент узла.
- ✓ `nextSibling`. Ссылка на следующий узел, дочерний по отношению к тому же родительскому узлу.
- ✓ `previousSibling`. Ссылка на предыдущий узел, дочерний по отношению к тому же родительскому узлу.

В листинге 10.4 показано, как использовать эти соотношения для обхода узлов DOM.

Листинг 10.4. Использование свойств `firstChild` и `lastChild` для выделения навигационных ссылок

```
<html>
<head>
  <title>Iguanas Are No Fun</title>
```

```
  <script>
    function boldFirstAndLastNav() {
      document.body.childNodes[1].firstChild.style.
          fontWeight="bold";
      document.body.childNodes[1].lastChild.style.
          fontWeight="bold";
    }
  </script>

</head>
<body>
  <nav><a href="home.html">Home</a> | <a
          href="why.html">Why Are Iguanas No Fun?</a> |
          <a href="what.html">What Can Be Done?</a> | <a
          href="contact.html">Contact Us</a></nav>
  <p>Iguanas are no fun to be around. Use the links above
          to learn more.</p>
  <script>
    boldFirstAndLastNav();
  </script>
</body>
</html>
```

Имейте в виду, что для корректного выбора и стилизации элементов с помощью свойств `firstChild` и `lastChild` все пробелы между элементами, содержащимися в элементе `<nav>`, должны быть удалены.

На рис. 10.3 показано, как выглядит документ, представленный в листинге 10.4, при его просмотре в браузере. Обратите внимание на то, что полужирным начертанием выделены лишь первая и последняя ссылки.

Это был первый пример, в котором мы использовали DOM для внесения изменений в существующие элементы в пределах документа. Однако такой метод выделения элементов почти никогда не применяется. Кроме того, он чреват внесением ошибок и слишком сложен для интерпретации и использования.

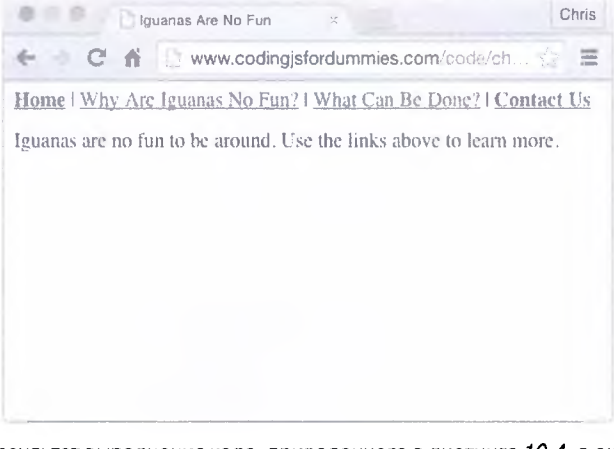


Рис. 10.3. Результат выполнения кода, приведенного в листинге 10.4, в окне браузере

Использование свойств и методов объекта Document

Объект Document предоставляет свойства и методы для работы с HTML-документами. Полный перечень свойств объекта Document приведен в табл. 10.1. Все методы объекта Document приведены в табл. 10.2.

Таблица 10.1. Свойства объекта Document

Свойство	Использование
anchors	Получает список всех якорей (элементов <a> с именами атрибутов) в документе
applets	Получает упорядоченный список всех аплетов в документе
baseURI	Получает базовый URI документа
body	Получает узел <body> или <frames> тела документа
cookie	Получает или задает пары "имя-значение" cookie-наборов в документе
doctype	Получает объявление типа документа
documentElement	Получает корневой элемент документа (например, элемент <html> в HTML-документе)
documentMode	Получает режим визуализации, используемый браузером
documentURI	Получает или устанавливает расположение документа
domain	Получает доменное имя сервера, загрузившего документ
embeds	Получает список всех элементов <embed> в документе
forms	Получает коллекцию всех элементов <form> в документе
head	Получает элемент <head> документа
images	Получает список всех элементов <img> в документе
implementation	Получает объект DOMImplementation, который обрабатывает документ
lastModified	Получает дату и время последнего изменения текущего документа
links	Получает коллекцию всех элементов <area> и <a> в документе, которые имеют атрибут href
readyState	Получает состояние загрузки документа. Возвращает значение loading в процессе загрузки документа, interactive — по завершении синтаксического анализа документа и complete — по завершении загрузки документа
referrer	Получает URL-адрес страницы, с которой был связан текущий документ
scripts	Получает список всех элементов <script> в документе
title	Получает или устанавливает заголовок документа
URL	Получает полный URL-адрес документа

Таблица 10.2. Методы объекта document

Метод	Использование
addEventListener()	Назначает обработчик событий в документе
adoptNode()	Заимствует узел из внешнего документа

Окончание табл. 10.2

Метод	Использование
`close()`	Закрывает для записи выходной поток документа, ранее открытый с помощью метода `document.open()`
`createAttribute()`	Создает узел атрибута
`createComment()`	Создает узел комментария
`createDocumentFragment()`	Создает пустой фрагмент документа
`createElement()`	Создает узел элемента
`createTextNode()`	Создает текстовый узел
`getElementById()`	Получает элемент по заданному атрибуту `Id`
`getElementsByClassName()`	Получает все элементы с указанным именем класса
`getElementsByName()`	Получает все элементы с указанным именем
`getElementsByTagName()`	Получает все элементы с указанным именем тега
`importNode()`	Копирует и импортирует узел из внешнего документа
`normalize()`	Удаляет пустые текстовые узлы и соединяет смежные
`open()`	Открывает документ для записи
`querySelector()`	Получает первый из тех элементов в документе, которые соответствуют указанным селекторам
`querySelectorAll()`	Получает список всех элементов, соответствующих указанным селекторам
`removeEventListener()`	Удаляет из документа обработчик событий, добавленный ранее с помощью метода `.addEventListener()`
`renameNode()`	Переименовывает существующий узел
`write()`	Записывает JavaScript-код или HTML-выражения в документ
`writeIn()`	Записывает JavaScript-код или HTML-выражения в документ с добавлением символа новой строки после каждой инструкции

Использование свойств и методов объекта Element

Объект `Element` предоставляет свойства и методы для работы с HTML-элементами в документе. Полный перечень свойств объекта `Element` приведен в табл. 10.3. Все методы объекта `Element` приведены в табл. 10.4.

Таблица 10.3. Свойства объекта `Element`

Свойство	Использование
`accessKey`	Получает или устанавливает атрибут `accessKey` элемента
`attributes`	Получает коллекцию всех атрибутов элемента, зарегистрированных в указанном узле (возвращает `NameNodeMap`)
`childElementCount`	Получает количество дочерних элементов в указанном узле

Продолжение табл. 10.3

Свойство	Использование
childNodes	Получает список дочерних узлов элемента
children	Получает список дочерних элементов данного элемента
classList	Получает имя класса (имена классов) элемента
className	Получает или устанавливает значение атрибута `class` элемента
clientHeight	Получает внутреннюю высоту элемента, включая внутренние отступы
clientLeft	Получает ширину левой границы элемента
clientTop	Получает ширину верхней границы элемента
clientWidth	Получает ширину элемента, включая внутренние отступы
contentEditable	Получает или устанавливает режим редактируемости элемента
dir	Получает или устанавливает значение атрибута `dir` элемента
firstChild	Получает первый дочерний узел элемента
firstElementChild	Получает первый дочерний элемент данного элемента
id	Получает или устанавливает значение атрибута `id` элемента
innerHtml	Получает или устанавливает содержимое элемента
isContentEditable	Возвращает значение `true`, если изменение содержимого элемента разрешено, иначе — `false`
lang	Получает URL-адрес страницы, с которой был связан текущий документ
lastChild	Получает список всех элементов `<script>` в документе
lastElementChild	Получает или устанавливает заголовок документа
namespaceURI	Получает URI имени пространства для первого узла в элементе
nextSibling	Получает следующий узел на том же уровне узлов
nextElementSibling	Получает следующий элемент на том же уровне узлов
nodeName	Получает имя текущего узла
nodeType	Получает тип текущего узла
nodeValue	Получает или устанавливает значение узла
offsetHeight	Получает высоту элемента, включая внутренние вертикальные отступы, границы и полосу прокрутки
offsetWidth	Получает высоту элемента, включая внутренние горизонтальные отступы, границы и полосу прокрутки
offsetLeft	Получает смещение левого края элемента относительно его родительского элемента (`offsetParent`)
offsetParent	Ссылка на контейнер элемента
offsetTop	Получает смещение верхнего края элемента относительно его родительского элемента (`offsetParent`)
ownerDocument	Получает корневой элемент (узел документа)
parentNode	Получает родительский узел элемента
parentElement	Получает узел элемента, являющегося родительским по отношению к данному элементу

Окончание табл. 10.3

Свойство	Использование
previousSibling	Получает предыдущий сестринский узел
previousElementSibling	Получает предыдущий сестринский элемент
scrollHeight	Получает или устанавливает полную высоту элемента с учетом внутренних отступов
scrollLeft	Получает или устанавливает количество пикселей, на которые содержимое элемента прокручивается в горизонтальном направлении
scrollTop	Получает или устанавливает количество пикселей, на которые содержимое элемента прокручивается в вертикальном направлении
scrollWidth	Получает или устанавливает полную ширину элемента с учетом внутренних отступов
style	Получает или устанавливает значение атрибута `style` элемента
tabIndex	Получает или устанавливает значение атрибута `tabIndex` элемента
tagName	Получает имя тега элемента
textContent	Получает или устанавливает текстовое содержимое узла и его потомков
title	Получает или устанавливает значение атрибута `title` элемента
length	Получает количество узлов в списке `NodeList`

Таблица 10.4. Методы объекта `Element`

Метод	Использование
addEventListener()	Назначает обработчик событий для элемента
appendChild()	Вставляет новый дочерний узел в элемент (в качестве последнего дочернего узла)
blur()	Лишает элемент фокуса ввода
click()	Имитирует щелчок мышью на элементе
cloneNode()	Клонирует элемент
compareDocumentPosition()	Сравнивает позиции двух элементов в документе
contains()	Возвращает значение `true`, если узел является потомком данного узла; иначе — `false`
focus()	Переводит фокус ввода на данный элемент
getAttribute()	Получает значение указанного атрибута узла элемента
getAttributeNode()	Получает узел указанного атрибута
getElementsByClassName()	Получает коллекцию всех дочерних элементов с заданным именем класса
getElementsByTagName()	Получает коллекцию всех дочерних элементов с заданным именем тега
getFeature()	Получает объект, который реализует API заданного средства

Окончание табл. 10.4

Метод	Использование
`hasAttribute()`	Возвращает значение `true`, если элемент содержит указанный атрибут, иначе — `false`
`hasAttributes()`	Возвращает значение `true`, если элемент содержит любой из указанных атрибутов, иначе — `false`
`hasChildNodes()`	Возвращает значение `true`, если элемент содержит дочерние узлы, иначе — `false`
`insertBefore()`	Вставляет новый дочерний узел перед заданным существующим узлом
`isDefaultNamespace()`	Возвращает значение `true`, если заданное значение `namespaceURI` является пространством имен по умолчанию, иначе — `false`
`isEqualNode()`	Проверяет равенство двух элементов
`isSameNode()`	Проверяет, являются ли два элемента одним и тем же узлом
`isSupported()`	Возвращает значение `true`, если заданная возможность поддерживается элементом, иначе — `false`
`normalize()`	Соединяет указанные узлы со смежными узлами и удаляет пустые текстовые узлы
`querySelector()`	Получает первый дочерний элемент, соответствующий заданным CSS-селекторам элемента
`querySelectorAll()`	Получает все дочерние элементы, соответствующие заданным CSS-селекторам элемента
`removeAttribute()`	Удаляет заданный атрибут из элемента
`removeAttributeNode()`	Удаляет заданный узел атрибута из элемента и извлекает удаленный узел
`removeChild()`	Удаляет заданный дочерний узел
`replaceChild()`	Заменяет указанный дочерний узел другим
`removeEventListener()`	Удаляет указанный слушатель событий
`setAttribute()`	Устанавливает указанное значение для заданного атрибута
`setAttributeNode()`	Изменяет или устанавливает заданный узел атрибута
`toString()`	Преобразует элемент в строку
`item()`	Получает узел с указанным индексом из списка `NodeList`

Работа с содержимым элементов

Используя HTML DOM, можно отображать типы и значения узлов. Кроме того, можно устанавливать значения свойств DOM-элементов с помощью объекта `Element`. Когда вы задаете свойства DOM-элементов с помощью JavaScript, новые значения изменяют внешний вид HTML-документа в режиме реального времени.

Изменение свойств элементов для их немедленного обновления в браузере без обновления или перезагрузки веб-страницы является краеугольным камнем комплексного подхода к проектированию веб-приложений, получившего название Web 2.0.

Свойство innerHTML

Самым важным свойством элемента, которое можно изменить посредством DOM, является свойство innerHTML.

Свойство innerHTML элемента содержит все, что заключено между его начальным и конечным тегами. Например, свойство innerHTML элемента div содержит элемент p и его дочерний текстовый узел:

```
<body><div><p>Это некоторый текст.</p></div></body>
```

В веб-программировании довольно часто сначала создают пустые элементы div в HTML-документе, а затем динамически заполняют их содержимым с помощью свойства innerHTML.

Для извлечения и отображения значения свойства innerHTML можно использовать следующий код.

```
var getTheInner = document.body.firstChild.innerHTML;
document.write (getTheInner);
```

Вот как будет выглядеть значение, которое в этом коде выведет метод document.write():

```
<p>Это некоторый текст.</p>
```

Установка значения свойства innerHTML осуществляется точно так же, как установка свойства любого объекта:

```
document.body.firstChild.innerHTML = "Привет!";
```

В результате выполнения предыдущего JavaScript-кода элемент p и текст в исходной разметке будут заменены словом "Привет!". При этом исходный HTML-документ останется неизменным, но его DOM-представление и визуализированная веб-страница обновятся и будут отражать новое значение. Поскольку DOM-представление HTML-документа — это именно то, что отображается в браузере, вид вашей страницы также обновится.

Установка значений атрибутов

Для установки значения HTML-атрибута можно использовать метод setAttribute():

```
document.body.firstChild.innerHTML.setAttribute("class","myclass");
```

Результат выполнения этой инструкции будет состоять в том, что первый дочерний элемент элемента body получит новый атрибут с именем "class" и значением "myClass".

Получение элемента по его идентификатору, имени тега или классу

Методы группы `getElementBy...()` обеспечивают легкий доступ к любому элементу или группе элементов в документе без привлечения родственных отношений узлов. Из них чаще всего используются следующие методы:

- ✓ `getElementById()`;
- ✓ `getElementsByTagName()`;
- ✓ `getElementsByClassName()`.

Метод `getElementById()`

Метод `getElementById()`, до сих пор чаще всего используемый для выбора элементов, занимает важное место в современной веб-разработке. Это весьма удобное средство, позволяющее находить и обрабатывать любой элемент по его идентификатору — уникальному атрибуту `id`. Что бы ни происходило с HTML-документом, он всегда будет у вас под рукой, обеспечивая надежный выбор именно того элемента, который вам нужен.

Потрясающие возможности метода `getElementById()`, позволяющие поддерживать в документе весь JavaScript-код или разбить его на модули, продемонстрированы в листинге 10.5.

Используя этот метод, можно работать с любым элементом в документе, где бы он ни находился, коль скоро вам известен его идентификатор.

Листинг 10.5. Выбор элементов с помощью метода `getElementById()`

```
<html>
<head>
  <title>Использование метода getElementById()</title>
  <script>
    function calculateMPG(miles,gallons){
      document.getElementById("displayMiles").
          innerHTML = parseInt(miles);
      document.getElementById("displayGallons").
          innerHTML = parseInt(gallons);
      document.getElementById("displayMPG").
          innerHTML = miles/gallons;
    }
  </script>
</head>
<body>
  <p>Вы проехали <span id="displayMiles">___</span>
          км.</p>
  <p>Вы использовали <span id="displayGallons">___</span>
          литров бензина.</p>
  <p>Пробег на 1 л топлива: <span id="displayMPG">___</span> км
  <script>
    var milesDriven = prompt("Введите пробег (км)");
```

```
    var gallonsGas = prompt("Введите расход бензина (л)");
    calculateMPG(milesDriven,gallonsGas);
  </script>
</body>
</html>
```

Метод getElementsByTagName()

Метод `getElementsByTagName()` возвращает список узлов всех элементов с указанным именем тега. Например, в листинге 10.6 этот метод используется для выбора всех элементов `h1` и замены значений их свойств `innerHTML` последовательными числами.

Листинг 10.6. Выбор и изменение элементов с помощью метода getElementsByTagName()

```
<html>
<head>
  <title>Использование метода getElementsByTagName</title>
  <script>
    function numberElements(tagName){
    var getTags =
        document.getElementsByTagName(tagName);
      for(i=0; i < getTags.length; i++){
        getTags[i].innerHTML = i+1;
      }
    }
</script>
</head>
<body>
  <h1>этот текст исчезнет</h1>
  <h1>этот текст будет перезаписан</h1>
  <h1>JavaScript затрет этот текст</h1>
  <script>
    numberElements("h1");
  </script>
</body>
</html>
```

Метод getElementsByClassName()

Метод `getElementsByClassName()` работает в основном так же, как и метод `getElementsByTagName()`, но выбирает элементы, используя значения атрибута `class`. Функция в листинге 10.7 выбирает элементы, в которых атрибут `class` имеет значение `"error"`, и изменяет значение их свойства `innerHTML`.

Листинг 10.7. Выбор и изменение элементов с помощью метода getElementsByClassName()

```
<html>
<head>
  <title>Использование метода getElementsByClassName()</title>
```

```
  <script>
    function checkMath(result){
      var userMath =
          document.getElementById("answer1").value;
      var errors =
          document.getElementsByClassName("error");
      if(parseInt(userMath) != parseInt(result)) {
        errors[0].innerHTML = "Неверно. Вы ввели "
          + userMath + ". Правильный ответ: " + result;
      } else {
        errors[0].innerHTML = "Верно!";
      }
    }
  </script>
</head>
<body>
    <label for = "number1">4+1 = </label><input type="text"
           id="answer1" value="">
    <button id="submit" onclick="checkMath(4+1);">Проверить!</button>
    <h1 class="error"></h1>
</body>
</html>
```

Результат выполнения листинга 10.7 в браузере с последующим вводом неверного ответа показан на рис. 10.4.

Обратите внимание на использование в листинге 10.7 атрибута `onclick` в элементе `button`. Более подробную информацию об обработчиках событий вы получите в главе 11.

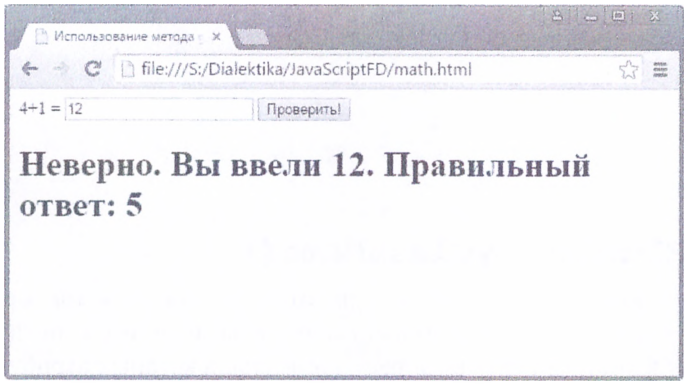


Рис. 10.4. Использование метода `getElementsByClassName()` для выбора элемента и вывода сообщения об ошибке

Использование свойств объекта Attribute

Объект `Attribute` предоставляет свойства для работы с атрибутами HTML-элементов. Полный перечень свойств объекта `Attribute` приведен в табл. 10.5.

Таблица 10.5. Свойства объекта `Attribute`

Свойство	Использование
`isId`	Возвращает значение `true`, если атрибутом является `id`, иначе — `false`
`name`	Получает имя атрибута
`value`	Получает или устанавливает значение атрибута
`specified`	Возвращает значение `true`, если данный атрибут задан, иначе — `false`

Создание и присоединение элементов

Для создания нового элемента в HTML-документе используется метод `document.createElement()`. В результате его применения создаются новые начальный и конечный теги указанного типа.

Пример использования этого метода для динамического создания списка в HTML-документе из массива приведен в листинге 10.8.

Листинг 10.8. Использование метода `document.createElement()` для генерации таблицы из массива

```
<html>
<head>
  <title>Генерирование списка</title>
</head>
<body>
  <h1>Некоторые типы мячей</h1>
  <ul id="ballList">
  </ul>

  <script>
    var typeOfBall = ["basket", "base", "soccer", "foot",
          "hand"];
    for (i=0; i<typeOfBall.length; i++) {
      var listElement = document.createElement("li");
      listElement.innerHTML = typeOfBall[i];
      document.getElementById("ballList").appendChild
            (listElement);
    }
  </script>

</body>
</html>
```

Удаление элементов

Несмотря на замечательные возможности манипулирования HTML-документами, DOM не пользуется большой популярностью у профессиональных JavaScript-программистов. Иногда эта модель ведет себя не совсем корректно и неоправданно усложняет выполнение некоторых операций.

Одним из больших недостатков DOM является отсутствие возможности непосредственного удаления элемента из документа. Вместо этого вы должны выполнить средствами DOM поиск родителя элемента, подлежащего удалению, а затем удалить данный элемент как дочерний по отношению к найденному элементу. Возможно, это объяснение звучит несколько запутанно, поэтому предлагаем ознакомиться с примером, приведенным в листинге 10.9, который должен прояснить все.

Листинг 10.9. Удаление элемента из документа

```
<html>
<head>
  <title>Удаление элемента</title>
  <script>
    function removeFirstParagraph(){
      var firstPara =
            document.getElementById("firstparagraph");
      firstPara.parentNode.removeChild(firstPara);
    }
  </script>
</head>
<body>
  <div id="gibberish">
    <p id="firstparagraph">Lorem ipsum dolor sit amet,
          consectetur adipiscing elit. Vestibulum
          molestie pulvinar ante, a volutpat est
          sodales et. Ut gravida justo ac leo euismod,
          et tempus magna posuere. Cum sociis natoque
          penatibus et magnis dis parturient montes,
          nascetur ridiculus mus. Integer non mi iaculis,
          facilisis risus et, vestibulum lorem. Sed quam
          ex, placerat nec tristique id, mattis fringilla
          ligula. Maecenas a pretium justo. Suspendisse
          sit amet nibh consectetur, tristique tellus
          quis, congue arcu. Etiam pellentesque dictum
          elit eget semper. Phasellus orci neque, semper
          ac tortor ac, laoreet ultricies enim.</p>
  </div>
  <button onclick="removeFirstParagraph();">Это
          тарабарщина!</button>
</body>
</html>
```

Когда вы запустите этот код в браузере и щелкнете на кнопке, событие `onclick` вызовет функцию `removeFirstParagraph()`.

Первое, что делает метод `removeFirstParagraph()`, — выбирает элемент, который вы фактически хотите удалить, а именно элемент с атрибутом `id`, равным `"firstpatagraph"`. Последующее удаление первого абзаца осуществляется с помощью метода `removeChild()`.

Глава 11

Использование событий в JavaScript

В этой главе...

- События JavaScript
- Использование обработчиков для реагирования на события
- Типы обработчиков событий

"А теперь вечерние новости о событиях минувшего дня".
Дэн Разер

Веб-страницы — это нечто гораздо большее, нежели простое статическое отображение текста и графики. JavaScript придает веб-страницам интерактивность, добавляя возможность выполнения полезной работы. Важной составляющей JavaScript, позволяющей веб-страницам выполнять полезные функции в браузере, являются средства, обеспечивающие возможность реагировать на события.

События

События — это все то, что происходит в браузере (например, загрузка веб-страницы), а также то, что делает пользователь (например, щелчки мышью, нажатия клавиш, перемещения мыши и т.п.). В браузере постоянно происходят какие-то события.

HTML DOM позволяет JavaScript распознавать события, происходящие в браузере, и реагировать на них. События можно разделить на отдельные группы в соответствии с тем, с какими HTML-элементами или объектами браузера они связаны. В табл. 11.1 представлены события, поддерживаемые любым HTML-элементом.

Существуют также другие типы событий, которые поддерживаются всеми элементами, кроме элементов `body` и `frame`. Список этих событий приведен в табл. 11.2.

Таблица 11.1. События, поддерживаемые всеми HTML-элементами

Событие	Условие возникновения
`abort`	Прерывание загрузки файла
`change`	Изменение значения элемента с момента потери и последующего приобретения фокуса ввода
`click`	Выполнения щелчка мышью на элементе

Событие	Условие возникновения
dbclick	Выполнение двойного щелчка мышью на элементе
input	Изменение значения элемента <input> или <textArea>
keydown	Нажатие клавиши
keyup	Отпускание клавиши после того, как она была нажата
mousedown	Нажатие кнопки мыши над элементом
mouseenter	Перемещение указателя мыши в область элемента, к которому подключен слушатель событий
mouseleave	Перемещение указателя мыши за пределы элемента, к которому подключен слушатель событий
mousemove	Перемещение указателя мыши над элементом
mouseout	Перемещение указателя мыши за пределы элемента или одного из его дочерних элементов, к которому подключен слушатель событий
mouseover	Перемещение указателя мыши над элементом или одним из его дочерних элементов, к которому подключен слушатель событий
mouseup	Отпускание кнопки мыши над элементом
mousewheel	Вращение колесика мыши
onreset	Сброс формы
select	Выделение текста
submit	Отправка формы

Таблица 11.2. События, поддерживаемые всеми элементами, за исключением <body> и <frameset>

Событие	Условие возникновения
blur	Потеря элементом фокуса ввода
error	Ошибка при загрузке файла
focus	Приобретение элементом фокуса ввода
load	Завершение загрузки файла и подключенных к нему файлов
resize	Изменение размера документа
scroll	Прокрутка документа или элемента
afterprint	Закрытие окна предварительного просмотра выводимого на печать документа или начало вывода документа на печать
beforeprint	Открытие окна предварительного просмотра выводимого на печать документа или завершение подготовки документа к печати
beforeunload	Окно, документ и подключенные к нему файлы подготовлены к выгрузке
hashchange	Изменение части URL-адреса, следующей за символом #
pagehide	Браузер покидает страницу в истории просмотра
pageshow	Браузер переходит на страницу в истории сеанса

Окончание табл. 11.2

Событие	Условие возникновения
popstate	Изменение элемента истории просмотра в активном сеансе
unload	Выгрузка документа или включенного в него файла

Существуют также другие спецификации, в которых определены другие возможные события. Например, спецификация File API определяет ряд событий, связанных с загрузкой файлов, а спецификация HTML5 Media — события, связанные с воспроизведением аудио- и видеофайлов. Как видите, в вашем браузере происходит (или может происходить) множество событий.

Чтобы ознакомиться с полным перечнем событий, посетите следующий сайт:

`https://developer.mozilla.org/en-US/docs/Web/Events`

Обработка событий

Выполнение в JavaScript определенных действий в ответ на события называется *обработкой событий*.

За последние годы производители браузеров реализовали несколько способов, позволяющих программам на JavaScript обрабатывать события. В результате этого события JavaScript стали одной из тех областей, в которых приходится сталкиваться с несовместимостью браузеров.

В наши дни язык JavaScript достиг в своем развитии той точки, когда устаревшие, неэффективные методы обработки событий вскоре будут забыты. Тем не менее в силу того, что эти устаревшие методики по-прежнему находят широкое применение, важно не оставить их без внимания.

Встроенные обработчики событий

Первая из систем обработки событий была введена одновременно с выпуском первых версий JavaScript. Она основана на использовании специальных атрибутов обработчиков событий, включая обработчик событий `onclick`.

Встроенные атрибуты обработчиков событий образуются путем добавления префикса `on` к имени события. Их использование сводится к добавлению атрибута события в HTML-элемент. Когда происходит указанное событие, выполняется JavaScript-код, записанный в виде значения атрибута. В качестве примера в листинге 11.1 приведен код, который отображает окно сообщения после щелчка на ссылке.

Листинг 11.1. Подключение обработчика событий `onclick` к ссылке посредством встраивания в код

```
<a href="home.html" onclick=
        "alert('Перейти на главную страницу!');">
        Щелкните здесь для перехода на главную страницу</a>
```

Если вы поместите эту разметку в HTML-документ и щелкнете на ссылке, то на экране отобразится окно сообщения с текстом "Перейти на главную страницу!". Когда вы закроете это окно, выполнится заданный по умолчанию обработчик событий для этого элемента, который осуществит переход по ссылке, указанной в атрибуте `href`.

Во многих случаях может оказаться желательным, чтобы действие по умолчанию, связанное с данным элементом, не было выполнено. Например, что если вы захотите, чтобы закрытие окна сообщения не сопровождалось никакими другими действиями?

Для предотвращения выполнения действий, заданных по умолчанию, программисты на JavaScript разработали несколько приемов. Один из них состоит в том, чтобы сделать действие по умолчанию нерезультативным. Например, замена значения атрибута `href` на # приведет к тому, что ссылка будет указывать на саму себя.

```
<a href="#" onclick=
        "alert('Перейти на главную страницу!'); return false">
        Щелкните здесь</a>
```

Обработка событий с использованием свойств элементов

Одним из наибольших недостатков, связанных с использованием устаревшей техники встраивания обработчиков событий в элементы, является то, что при этом нарушается главный принцип наилучшей практики программирования, а именно отделение кода представления (ответственного за внешний вид объектов) от функционального кода (ответственного за выполнение полезных действий). Смешивание обработчиков событий с тегами HTML затрудняет сопровождение веб-страниц и их отладку, одновременно затрудняя понимание кода.

В версии 3 браузера Netscape была введена новая модель событий, позволяющая программистам подключать события к элементам в виде свойств. В листинге 11.2 приведен пример того, как работает эта модель.

Листинг 11.2. Подключение событий к элементам с помощью свойств

```
<html>
<head>
  <title>Приложение для счета</title>
  <script>
  // дождаться окончания загрузки окна и лишь потом
  // зарегистрировать событие onclick
  window.onload = initializer;
  // создать глобальную переменную для счетчика
  var theCount = 0;
  /**
  Регистрирует событие onclick
  */
  function initializer(){
  document.getElementById("incrementButton").onclick = increaseCount;
  }
  /**
  Инкрементирует theCount и отображает результат
  */
```

```
  function increaseCount(){
  theCount++;
  document.getElementById("currentCount").innerHTML = theCount;
  }
  </script>
</head>
<body>
  <h1>Щелкните на кнопке для счета</h1>
  <p>Текущее значение: <span id="currentCount">0</span></p>
  <button id="incrementButton">Инкрементировать счетчик</button>
</body>
</html>
```

Относительно листинга 11.2 следует сделать одно замечание: за именами функций, назначаемых обработчику событий, не следуют круглые скобки. Такой синтаксис не означает вызова функции и лишь указывает на то, что эта функция должна выполняться всякий раз, когда происходит данное событие. Если вы добавите круглые скобки после имени функции, то функция выполнится, а возвращаемый ею результат будет назначен обработчику события `onclick`, что явно не входит в ваши намерения.

Обработка событий с использованием метода `addEventListener()`

Метод `addEventListener()` прослушивает события на любом DOM-узле и запускает выполнение заданных действий в зависимости от наступившего события. Когда запускается функция, указанная для обработчика события в качестве выполняемого действия, она автоматически получает единственный аргумент — объект `Event`. В соответствии с общепринятым соглашением для этого аргумента используется имя `e`.

По сравнению с использованием атрибутов DOM-событий метод `addEventListener()` обладает следующими преимуществами:

- ✓ одному элементу может быть назначено несколько обработчиков событий;
- ✓ он работает на любом узле DOM, а не только на элементах;
- ✓ когда он активизируется, он предоставляет вам большую степень контроля над обработкой события.

Пример использования метода `addEventListener()` приведен в листинге 11.3. В основе примера лежит та же функция счета, что и в листинге 11.2, но добавлен второй обработчик событий для кнопки, который с каждым щелчком увеличивает размер шрифта, используемого для вывода значения счетчика.

Листинг 11.3. Назначение события с помощью метода `addEventListener()`

```
<html>
<head>
  <title>Приложение для счета</title>
  <script>
  // дождаться окончания загрузки окна и лишь потом
```

```
  // зарегистрировать событие onclick
  window.addEventListener('load',registerEvents,false);
  // создать глобальную переменную для счетчика
  var theCount = 0;
  /**
  Регистрирует событие onclick
  */
  function registerEvents(e){
document.getElementById("incrementButton").addEventListener
          ('click',increaseCount,false);
document.getElementById("incrementButton").addEventListener
          ('click',changeSize,false);
  }

  /**
  Инкрементирует theCount и отображает результат
  */
  function increaseCount(e){
  theCount++;
  document.getElementById("currentCount").innerHTML =
          theCount;
  }
  /**
  Изменяет размер шрифта для вывода значения счетчика
  */
  function changeSize(e){
  document.getElementById("currentCount").style.fontSize =
          theCount;
  }
  </script>
</head>
<body>
  <h1>Щелкните на кнопке для счета</h1>
  <p>Текущее значение: <span id="currentCount">0</span></p>
  <button id="incrementButton">Инкрементировать счетчик</button>
</body>
</html>
```

На рис. 11.1 показано, как будет выглядеть страница, созданная с помощью листинга 11.3, после того, как вы проведете немало времени, щелкая на кнопке.

Метод `addEventListener()` реализован с использованием трех аргументов.

Первый аргумент — это тип события. В отличие от двух других способов обработки событий, метод `addEventListener()` требует указания лишь имени события без префикса `on`.

Второй аргумент — это функция, которая должна вызываться при наступлении события. Как и в случае способа, основанного на использовании свойств событий, очень важно не записывать скобки после имени функции, чтобы обработчику события назначалась именно функция, а не результат ее выполнения.

Третий аргумент — это булево значение (`true` или `false`), которое определяет очередность выполнения обработчиков событий в тех случаях, когда у элемента, с которым связано событие, имеется родительский элемент, также связанный с событием.

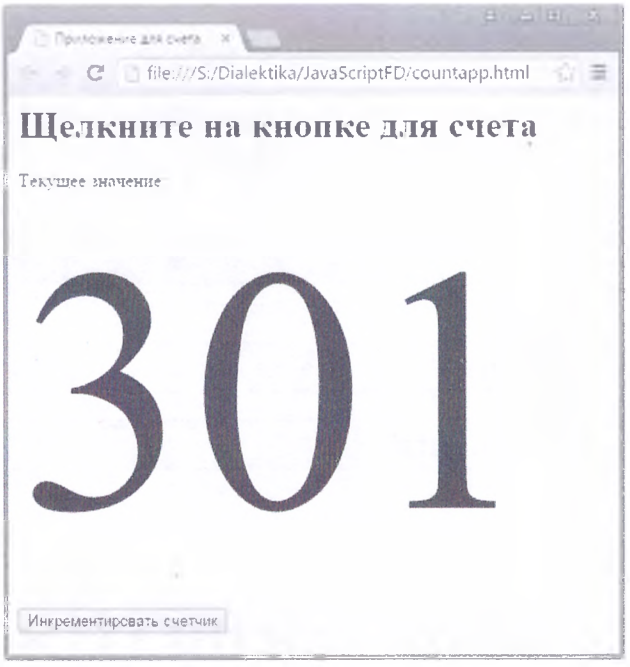


Рис. 11.1. Подключение двух событий к одному и тому же элементу расширяет ваши возможности!

В случае вложенных элементов очень важно знать, какое из двух событий произойдет первым. Суть проблемы иллюстрирует рис. 11.2, где на щелчки реагирует не только внутренний круг, но и внешний квадрат. Если щелкнуть на внутреннем круге, то какое событие должно произойти первым: связанное с квадратом или с кругом?

Большинство людей сказали бы, что первым должно считаться событие щелчка, выполненного на внутреннем круге. Однако компания Microsoft, следуя своей стратегии обработки событий в браузере Internet Explorer, решила, что первым должно считаться внешнее событие (щелчок на квадрате).

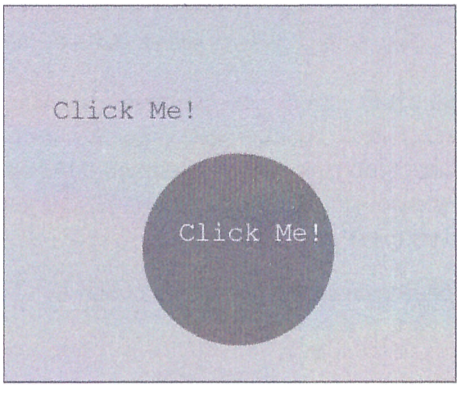


Рис. 11.2. Вложенные элементы

Наиболее распространенной стратегией обработки событий в ситуациях наподобие той, которая изображена на рис. 11.2, является модель *всплытия событий*. События, относящиеся к наиболее глубоко вложенным элементам, происходят первыми, а затем "всплывают" в направлении наружных элементов. Чтобы использовать эту стратегию, следует установить для последнего аргумента метода `addEventListener()` значение `false`, которое является также значением по умолчанию.

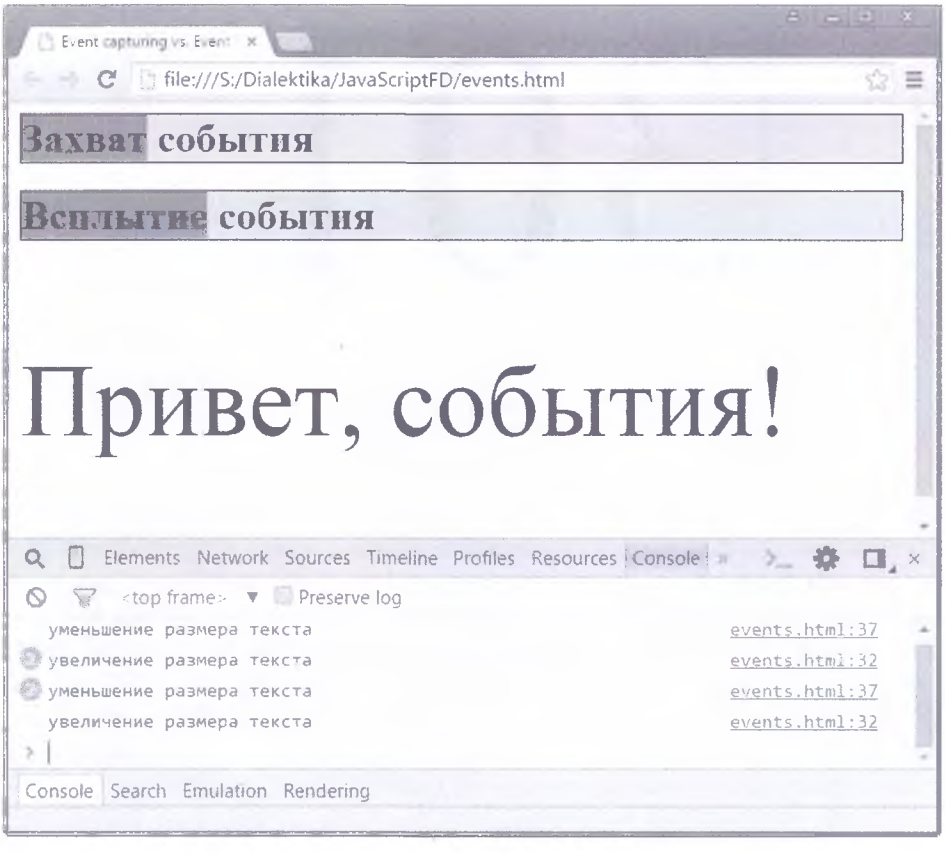


Рис. 11.3. Обработка вложенных событий

Другой способ обработки подобных сценариев называется *захватом событий*. В этом случае первыми происходят внешние события, а последними — внутренние.

В листинге 11.4 приведен пример, демонстрирующий, как важно знать очередность выполнения обработчиков событий. С элементами `h1` связано событие щелчка, но это же событие связано и с текстом заголовка.

Листинг 11.4. Демонстрация захвата и всплытия событий

```
<html>
<head>
  <title>Захват и всплытие событий</title>
  <style>
```

```
  #theText {font-size: 18px;}
  h1 {
  border:1px solid #000;
  background-color: #dadada;
  }
  #capEvent, #bubEvent {
  background-color: #777;
  }
  </style>
  <script>
  // дождаться окончания загрузки окна и лишь потом
  // зарегистрировать событие
  window.addEventListener('load',registerEvents,false);
  /**
  Регистрирует событие
  */
  function registerEvents(e){
document.getElementById("capTitle").addEventListener
          ('click',makeTiny,true);
document.getElementById("capEvent").addEventListener
          ('click',makeHuge,true);
document.getElementById("bubTitle").addEventListener
          ('click',makeTiny,false);
document.getElementById("bubEvent").addEventListener
          ('click',makeHuge,false);
  }
  function makeHuge(e){
  console.log("увеличение размера текста");
  document.getElementById("theText").style.fontSize = "80px";
  }
  function makeTiny(e){
  console.log("уменьшение размера текста");
  document.getElementById("theText").style.fontSize = "10px";
  }
  </script>
</head>
<body>
  <h1 id="capTitle"><span id="capEvent">
  Захват</span> события</h1>
  <h1 id="bubTitle"><span id="bubEvent">
  Всплытие</span> события</h1>
  <p id="theText">Привет, события!</p>
</body>
</html>
```

Отмена распространения событий

Помимо стратегий всплытия и захвата событий, вложенные события можно обрабатывать третьим способом: обработать единственное событие и на этом остановиться. Вы можете отключить механизмы всплытия и захвата для некоторого события (или даже для всех событий), используя метод `stopPropagation()`.

Если распространение событий в вашем сценарии не требуется, то целесообразно вообще отключить его, поскольку обработка всплытия и захвата событий требуют затрат системных ресурсов и могут замедлить работу вашего сайта.

В листинге 11.5 показано, как отключить распространение событий.

Листинг 11.5. Отмена распространения событий

```
function load(e){
  if (!e) var e = window.event;
  // установить cancelBubble для IE 8 и более ранних версий
  e.cancelBubble = true;

  if (e.stopPropagation) e.stopPropagation();

document.getElementById("capTitle").addEventListener
          ('click',makeTiny,true);
document.getElementById("capEvent").addEventListener
          ('click',makeHuge,true);
document.getElementById("bubTitle").addEventListener
          ('click',makeTiny,false);
document.getElementById("bubEvent").addEventListener
          ('click',makeHuge,false);
}
```

Глава 12

Интеграция ввода и вывода данных

В этой главе...

- Работа с формами
- Ввод данных
- Отправка выходных данных

"Никогда не отвечайте на вопрос, пока не выясните точно, кто и почему его задает и куда дальше пойдет эта информация".

Скотт Адамс

Обработка вводимых пользователем данных и пересылка их на сервер — вот основные и необходимые задачи любой компьютерной программы. В этой главе вы узнаете о том, как совместная работа JavaScript и HTML обеспечивает получение и отправку данных.

HTML-формы

Получение данных от пользователей веб-приложений осуществляется главным образом посредством HTML-форм. Формы предоставляют веб-разработчикам возможность создавать текстовые поля, раскрывающиеся списки, переключатели, флажки и кнопки. Используя CSS-стили, можно согласовывать внешний вид формы с конкретным стилем оформления сайта. JavaScript позволяет улучшать функциональность форм.

Элемент `form`

Любая HTML-форма содержится в элементе `form`. Элемент `form` играет роль контейнера, в котором хранятся поля для ввода данных, кнопки, флажки и ярлыки, образующие область пользовательского ввода. Этот контейнер функционирует подобно любому другому контейнерному элементу, такому как `div`, `article` или `selector`. Но он включает также атрибуты, которые сообщают браузеру о том, что нужно сделать с пользовательскими данными, введенными в полях формы.

В листинге 12.1 приведен пример HTML-формы с двумя полями ввода и кнопкой отправки формы.

Листинг 12.1. Пример HTML-страницы, содержащей форму

```
<html>
<head>
  <title>HTML-форма</title>
</head>
<body>

  <form action="subscribe.php" name="newsletterSubscribe"
        method="post">
    <label for="firstName">Имя: </label>
    <input type="text" name="firstName"
        id="firstName"><br>
    <label for="email">Email: <input type="text"
        name="email" id="email"></label><br>
    <input type="submit" value="Подпишитесь на нашу газету!">
  </form>

</body>
</html>
```

На рис. 12.1 показано, как будет выглядеть эта форма, если открыть ее в браузере.

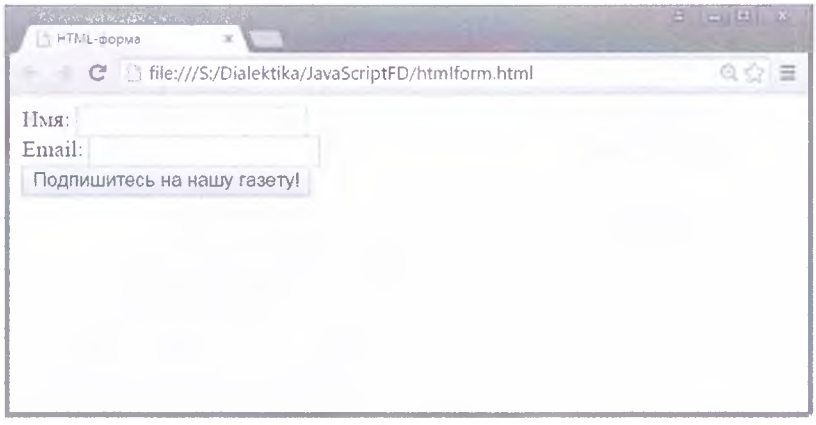


Рис. 12.1. HTML-форма

В предыдущем примере элемент `form` имеет следующие три атрибута.

- ✓ `action`. Сообщает браузеру о том, что требуется сделать с пользовательским вводом. Во многих случаях необходимым действием является выполнение сценария на стороне сервера.
- ✓ `name`. Это имя, назначенное программистом данной форме. Атрибут `name` удобно использовать для доступа к форме с помощью DOM.
- ✓ `method`. Принимает одно из двух возможных значений — `get` или `post`, которые соответственно указывают на то, должен ли браузер отправлять данные на сервер в составе URL-адреса или в составе HTTP-заголовка.

Кроме этих трех атрибутов, элемент `form` может содержать ряд других атрибутов, перечень которых приводится ниже.

- ✓ `accept-charset`. Указывает кодировку символов, приемлемую для сервера. Если только вы не работаете с многоязычным контентом (но даже и в этом случае), этот атрибут можно безопасно опускать.
- ✓ `autocomplete`. Указывает, должен ли браузер использовать режим автозаполнения в полях `input` формы.
- ✓ `enctype`. Указывает тип содержимого, отправляемого формой на сервер. В случае форм, отправляющих только текстовые данные, этот атрибут должен иметь значение `text/html`. Отправке файла (например, выгрузке графики) соответствует значение `multipart/form-data`. По умолчанию для этого атрибута устанавливается значение `application/x-www-form-urlencoded`.
- ✓ `novalidate`. Булево значение, определяющее, будет ли выполняться браузером *валидация данных* — проверка корректности введенных пользователем данных при отправке формы. Если этот атрибут не указан, то проверка данных осуществляется по умолчанию.
- ✓ `target`. Указывает, должен ли отображаться ответ от сервера после того, как форма отправлена. Значению по умолчанию (`"_self"`) соответствует открытие ответа в том же окне браузера, в котором открыта форма. Альтернативный вариант — открытие ответа в новом окне (`"_blank"`).

Элемент `label`

С помощью элемента `label` вы можете связывать с полями ввода надписи к ним (ярлыки). Атрибут `for` этого элемента принимает значение атрибута `id` элемента, с которым должен быть связан ярлык.

```
<label for="firstName">Имя: </label>
<input type="text" name="firstName">
```

Другим возможным способом связывания ярлыка с полем формы является вложение поля формы в элемент `label`.

```
<label>Имя: <input type="text"
        name="firstName"></label>
```

Преимуществом этого метода является то, что он не требует наличия у поля ввода атрибута `id` (который часто лишь дублирует атрибут `name`).

Элемент `input`

Элемент `input` — один из самых фундаментальных HTML-элементов, связанных с формами. В зависимости от значения его атрибута `type` он заставляет браузер отображать (или не отображать) несколько типов полей `input`.

Как правило, типом элемента `input` является `"text"`, что соответствует созданию в браузере поля для ввода текста. Необязательный атрибут `option` позволяет назначить элементу значение по умолчанию, а атрибут `name` — имя. Последнее вместе со связанным с ним значением образует пару "имя–значение", к которой можно получать доступ с помощью DOM и которая пересылается вместе с остальными значениями формы на сервер при отправке формы.

В своей простейшей форме текстовое поле ввода имеет следующий вид:

```
<input type="text" name="streetAddress">
```

В HTML5 элемент `input` обогатился целым рядом новых возможных значений атрибута `type`. С их помощью веб-разработчик может более точно указать тип значения, которое должно быть предоставлено в поле ввода. Руководствуясь этими значениями, браузер может предоставлять полям ввода, требующим валидации данных, наиболее пригодные для них элементы управления, в результате чего повышается качество веб-приложений.

Возможно, вам кажется немного странным, что вместо непосредственного перехода к рассмотрению соответствующего кода на JavaScript в этой главе мы уделяем так много внимания возможностям HTML-форм. Однако формы — это именно та область, где HTML может реально снизить нагрузку, ложащуюся на плечи программистов. Поэтому очень важно, чтобы вы хорошо усвоили все то, что может быть реализовано с помощью форм посредством HTML.

Возможные значения атрибута `type` элемента `input` приведены в табл. 12.1.

Таблица 12.1. Возможные значения атрибута `type` элемента `input`

Значение	Описание
`button`	Кнопка, воспринимающая щелчки
`checkbox`	Флажок
`color`	Селектор цвета
`date`	Элемент управления датой (год, месяц, день)
`datetime`	Элемент управления для выбора даты и времени (год, месяц, день, часы, минуты, секунды и доли секунды в стандарте UTC — всемирного скоординированного времени)
`datetime-local`	Элемент управления для выбора даты и времени (год, месяц, день, часы, минуты, секунды и доли секунды без привязки к часовому поясу)
`email`	Текстовое поле для ввода адресов электронной почты
`file`	Поле для выбора файла и кнопка просмотра
`hidden`	Скрытое текстовое поле
`image`	Кнопка отправки формы, использующая изображение вместо кнопки, заданной по умолчанию
`month`	Элемент управления для выбора месяца

Окончание табл. 12.1

Значение	Описание
`number`	Поле для ввода чисел
`password`	Поле для ввода пароля
`radio`	Переключатель
`range`	Поле для выбора значения из заданного диапазона (например, с помощью ползунка)
`reset`	Кнопка сброса (очистки) формы
`search`	Текстовое поле, предназначенное для ввода поискового запроса
`submit`	Кнопка отправки формы
`tel`	Текстовое поле для ввода телефонных номеров
`text`	Элемент управления по умолчанию; предназначен для ввода однострочного текста
`time`	Элемент управления для ввода времени (без привязки к часовому поясу)
`url`	Текстовое поле для ввода URL
`week`	Элемент управления для выбора месяца и года (без привязки к часовому поясу)

На момент написания книги не все браузеры поддерживали полный набор значений атрибута `type` элемента `input`. Использование значения атрибута `type`, не распознаваемого браузером, приведет лишь к тому, что отобразится обычное поле для ввода текста.

Элемент `select`

HTML-элемент `select` определяет раскрывающийся список или список с одиночным или множественным выбором. Элемент `select` содержит элементы `option`, предоставляющие пользователю возможные варианты выбора (листинг 12.2).

Листинг 12.2. Раскрывающийся список формы, созданный с помощью элемента `select`

```
<select name="favoriteColor">
  <option value="red">красный</option>
  <option value="blue">синий</option>
  <option value="green">зеленый<option>
</select>
```

На рис. 12.2 показано, как выглядит форма, созданная с помощью разметки из листинга 12.2.

Элемент `textarea`

Элемент управления `textarea` предназначен для ввода многострочного текста.

```
<textarea name="description" rows="4"
          cols="30"></textarea>
```

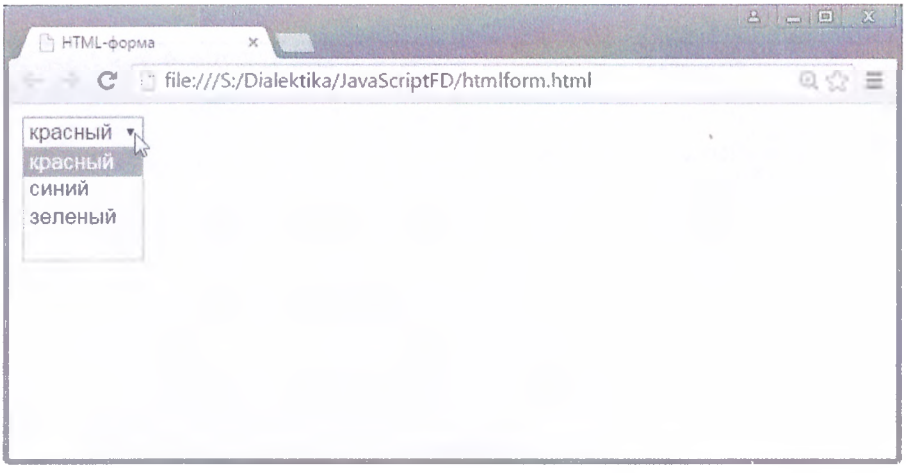


Рис. 12.2. Раскрывающийся список HTML

Элемент `button`

Элемент `button` предоставляет еще один способ создания кнопки, реагирующей на щелчки:

```
<button name="myButton">Щелкните на кнопке</button>
```

Элемент `button` можно использовать вместо элементов `input`, для атрибута `type` которых установлено значение `'submit'`. Его также можно использовать в тех случаях, когда вам нужна обычная кнопка, щелчок на которой не сопровождается отправкой формы.

Если вы не хотите, чтобы после щелчка на кнопке происходила отправка формы, в кнопку следует добавить атрибут `type` со значением `'button'`.

Работа с объектом Form

В HTML DOM для представления форм используется объект `Form`. Посредством этого объекта можно получать и устанавливать значения полей формы, управлять действиями, выполняемыми при отправке пользователем формы, а также изменять поведение формы.

Использование свойств объекта `Form`

Свойства объекта `Form` соответствуют рассмотренным ранее атрибутам HTML-элемента `form`. Они используются для получения и установки значений атрибутов HTML-элемента `form` с помощью JavaScript. Полный перечень свойств объекта `Form` приведен в табл. 12.2.

DOM-объекты являются представлением HTML-страниц. Их назначение — предоставить вам доступ (в терминологии программистов — *программный интерфейс*) к различным частям документа посредством JavaScript. В сочетании с DOM JavaScript обеспечивает возможность доступа к любому элементу содержимого HTML-документа.

Таблица 12.2. Свойства объекта `Form`

Свойство	Использование
`acceptCharset`	Получает или устанавливает список кодировок символов, поддерживаемых сервером
`action`	Получает или устанавливает значение атрибута `action` элемента `form`
`autocomplete`	Получает или устанавливает режим автоматического заполнения элементов `input` браузером
`encoding`	Сообщает браузеру, как должны кодироваться данные формы (в виде текста или файла). Это свойство — синоним свойства `enctype`
`enctype`	Сообщает браузеру, как должны кодироваться данные формы (в виде текста или файла)
`length`	Получает количество элементов управления в форме
`method`	Получает или устанавливает метод HTTP, используемый браузером для отправки формы
`name`	Получает или устанавливает имя формы
`noValidate`	Указывает на то, что данные формы не должны проверяться при их отправке на сервер
`target`	Указывает место для отображения результатов отправки формы

Использование атрибута `autocomplete`

Атрибут `autocomplete` HTML-элемента `form` устанавливает значение по умолчанию для режима автозаполнения элементов `input` формы. Если вы хотите, чтобы браузер обеспечивал функциональность автозаполнения для всех элементов `input`, установите для этого атрибута значение `'on'`. Если же вы хотите самостоятельно выбирать, к каким элементам должна применяться эта функциональность, или если ваш документ предоставляет собственную функциональность автозаполнения (посредством JavaScript), установите значение `'off'` и при необходимости устанавливайте режим автозаполнения для отдельных элементов `input`, используя их атрибуты `autocomplete`.

Способы получения и установки значений свойств формы рассмотрены в главе 10. Используя ссылку на форму, вы сможете получить доступ к любому ее свойству с помощью точечной или скобочной нотации.

Для получения значения свойства `name` первой формы в документе можно использовать следующую инструкцию:

```
document.getElementByTagName("form")[0].name
```

Более распространенный способ доступа к форме предполагает назначение ей атрибута `id` и последующее использование метода `getElementById()` для ее выбора.

DOM предоставляет другой, более удобный способ доступа к формам: коллекции форм. Коллекция форм позволяет получать доступ к формам в документе двумя способами, которые описаны ниже.

- ✓ **По индексу.** Когда в документе создаются элементы `form`, им присваиваются индексы, отсчет которых начинается с нуля. Для доступа к первой форме используйте выражение `document.forms[0]`.
- ✓ **По имени.** К формам также можно получать доступ, используя значение атрибута `name` элемента `form`. Например, для получения значения свойства `action` формы с именем `"subscribeForm"` следует использовать инструкцию `document.forms.subscribeForm.action`. Кроме того, можно воспользоваться скобочной нотацией, применив инструкцию `document.forms["subscribeForm"].action`.

Использование методов объекта `Form`

Объект `Form` имеет два метода: `reset()` и `submit()`.

Метод `reset()`

Метод `reset()` отменяет любые изменения, внесенные в полях формы после загрузки страницы, и восстанавливает их значения по умолчанию. Он делает то же самое, что и кнопка `reset` в HTML, которая создается путем установки значения `"reset"` для атрибута `type` элемента `input`:

```
<input type="reset" value="Очистить форму">
```

Метод `submit()`

Метод `submit()` вынуждает форму отправить свои данные на сервер в соответствии со значениями свойств формы (`action`, `method`, `target` и др.). Этот метод делает то же самое, что и кнопка `submit` в HTML, создаваемая путем установки значения `"submit"` для атрибута `type` элемента `input`:

```
<input type="submit" value="Отправить форму">
```

Использование методов `reset()` и `submit()` наряду с некоторыми свойствами объекта `Form` продемонстрировано в листинге 12.3.

Листинг 12.3. Использование свойств и методов объекта `Form`

```
<html>
<head>
  <title>Подпишитесь на нашу газету!</title>
  <script>
    function setFormDefaults(){
      document.forms.subscribeForm.method = "post";
      document.forms.subscribeForm.target = "_blank";
      document.forms.subscribeForm.action =
          "http://watzthis.us9.list-manage.com/subscribe/
          post?u=1e6d8741f7db587af747ec056&amp;
          id=663906e3ba";
```

```
      // регистрация событий кнопки
      document.getElementById('btnSubscribe').
          addEventListener('click', submitForm);
      document.getElementById('btnReset').
          addEventListener('click', resetForm);
    }
    function submitForm() {
      document.forms.subscribeForm.submit();
    }
    function resetForm() {
      document.forms.subscribeForm.reset();
    }
  </script>
</head>
<body onload="setFormDefaults();">
  <form name="subscribeForm">
    <h2>Подпишитесь на наш список рассылки</h2>
    <label for="mce-EMAIL">Адрес email </label>
    <input type="email" value="" name="EMAIL" id="mce-
          EMAIL">
    <button type="button" id="btnSubscribe">Subscribe!
          </button>
    <button type="button" id="btnReset">Reset</button>
  </form>
</body>
</html>
```

Доступ к элементам формы

JavaScript предлагает несколько способов доступа к элементам `input` формы и их значениям. Однако не все они равноценны, и среди программистов существуют различные точки зрения относительно того, какой из них является лучшим. Ниже приведен перечень этих методик с описанием их преимуществ и недостатков.

- ✓ **Использование индексов формы и ее полей `input`.** Например, чтобы получить доступ к первому полю `input` в первой форме, можно использовать следующий код:

  ```
  document.forms[0].elements[0]
  ```

  Старайтесь не прибегать к этой методике, поскольку в ней предполагается, что структура документа и порядок следования элементов в форме остаются неизменными. Если кто-то решит, что поле `email` должно предшествовать первому полю `name`, то весь ваш сценарий разрушится.

- ✓ **Использование имени формы и имен ее полей `input`.** Например:

  ```
  document.myForm.firstName
  ```

  Преимуществом этой методики является простота ее синтаксиса и способа использования. Она поддерживается всеми браузерами (и поддерживалась уже на самых ранних этапах разработки DOM).

- ✓ **Использование метода `getElementById()` для выбора формы и имен элементов `input` для их выбора.** Например:

  ```
  document.getElementById("myForm").firstName
  ```

  Эта методика требует, чтобы форме был назначен атрибут `id`. Например, предыдущему коду соответствовал бы элемент `input` с именем `firstName`, принадлежащий следующей форме.

  ```
  <form id="myForm" action="myaction.php">
  . . .
  </form>
  ```

- ✓ **Использование уникального значения атрибута `id` для непосредственного доступа к полю.** Например:

  ```
  document.getElementById("firstName")
  ```

  Используя эту методику, не забывайте о том, что в тех случаях, когда на одной странице имеется несколько форм, каждая из них должна иметь уникальное значение атрибута `id` (впрочем, значения атрибута `id` в любом случае должны быть уникальными, поэтому на самом деле никакой дополнительной проблемы здесь не существует).

Получение и установка значений элементов формы

DOM предоставляет доступ к именам и значениям элементов формы через свойства `name` и `value`.

Получение и установка значений полей `input` формы продемонстрировано в листинге 12.4 на примере простого калькулятора.

Листинг 12.4. Демонстрация получения и установки значений полей `input` формы на примере простого калькулятора

```
<html>
<head>
  <title>Математическая забава</title>
  <script>

    function registerEvents() {
      document.mathWiz.operate.addEventListener('click',
            doTheMath,false);
    }

    function doTheMath() {
      var first =
            parseInt(document.mathWiz.numberOne.value);
      var second =
            parseInt(document.mathWiz.numberTwo.value);
      var operator = document.mathWiz.operator.value;

      switch (operator){
        case "add":
          var answer = first + second;
```

```
          break;
        case "subtract":
          var answer = first - second;
          break;
        case "multiply":
          var answer = first * second;
          break;
        case "divide":
          var answer = first / second;
          break;
      }

      document.mathWiz.theResult.value = answer;
    }
  </script>
</head>
<body onload="registerEvents();">
  <form name="mathWiz">
  <label>Первое число: <input type="number"
         name="numberOne"></label><br>
    <label>Второе число: <input type="number"
         name="numberTwo"></label><br>
    <label>Оператор:
      <select name="operator">
        <option value="add"> + </option>
        <option value="subtract"> - </option>
        <option value="multiply"> * </option>
        <option value="divide"> / </option>
      </select>
    </label>
    <br>
    <input type="button" name="operate" value="Вычислить!"><br>
    <label>Результат: <input type="number" name="theResult">
         </label>
  </form>
</body>
</html>
```

Проверка пользовательского ввода

Одно из наиболее распространенных применений JavaScript — это проверка, или *валидация*, данных, введенных в форму, прежде чем они будут отправлены на сервер. Это обеспечивает дополнительную защиту от проникновения некорректных или потенциально небезопасных данных в веб-приложение. Кроме того, в случае ввода неверных данных пользователь немедленно извещается о допущенной ошибке.

В HTML5 некоторые наиболее типичные задачи проверки данных решаются с помощью HTML-атрибутов. Однако в силу несовместимости браузеров по-прежнему целесообразно проверять отправляемые пользователем данные с помощью JavaScript.

В простой программе калькулятора, приведенной в табл. 12.4, для полей операндов был установлен числовой тип (`number`), чтобы ввод пользователем нечисловых

значений в этих полях предотвращался браузером. Однако, поскольку тип `number` введен для полей `input` сравнительно недавно, вы не можете всегда рассчитывать на то, что браузер поддерживает эту возможность, и поэтому валидация данных с помощью JavaScript имеет важное значение.

В листинге 12.5 приведен пример организации проверки данных с помощью сценария JavaScript. Здесь важно обратить ваше внимание на то, что в качестве действия для формы определена функция, осуществляющая проверку данных. Метод `submit()` формы выполняется лишь после того, как эта функция удостоверится в корректности введенных данных.

Кодом, творящим истинные чудеса в процессе проверки данных функцией `validate()`, является следующая строка довольно необычного вида:

```
if (/^\d+$/.test(first) && /^\d+$/.test(second)) {
```

Символы, заключенные между двумя косыми чертами / и /, образуют так называемое *регулярное выражение*. Регулярное выражение — это шаблон поиска, составленный из символов, которые представляют группы других символов. В данном случае мы используем регулярное выражение, с помощью которого проверяем, являются ли оба введенных пользователем значения числами. Более подробно о регулярных выражениях рассказано в главе 14.

Проверка корректности введенных пользователем данных с помощью JavaScript настолько распространена, что для решения этой задачи было разработано множество различных методик. Прежде чем "изобретать велосипед", выполните поиск в Интернете по ключевой фразе "валидация данных с помощью JavaScript", и вы найдете подсказки, которые позволят вам сэкономить массу времени и намного улучшить функциональность своего приложения.

Листинг 12.5. Валидация данных с помощью JavaScript

```
<html>
<head>
  <title>Математическая забава</title>
  <script>

    function registerEvents() {
      document.mathWiz.operate.addEventListener('click',
            validate,false);
    }

    function validate() {
      var first = document.mathWiz.numberOne.value;
      var second = document.mathWiz.numberTwo.value;
      var operator = document.mathWiz.operator.value;

      if (/^\d+$/.test(first) && /^\d+$/.test(second)) {

      doTheMath();
```

```
    } else {

      alert("Ошибка: Оба операнда должны быть числами!");
    }
  }

  function doTheMath() {
    var first =
          parseInt(document.mathWiz.numberOne.value);
    var second =
          parseInt(document.mathWiz.numberTwo.value);
    var operator = document.mathWiz.operator.value;

    switch (operator){
      case "add":
        var answer = first + second;
        break;
      case "subtract":
        var answer = first - second;
        break;
      case "multiply":
        var answer = first * second;
        break;
      case "divide":
        var answer = first / second;
        break;
    }

    document.mathWiz.theResult.value = answer;
  }
  </script>
</head>
<body onload="registerEvents();">
  <div id="formErrors"></div>
  <form name="mathWiz">
  <label>Первое число: <input type="number"
        name="numberOne"></label><br>
    <label>Второе число: <input type="number"
        name="numberTwo"></label><br>
    <label>Оператор:
      <select name="operator">
        <option value="add"> + </option>
        <option value="subtract"> - </option>
        <option value="multiply"> * </option>
        <option value="divide"> / </option>
      </select>
    </label>
    <br>
    <input type="button" name="operate" value="Вычислить!"><br>
    <label>Результат: <input type="number" name="theResult">
        </label>
  </form>
</body>
</html>
```

Глава 13

Работа с CSS и графикой

В этой главе...

- Изменение стилей
- Использование изображений
- Создание анимации в JavaScript
- Демонстрация слайд-шоу

"Стремясь к обретению стиля, начните с отказа от подражания".
Элвин Брукс Уайт

Теперь, когда вы понимаете, как манипулировать DOM-объектами с помощью JavaScript, вы в состоянии создавать не просто статичные веб-документы, а интерактивные приложения, способные реагировать на вводимые пользователем данные, изменяться без перезагрузки и доставлять активное содержимое на различные компьютерные устройства.

Использование объекта Style

Определенный в DOM объект `Style` — это мощный инструмент, обеспечивающий изменение внешнего вида веб-страниц и их адаптацию к пользовательскому вводу или текущим условиям в браузере в режиме реального времени. С помощью объекта `Style` программисты получают доступ к стилевым свойствам CSS любого выбранного элемента или коллекции элементов в документе. (Более подробно об основных правилах и синтаксисе CSS см. в главе 1.)

Вот лишь некоторые из операций, возможных за счет применения CSS-стилей:

- ✓ изменение цвета текста для выделения ключевых слов, введенных в окне поиска;
- ✓ анимация объекта, на котором выполнен щелчок мышью;
- ✓ изменение цвета границы и фона для той части формы, в которую пользователь в настоящее время вносит изменения;
- ✓ развертывание и свертывание или сокрытие и отображение различных элементов страницы;
- ✓ создание всплывающих подсказок, появляющихся над содержимым веб-страницы после щелчка пользователем на ссылке.

Объект `Style` работает подобно любому другому DOM-объекту. Он включает набор свойств, которые могут быть использованы для получения или установки различных аспектов выбранного элемента.

Свойства объекта `Style` являются зеркальным отражением свойств CSS. Различие между ними состоит лишь в том, что при записи свойств DOM-объекта `Style` составные слова не разделяются дефисами, как в формате CSS, а располагаются слитно с использованием так называемого "верблюжьего" формата, т.е. каждое слово начинается с прописной буквы.

В табл. 13.1 приведены некоторые из наиболее часто используемых свойств объекта `Style`, а также CSS-свойства, которые они позволяют изменять.

Таблица 13.1. Часто используемые свойства объекта `Style` и их CSS-эквиваленты

Свойство	Стиль CSS	Описание
`backgroundColor`	`background-color`	Получает или устанавливает цвет фона элемента
`borderWidth`	`border-width`	Устанавливает ширину всех четырех границ элемента
`fontFamily`	`font-family`	Получает или устанавливает список имен семейств шрифтов, назначенных тексту в элементе
`lineHeight`	`line-height`	Получает или устанавливает междустрочный интервал в тексте
`textAlign`	`text-align`	Получает или устанавливает вид горизонтального выравнивания текста в пределах элемента

Полный список свойств объекта `Style`, а также свойств всех других DOM-объектов можно найти по адресу `http://overapi.com/html-dom`.

Получение текущего стиля элемента

Объект `Style` возвращает текущие встроенные стили, назначенные элементу. Он не может ничего сообщить о том, какой фактический стиль будет использован браузером при визуализации данного элемента, поскольку не включает внешних стилей, хранящихся в CSS-файлах, или внедренных стилей, заданных в элементах `style`.

По этой причине использование объекта `Style` — не лучший способ получения стилей элементов. В листинге 13.1 элементу `div` назначен встроенный стиль и несколько стилевых правил, определенных в элементе `style`.

Когда объект `Style` используется для получения стилевых свойств элемента, он возвращает лишь стили, установленные с помощью JavaScript, или встроенные CSS-стили.

Листинг 13.1. Неудачный способ получения текущего стиля элемента

```
<html>
<head>
  <title>Получение встроенных стилей</title>
  <style>
    #myText {
      color: white;
```

```
      background-color: black;
      font-family: Arial;
      margin-bottom: 20px;
    }
    #stylesOutput {
      font-size: 18px;
      font-family: monospace;
    }
  </style>
  <script>
    function getElementStyles(e){
      var colorOutput = "цвет: " + e.target.style.color;
      var fontSizeOutput = "размер шрифта: " + e.target.style.
          fontSize;
      document.getElementById("stylesOutput").innerHTML =
          colorOutput + "<br>" + fontSizeOutput;
  }
  </script>
</head>
<body>
  <div id="myText" style="font-size: 26px;"
          onclick="getElementStyles(event);">Некоторый
          текст.</div>
  <div id="stylesOutput"></div>
</body>
</html>
```

На рис. 13.1 показано, что можно увидеть, если загрузить эту страницу и щелкнуть на элементе `div`.

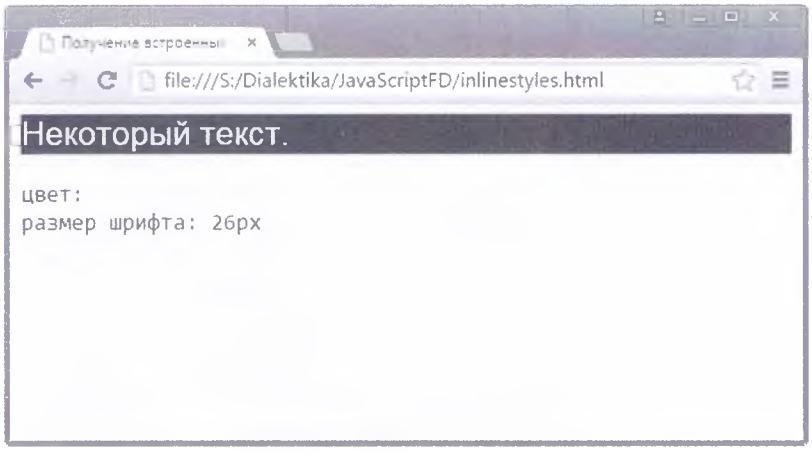


Рис. 13.1. Результат использования объекта `Style` для получения стиля элемента

Относительно результатов работы этого сценария следует сделать два важных замечания.

- ✓ В то время как в заголовке с помощью CSS для элемента `div` установлен цвет `white`, значение свойства `color` объекта `Style` остается пустым.
- ✓ Тем не менее значение свойства `fontSize` объекта `Style` устанавливается корректно, поскольку свойство `font-size` было установлено с помощью встроенного CSS-стиля.

Свойства объекта `Style` ведут себя подобно встроенным стилям и извлекают лишь значения встроенных стилей, назначенных элементу.

Для получения текущего стиля элемента целесообразно использовать метод `window.getComputedStyle()` (листинг 13.2).

Листинг 13.2. Корректный способ получения текущего стиля элемента

```
<html>
<head>
  <title>Получение вычисленных стилей</title>
  <style>
    #myText {
      color: white;
      background-color: black;
      font-family: Arial;
      margin-bottom: 20px;
    }
    #stylesOutput {
      font-size: 18px;
      font-family: monospace;
    }
  </style>
  <script>
    function getElementStyles(e){
      var computedColor =
          window.getComputedStyle(e.target).
          getPropertyValue("color");
      var computedSize = window.getComputedStyle
          (e.target).getPropertyValue("font-size");

      var colorOutput = "цвет: " + computedColor;
      var fontSizeOutput = "размер шрифта: " + computedSize;
      document.getElementById("stylesOutput").innerHTML =
          colorOutput + "<br>" + fontSizeOutput;
    }
  </script>
</head>
<body>
  <div id="myText" style="font-size: 26px;"
          onclick="getElementStyles(event);">Некоторый
          текст.</div>
  <div id="stylesOutput"></div>
</body>
</html>
```

На рис. 13.2 показаны результаты выполнения в браузере кода, приведенного в листинге 13.2, который получает корректные значения свойства `style`.

Заметьте, что в листинге 13.2 функция `getPropertyValue()` принимает в качестве аргумента CSS-свойство (`font-size`), а не свойство `style` (`fontSize`). Таким образом, сценарий запрашивает значение свойства `font-size` непосредственно у элемента, а не через объект `Style` (который предоставляет информацию лишь о встроенных стилях).

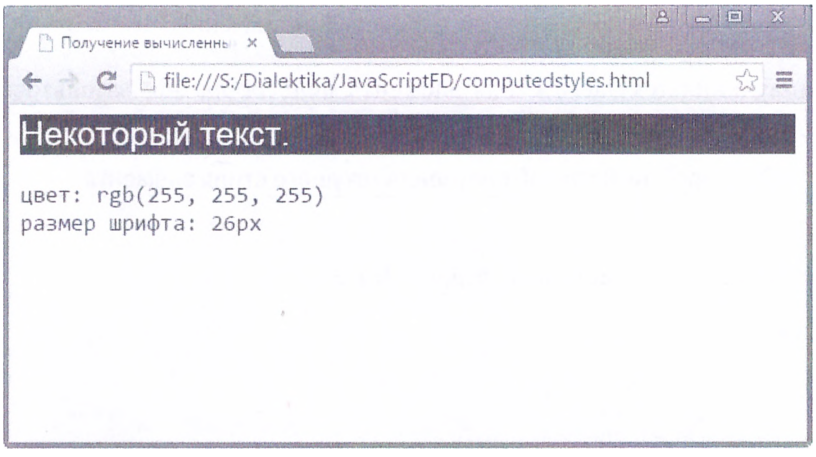


Рис. 13.2. Результат получения стиля элемента с помощью объекта `Style`

Установка стилевых свойств

Чтобы задать свойства объекта `Style`, следует сначала выбрать элемент, к которому необходимо применить новый стиль, а затем назначить ему новый стиль с помощью объекта `Style`, используя точечную или скобочную нотацию.

Для изменения ширины границы элемента с идентификатором `"borderWidth"` можно использовать следующую инструкцию:

```
document.getElementById("borderedSquare").style.borderWidth = "15px";
```

Анимация элементов с помощью объекта Style

С помощью CSS можно управлять не только внешним видом элементом, но и их позиционированием. Изменяя свойство `style` в цикле JavaScript, можно без особых усилий создавать простые анимации.

В листинге 13.3 функция JavaScript перемещает квадрат по экрану, используя цикл `for` для изменения CSS-свойства `'left'`.

Листинг 13.3. Анимация элемента с использованием объекта `Style`

```
<html>
<head>
  <title>Анимация с помощью JavaScript</title>
  <style>
    #square {
      width: 100px;
      height: 100px;
      background-color: #333;
      position: absolute;
      left: 0px;
      top: 100px;
    }
</style>
<script>
    function moveSquare() {
      for (i=0; i<500; i++){
        document.getElementById("square").style.left =
          i+"px";
      }
    }
</script>
</head>
<body onload="moveSquare();">
<div id="square"></div>
</body>
</html>
```

После запуска этого сценария в браузере может создаться впечатление, будто он открылся с уже завершенной анимацией. На самом деле анимация начинает выполняться лишь после открытия сценария, но все происходит настолько быстро, что заметить это невозможно (разве что у вас очень медленный компьютер или необычайно зоркие глаза).

Чтобы человеческий взгляд мог проследить за анимацией, необходимо ввести задержку на каждой итерации цикла. Самый распространенный подход, позволяющий создать задержку в цикле, заключается в использовании метода `setTimeOut()` объекта `Window`.

Метод `setTimeOut()` принимает два аргумента:

- ✓ функцию или выполняемый код;
- ✓ длительность периода задержки перед выполнением (в миллисекундах).

Поместив вызов в функцию и вызывая функцию рекурсивно, мы получаем возможность контролировать скорость анимации. (Более подробно о написании рекурсивных функций см. в главе 7.)

В листинге 13.4 квадрат движется гораздо медленнее, чем прежде, — со скоростью 1 пиксель за 1/100 секунды. Этот пример отличается от листинга 13.3 еще двумя усовершенствованиями.

- ✓ Теперь квадрат способен реагировать на щелчки: щелчок на нем запускает анимацию.
- ✓ Анимация квадрата зависит от позиции, в которой он находится во время выполнения щелчка. Щелчок на квадрате приводит к его перемещению на 75 пикселей вправо от его текущей позиции.

Листинг 13.4. Анимация элемента с использованием объекта `Style`, метода `setTimeOut()` и рекурсии

```
<html>
<head>
  <title>Анимация средствами JavaScript</title>
  <style>
    #square {
      width: 100px;
      height: 100px;
      background-color: #333;
      position: absolute;
      left: 0px;
      top: 50px;
  }
</style>
<script>
    // дождаться, когда загрузится окно
    window.addEventListener('load',initialize,false);

  function initialize(){

    // переместить квадрат после щелчка на нем
    document.getElementById("square").
          addEventListener('click',function(e){
      // получить начальную позицию
      var left = window.getComputedStyle(e.target).
          getPropertyValue("left");

      // преобразовать left в десятичное целое число
      left = parseInt(left, 10);
      moveSquare(left,100);
     }, false);
  }

  function moveSquare(left,numMoves) {
    document.getElementById("square").style.left =
          left+"px";

    if (numMoves > 0) {
      numMoves--;
      left++;
      setTimeout(moveSquare,10,left,numMoves);
    } else {
```

```
      return;
    }
  }
  </script>
</head>
<body>
  <div id="square"></div>
</body>
</html>
```

На рис. 13.3 показаны результаты, полученные при запуске листинга 13.4 в браузере.

Присмотритесь внимательно к коду, ответственному за регистрацию события щелчка на квадрате. В качестве обработчика событий используется анонимная функция. На первый взгляд, этот код выглядит довольно необычно, но если вы разложите его на базовые составляющие, то увидите, что он работает в точности так, как и метод `addEventListener()` с его тремя аргументами: типом события, слушателем события (в данном случае это анонимная функция) и булевым значением, которое управляет захватом событий.

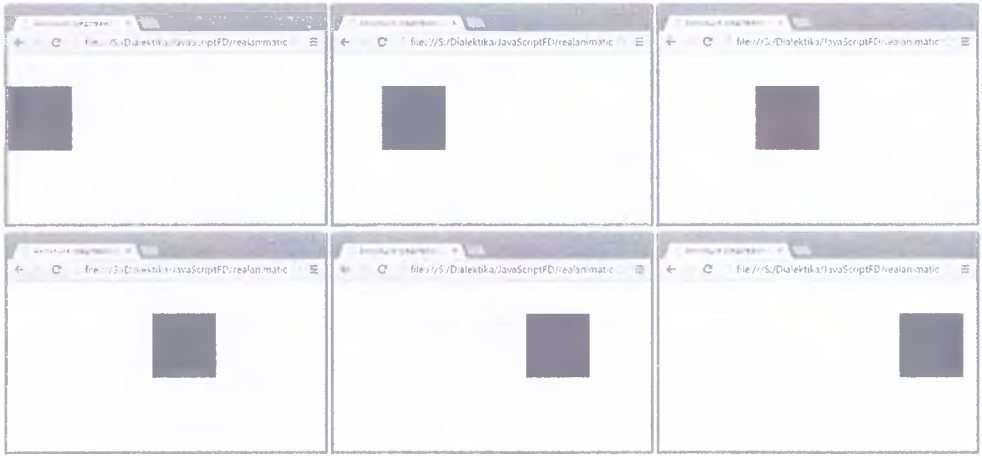


Рис. 13.3. Запуск анимации с помощью JavaScript в ответ на щелчок мышью

Работа с изображениями

Обычно HTML-элементы `img` статичны и неизменны, кроме, разумеется, тех случаев, когда изображением является анимация. Манипулируя атрибутами элемента `img` и изменяя CSS-стили с помощью JavaScript, можно создавать всевозможные эффекты, такие как перемещение изображений и изменение их размеров, эффекты свечения, эффекты трансформации объектов при прохождении над ними курсора и многое другое.

Использование объекта Image

DOM-объект Image обеспечивает доступ к свойствам HTML-элемента img. Имея такую возможность, вы можете изменять атрибуты элементов, получая и устанавливая значения свойств объекта Image, которые представлены в табл. 13.2.

Наиболее важными и чаще всего используемыми свойствами объекта Image являются свойства src, width и height. Используя эти три свойства, вы можете создавать эффекты смены изображений, изумительные эффекты масштабирования графики, впечатляющие трансформируемые кнопки и множество других эффектов!

Создание трансформируемых кнопок

Трансформируемая кнопка — это кнопка, способная изменяться тем или иным образом при наведении на нее указателя мыши. Трансформируемые кнопки — отличный способ показать пользователю, что данное изображение способно реагировать на щелчки. Кроме того, вы можете использовать их для предоставления более подробной информации о том, что произойдет, если щелкнуть на кнопке или ссылке. Некоторые веб-дизайнеры любят прятать так называемые *пасхальные яйца* на своих сайтах, которые инициируют изменение изображения или иных элементов при прохождении над ними указателя мыши или после щелчка над некоторыми скрытыми областями страницы.

Таблица 13.2. Свойства объекта Image

Свойство	Описание
alt	Получает или устанавливает значение атрибута alt элемента изображения
complete	true, если браузер завершил загрузку изображения
height	Получает или устанавливает значение атрибута height изображения
isMap	Получает или устанавливает логическое значение, указывающее на то, является ли данное изображение частью карты изображений на стороне сервера
naturalHeight	Получает или устанавливает исходное значение высоты изображения
naturalWidth	Получает или устанавливает исходное значение ширины изображения
src	Получает или устанавливает значение атрибута src изображения
useMap	Получает или устанавливает значение атрибута useMap изображения
width	Получает или устанавливает значение атрибута width изображения

Трансформируемые кнопки можно создавать исключительно средствами CSS, однако для создания более сложных эффектов или смены изображений требуется использование JavaScript или же JavaScript в сочетании с CSS.

В листинге 13.5 показано, как создать простой эффект трансформируемого изображения с помощью JavaScript.

Листинг 13.5. Эффект трансформируемого изображения

```
<html>
<head>
  <title>Трансформируемое изображение</title>
  <script>

    function swapImage(imgToSwap){
      imgToSwap.src = "button2.png";
      imgToSwap.alt = "Указатель над  кнопкой!";
    }
    function swapBack(imgToSwap){
      imgToSwap.src = "button1.png";
      imgToSwap.alt = "Наведите на меня указатель!";
    }
  </script>
</head>
<body>
  <img src="button1.png" id="myButton"
          onmouseover="swapImage(this);"
          onmouseout="swapBack(this);"
          alt="Наведите на меня указатель!">
</body>
</html>
```

Для нормальной работы страницы, созданной на основе листинге 13.5, требуется, чтобы в одной папке с HTML-файлом находились изображения `button1.png` и `button2.png`. Вы можете создать собственные изображения или же загрузить готовые изображения, которые найдете по адресу `http://www.codingjsfordummies.com/code/`.

Увеличение размеров изображения при наведении на него указателя мыши

Другой полезный трюк, который поможет сделать интерфейс вашего веб-приложения более дружественным по отношению к пользователю, — это незначительное увеличение кнопки при наведении на нее указателя мыши. Это небольшое ухищрение позволяет улучшить интерактивность приложения и указать пользователю на то, что данный элемент управления реагирует на щелчки.

В листинге 13.6 представлен видоизмененный код из листинга 13.5, который увеличивает размер изображения на 5 процентов каждый раз, когда на него перемещается указатель мыши.

Листинг 13.6. Увеличение размеров изображения при наведении на него указателя мыши

```
<html>
<head>
  <title>Изменение размера изображения</title>
```

```
  <script>

    function growImage(imgToGrow){
      imgToGrow.width += imgToGrow.width * .05;
      imgToGrow.height += imgToGrow.width * .05;
    }
    function restoreImage(imgToShrink){
      imgToShrink.width = imgToShrink.naturalWidth;
      imgToShrink.height = imgToShrink.naturalHeight;
    }
  </script>
</head>
<body>
  <img src="button1.png" id="myButton"
         onmouseover="growImage(this);" onmouseout="rest
         oreImage(this);" alt="Наведите на меня указатель!">
</body>
</html>
```

Возможно, вы заметили, что в листингах 13.5 и 13.6 используются встроенные методы обработки событий. При разработке реальных веб-приложений такой подход не является идеальным, однако встроенные обработчики событий часто используют, и это вполне оправданно, для создания простых эффектов наподобие рассмотренных, которые связаны фактически с интерфейсом, а не с функциональностью.

Создание слайд-шоу

Слайд-шоу (демонстрация слайдов) — это популярный метод демонстрации последовательности изображений в одном разделе сайта. Они часто используются на главных страницах сайтов для их оживления.

В слайд-шоу для смены изображений часто используют эффекты перехода. Обычно для создания эффектов перехода пользуются библиотеками функций JavaScript наподобие jQuery. Однако для создания простых эффектов перехода вполне можно обойтись обычными средствами JavaScript, CSS и DOM. Чтобы упростить пример, мы ограничились в слайд-шоу, представленном в листинге 13.7, лишь сменой изображений без применения эффектов перехода.

Листинг 13.7. Слайд-шоу, созданное на основе JavaScript и CSS

```
<html>
<head>
  <title>Слайд-шоу средствами JavaScript</title>

  <style>
    #carousel {
      position: absolute;
      width: 800px;
      height: 400px;
```

```
      top: 100px;
      left: 100px;
      display: hidden;
    }

  </style>
  <script>
    var slides = [
      "<div id='slide1'>мой первый слайд<br><img
          src='image1.jpg'></div>",
      "<div id='slide2'>мой второй слайд<br><img
          src='image2.jpg'></div>",
      "<div id='slide3'>мой третий слайд<br><img
          src='image3.jpg'></div>"];

    var currentSlide = 0;
    var numberOfSlides = slides.length-1;

    window.addEventListener("load",loader,false);

    function loader(){
      changeImage();
    }

    function changeImage(){
      console.log("функция changeImage()");
      if (currentSlide > numberOfSlides){
        currentSlide = 0;
      }

      document.getElementById("carousel").
          innerHTML=slides[currentSlide];

      console.log('отображается слайд ' + currentSlide +
          " из " + numberOfSlides);
      currentSlide++;

      setTimeout(changeImage,1000);
    }

  </script>
</head>
<body>
  <div id="carousel"></div>
</body>
</html>
```

Использование анимационных свойств объекта Style

В CSS3 и DOM-объекте `Style` предусмотрены свойства, упрощающие анимацию элементов. Их совместное использование позволяет создавать впечатляющие анимации с минимальными усилиями. Анимационные свойства объекта `Style` приведены в табл. 13.3.

Таблица 13.3. Свойства объекта `Style`, связанные с анимацией

Свойство	Описание
`animation`	Устанавливает одновременно все анимационные свойства, за исключением свойства `animationPlayState`
`animationDelay`	Получает или устанавливает длительность задержки перед началом анимации
`animationDirection`	Получает или устанавливает режим воспроизведения в обратном направлении для некоторых или всех циклов
`animationDuration`	Получает или устанавливает время, отводимое для завершения одного цикла анимации
`animationFillMode`	Получает или устанавливает стили, которые должны применяться к элементу, когда анимация уже закончилась
`animationIterationCount`	Получает или устанавливает количество повторений анимации
`animationName`	Получает или устанавливает список анимаций, используя правила `@keyframes`
`animationTimingFunction`	Получает или устанавливает кривую скорости, описывающую развитие анимации во времени
`animationPlayState`	Получает или устанавливает режим выполнения или приостановки анимации

В листинге 13.8 приведен пример простой анимации, созданной средствами CSS. Сначала с помощью CSS конфигурируются скорость и ключевые кадры анимации, а в дальнейшем управление приостановкой и возобновлением анимации осуществляется с помощью JavaScript. При творческом подходе можно придумать множество различных способов управления этой анимацией с помощью JavaScript.

Предоставляемые в CSS3 средства анимации сравнительно новы, и не все браузеры в равной степени поддерживают соответствующие возможности. Поскольку эти средства все еще считаются экспериментальными, некоторые браузеры требуют указания специальных префиксов перед именем анимационного свойства.

В листинге 13.8 код включает как стандартные, так и снабженные префиксом стили CSS.

Листинг 13.8. Управление анимацией CSS3 с помощью JavaScript

```
<!DOCTYPE html>
<html>
<head>
  <style>
    #words {

      position: relative;
      width: 300px;
      height: 200px;
      text-align: center;
      padding-top: 20px;
      font-family: Arial;
      border-radius: 6px;
      color: white;

      /* Chrome, Safari, Opera */
      -webkit-animation-name: movewords;
      -webkit-animation-duration: 6s;
      -webkit-animation-timing-function: linear;
      -webkit-animation-delay: 0s;
      -webkit-animation-iteration-count: infinite;
      -webkit-animation-direction: alternate;
      -webkit-animation-play-state: running;
      /* Стандартный синтаксис */
      animation-name: movewords;
      animation-duration: 6s;
      animation-timing-function: linear;
      animation-delay: 0s;
      animation-iteration-count: infinite;
      animation-direction: alternate;
      animation-play-state: running;
    }

    /* Chrome, Safari, Opera */
    @-webkit-keyframes movewords {
      0%   {background:red; left:100px; top:0px;}
      25%  {background:blue; left:200px; top:100px;}
      50%  {background:blue; left:300px; top:200px;}
      75%  {background:blue; left:200px; top:200px;}
      100% {background:red; left:100px; top:0px;}
    }

    /* Стандартный синтаксис */
    @keyframes movewords {
      0%   {background:red; left:100px; top:0px;}
      25%  {background:blue; left:200px; top:100px;}
      50%  {background:blue; left:300px; top:200px;}
      75%  {background:blue; left:200px; top:200px;}
      100% {background:red; left:100px; top:0px;}
    }
  </style>
<script>
```

```
    window.addEventListener("load",registerEvents,false);

    function registerEvents(e){
      document.getElementById("stop").
          addEventListener("click",stopAni,false);
      document.getElementById("go").
          addEventListener("click",startAni,false);
    }
    function stopAni(){
      document.getElementById("words").style.
          webkitAnimationPlayState = "paused";
      document.getElementById("words").style.
          AnimationPlayState = "paused";

    }
    function startAni(){
        document.getElementById("words").style.
            webkitAnimationPlayState = "running";
        document.getElementById("words").style.
            AnimationPlayState = "running";
    }


</script>

</head>
<body>

  <h1 id="words">Перемещение по странице</h1>

  <button type="button" id="stop">Пауза</button>
  <button type="button" id="go">Продолжить</button>

</body>
</html>
```

Часть IV

Дополнительные темы

В этой части...

- ✓ Поиск с использованием регулярных выражений
- ✓ Функции обратного вызова и замыкания
- ✓ Технологии AJAX и JSON

Глава 14

Поиск с использованием регулярных выражений

В этой главе...

- Поиск по шаблонам с помощью регулярных выражений
- Написание регулярных выражений
- Использование регулярных выражений в JavaScript

"Создать проблему нетрудно. Мы делаем это постоянно. А вот поиск надежных и результативных решений требует усилий и настойчивости".

Генри Роллинз

Регулярные выражения — мощное инструментальное средство, включенное во многие языки программирования, которое облегчает поиск и изменение текста в документе по заданному шаблону. Поначалу синтаксис регулярных выражений может показаться вам чрезвычайно сложным, но разобравшись в сути дела, вы сможете делать с текстом все, что захотите.

Поиск текста с помощью регулярных выражений

Регулярные выражения — это способ поиска шаблонов или символьных комбинаций в текстовых строках.

Приведем некоторые из возможных примеров применения регулярных выражений:

- ✓ проверка введенного пользователем адреса электронной почты, гарантирующая соблюдение правильного формата;
- ✓ поиск и замена всех вхождений имени человека в статье;
- ✓ поиск всех слов, не являющихся первыми или последними словами предложений, по всей книге;
- ✓ поиск в документе всех числовых строк, которые могут быть телефонными номерами.

Вот какой вид может иметь регулярное выражение:

```
^((\(\d{3}\) ?)|(\d{3}-))?\d{3}-\d{4}$
```

Выглядит довольно устрашающе, не правда ли? Не огорчайтесь. Очень скоро вы узнаете, как расшифровывать подобные выражения, и поймете, что приведенная строка предназначена для поиска телефонных номеров в формате, принятом в США и Канаде:

```
(555)555-5555
```

Однако регулярные выражения не всегда бывают такими сложными, как в последнем примере. В листинге 14.1 приведен простой пример сценария, в котором используется регулярное выражение, а на рис. 14.1 показан результат выполнения этого сценария в браузере.

Листинг 14.1. Определить, включает ли строка слово "JavaScript"

```
<html>
<head>
  <title>Поиск JavaScript</title>
  <script>
    window.addEventListener("load",registerEvents,false);

    function registerEvents(e){
      document.getElementById("ask").
          addEventListener("click",findAnswer,false);
    }

    function findAnswer(){
      // получить пользовательский запрос
      var question = document.
          getElementById("userQuestion").value;
      /* создать новый объект регулярного выражения,
         которое ищет полное совпадения со строкой
         "JavaScript" */
      var re = new RegExp("JavaScript");

      // если в запросе найдена строка "JavaScript"
      if (re.test(question)===true){

        // вывести ответ
        document.getElementById("answer").innerHTML =
            "Вопросы по JavaScript? Обратитесь на сайт
             Coding with JavaScript For Dummies
             Криса Минника и Евы Холланд";
        // записать "JavaScript!" в консоль
        console.log("JavaScript!");
      }
    }
  </script>
</head>
<body>
```

```
  <form id="userInput">
    <label>Введите вопрос:
    <textarea id="userQuestion"></textarea>
    </label>
    <button id="ask" type="button">Получить ответ</button>
  </form>
  <div id="answer"/>
</body>
</html>
```

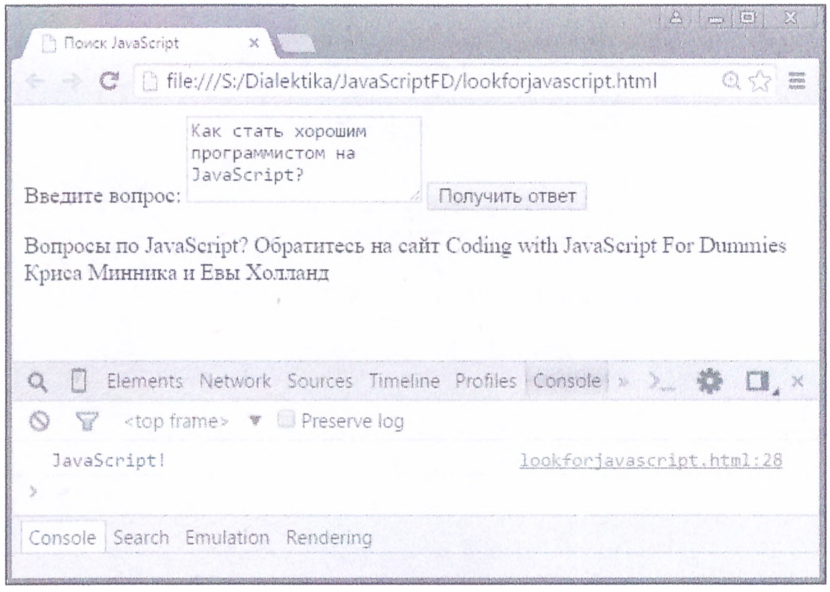


Рис. 14.1. Результат выполнения листинга 14.1 в браузере

Создание регулярных выражений

Прежде чем регулярное выражение можно будет использовать, необходимо создать объект, содержащий это выражение. Регулярное выражение можно записать одним из двух способов:

- ✓ в литеральной форме;
- ✓ посредством конструктора объекта `RegExp`.

Использование объекта `RegExp`

Если регулярное выражение создается путем вызова функции конструктора объекта `RegExp`, то результирующий объект будет создан на стадии выполнения сценария, а не на стадии загрузки. Используйте эту возможность в тех случаях, когда во время написания сценария значение регулярного выражения вам неизвестно. Например, можно запрашивать регулярное выражение у пользователя, получать его из внешнего источника или вычислять некоторую часть выражения во время выполнения.

В листинге 14.2 приведен пример программы, которая создает регулярное выражение на основе буквенных символов, выбираемых по случайному закону, а затем предлагает пользователю ввести целевое предложение. При отправке формы программа подсчитывает, сколько раз выбранная случайная буква встречается в пользовательском тексте.

Листинг 14.2. Создание регулярного выражения на стадии выполнения приложения с помощью объекта RegExp

```
<html>
<head>
  <title>Игра в подсчет букв</title>
  <script>
    window.addEventListener('load',loader,false);

    // получить случайную букву
    var letter = String.fromCharCode(97 +
        Math.floor(Math.random() * 26));

    /* Создать регулярное выражение, используя букву.
       Установить опцию g для нахождения всех вхождений. */
    var re = new RegExp(letter,'g');

    function loader(e){
      document.getElementById("getText").addEventListener(
            'submit',countLetter,false);
    }

    function countLetter(e){
      e.preventDefault();
      document.getElementById("results").innerHTML =
            "Была выбрана секретная буква " + letter +".";
      var userText = document.getElementById("userWords").
            value;
      var matches = userText.match(re);
      if (matches){
        var count = matches.length;
      } else {
       var count = 0;
      }

      document.getElementById("results").innerHTML +=
            " Вы ввели секретную букву " + count + " раз.";
    }
  </script>
</head>
<body>
  <form id="getText">
    <p>Я задумал букву! Введите предложение, и я скажу вам,
       сколько раз моя секретная буква встречается в вашем
       предложении!</p>
    <input type="text" name="userWords" id="userWords">
```

```
    <input type="submit" name="submit">
  </form>
  <div id="results"></div>
</body>
</html>
```

Результат выполнения этой программы в браузере показан на рис. 14.2.

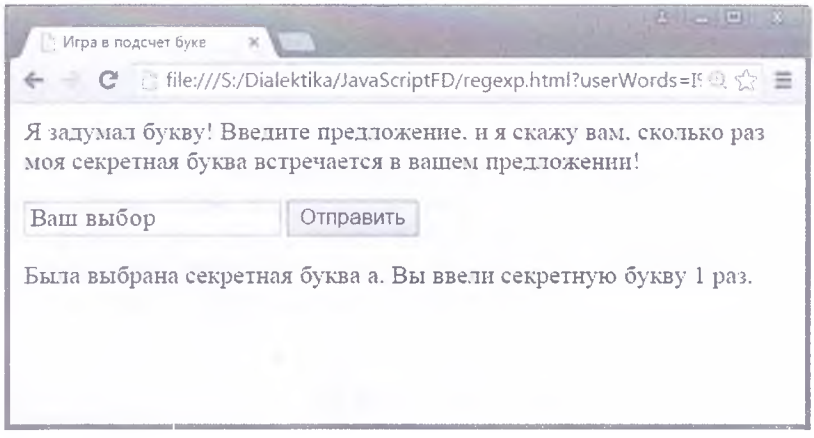


Рис. 14.2. Результат выполнения листинга 14.2 в браузере

Литеральные регулярные выражения

Чтобы создать литеральное регулярное выражение, необходимо окружить регулярное выражение символами косой черты, а не заключать его в кавычки. Например:

```
var myRegularExpression = /JavaScript/;
```

Литералы регулярных выражений компилируются браузером во время загрузки сценария и остаются неизменными на протяжении всего времени его жизни. Во многих случаях использование литеральных регулярных выражений способствует повышению производительности.

В предыдущем примере регулярное выражение используется для поиска точных совпадений со строкой `"JavaScript"`. Регулярное выражение, содержащее строку символов, для которой ищется точное соответствие, называется *простым шаблоном*.

В реальном приложении или программе вы должны предусмотреть, что пользователи могут вводить поисковые слова в различных формах написания. Например, пользователь может ввести любое из перечисленных ниже слов, со всей очевидностью подразумевая JavaScript:

- ✓ javascript
- ✓ Javascript
- ✓ java script
- ✓ JS
- ✓ js

Возможны еще более экзотические варианты. Одной из особенностей обработки данных, введенных реальными пользователями, является то, что вы не можете знать заранее, в каком именно виде будет введен ожидаемый текст! Чтобы распознавать видоизмененные варианты написания слов, в том числе и те, которые отличаются использованием прописных и строчных букв, можно использовать более сложные регулярные выражения, выполняющие поиск по шаблонам или наборам символов, а не просто по литеральной строке.

Ниже приведено переработанное регулярное выражение, которому будет соответствовать любая из строк `"JavaScript"`, `"Javascript"` и `"javascript"`:

```
var myRegularExpression = /[Jj]ava[Ss]cript/;
```

Ситуация начинает усложняться, но, разобравшись в том, что означают те или иные символы, вы увидите, что на самом деле все очень просто. Последовательность символов, заключенная в квадратные скобки, представляет *символьный набор*. Символьному набору соответствует любой одиночный символ из числа тех, которые в нем содержатся. Запись `[Jj]` говорит о том, что условию поиска будет соответствовать буква `j` в любом регистре.

Тестирование регулярных выражений

Иногда при написании регулярных выражений полезно иметь под рукой простой инструмент их тестирования, который гарантировал бы, что вы получаете именно то, что вам нужно. Существует ряд веб-сайтов и инструментов, позволяющих тестировать регулярные выражения. Один из таких сайтов — `http://regex101.com`. Чтобы воспользоваться им, введите регулярное выражение в верхнем текстовом поле и какой-нибудь текст в нижней текстовой области. Сайт сверит текст с заданным регулярным выражением и выделит обнаруженные совпадения.

На рис. 14.3 показана страница сайта `http://regex101.com` в процессе тестирования нашего примера регулярного выражения с использованием текста в виде вопроса, касающегося JavaScript.

Специальные символы в регулярных выражениях

Регулярные выражения делают возможным поиск цифр в строках, символов, групп символов, повторений символов и многого другого.

Для создания сложных шаблонов поиска можно использовать специальные символы регулярных выражений. Наиболее часто используемые символы регулярных выражений приведены в табл. 14.1.

Таблица 14.1. Специальные символы регулярных выражений

Специальный символ	Значение
\	Указывает, должен ли следующий символ интерпретироваться как специальный символ или как литеральный. Если следующий символ специальный, то \ указывает на то, что он должен интерпретироваться буквально
^	Находит начало строки

Специальный символ	Значение
`$`	Находит конец строки
`*`	Находит предшествующий символ, встречающийся 0 или несколько раз
`+`	Находит предшествующий символ, встречающийся 1 или несколько раз
`?`	Находит предшествующий символ, встречающийся 0 или 1 раз
`.`	Находит любой одиночный символ, кроме символа новой строки
`x|y`	Находит `x` или `y`
`{n}`	Находит ровно `n` следующих подряд вхождений предшествующего символа
`[xyz]`	Находит любой из символов, указанных в скобках
`[^xyz]`	Находит любой символ, кроме тех, которые указаны в скобках
`[\b]`	Находит символ забоя
`\b`	Находит границу слова
`\B`	Находит границу, не являющуюся границей слова
`\d`	Находит цифровой символ
`\D`	Находит любой нецифровой символ
`\n`	Находит символ перевода строки
`\s`	Находит одиночный пробельный символ, включая символы пробела, табуляции, подачи формы и перевода строки
`\S`	Находит одиночный непробельный символ
`\t`	Находит символ табуляции
`\w`	Находит любой буквенно-цифровой символ, включая символ подчеркивания
`\W`	Находит любой несловарный символ

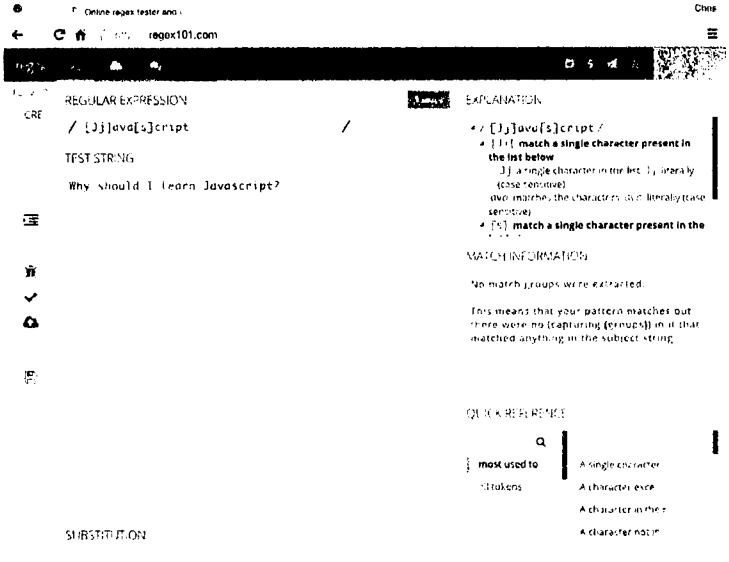


Рис. 14.3. Использование сайта `regex101.com` для тестирования регулярного выражения

Использование модификаторов

Существует возможность управлять некоторыми аспектами выполнения поиска с помощью *модификаторов*. Чтобы использовать модификаторы, передайте их в качестве второго аргумента функции конструктора `RegExp()` при создании объекта регулярного выражения или укажите их после замыкающего символа / литерального регулярного выражения.

Три таких модификатора описаны ниже.

- ✓ `g`. Устанавливает режим глобального поиска, при котором поиск выполняется в пределах всей строки, а не только до нахождения первого соответствия.
- ✓ `i`. Означает, что регистр символов в процессе поиска игнорируется.
- ✓ `m`. Устанавливает режим многострочного поиска. Например, в этом режиме специальные символы ^ и $ означают не начало и конец всего текста, а начало и конец каждой отдельной строки многострочного текста.

Приведенному ниже регулярному выражению будут соответствовать все приведенные выше вариации слова JavaScript во всем документе:

```
/javascript/ig
```

Использование регулярных выражений в коде

Регулярные выражения используются вместе с методами регулярных выражений и подмножеством функций для работы со строками (см. главу 3).

Для работы с регулярными выражениями предназначены следующие два метода.

- ✓ `test()`. Проверяет наличие совпадений с шаблоном, возвращая `true`, если совпадение найдено, и `false` в противном случае.
- ✓ `exec()`. Проверяет наличие совпадений с шаблоном и возвращает массив соответствующих данных.

Если необходимо узнать, содержит ли строка подстроку, соответствующую регулярному выражению, то следует использовать метод `test()`. Если же вы хотите узнать не только это, но и количество обнаруженных в строке соответствий шаблону, а также каким именно фрагментам текста они соответствуют, то следует использовать метод `exec()`.

Функции для работы со строками, допускающие использование регулярных выражений, приведены в табл. 14.2.

Хорошим и на удивление сложным примером использования регулярных выражений может служить проверка корректности адреса электронной почты. Любой такой адрес должен подчиняться ряду определенных правил. Основные правила следующие:

- ✓ адрес должен содержать только один символ @;
- ✓ адрес должен содержать символы как перед символом @, так и после него;
- ✓ адрес должен содержать по крайней мере одну точку, разделяющую группы символов, следующих за символом @.

Таблица 14.2. Функции для работы со строками, использующие регулярные выражения

Функция	Использование
`match()`	Просматривает строку в поиске совпадений с регулярным выражением. Возвращает массив данных о совпадениях или `null` в случае отсутствия совпадений
`search()`	Проверяет наличие совпадений в строке. В случае обнаружения совпадений возвращает индекс первого совпадения. Если совпадения не найдены, то возвращает `-1`
`replace()`	Ищет совпадения в строке и заменяет их заданной строкой
`split()`	Разбивает строку на массив подстрок, используя регулярное выражение или фиксированную строку

Существуют и другие правила, но полный учет всевозможных деталей, например, принятие во внимание того факта, что в некоторых ситуациях в адресах электронной почты могут встречаться пробелы или символы других языков, значительно усложняет задачу.

Если сайту требуется, чтобы пользователь ввел свой адрес электронной почты, то даже самая элементарная проверка способна значительно уменьшить вероятность отправки адреса в неверном формате.

В листинге 14.3 приведен пример сценария, проверяющего формат адресов электронной почты. После того как пользователь введет адрес и щелкнет на соответствующей кнопке, сценарий сверит адрес со следующим литеральным регулярным выражением:

```
/\b[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}\b/i
```

Это выражение начинается со специального символа `\b`, которому соответствует граница слова, т.е. начало нового слова. За этим символом следует шаблон

```
[A-Z0-9._%+-]+
```

Этому шаблону соответствует сочетание одной или нескольких букв и цифр, среди которых допускаются символы подчеркивания, знаки процента и дефисы.

```
@[A-Z0-9.-]+
```

Эта часть требует наличия символа @, за которым следует сочетание одной или нескольких букв, цифр или дефисов.

```
\.[A-Z]{2,4}\b/i
```

Завершающая часть регулярного выражения выполняет поиск строки длиной от двух до четырех символов (часть адреса наподобие `com`, `net` или `org`), за которой следует граница слова. В самом конце регулярного выражения используется

модификатор /i, указывающий на то, что в процессе поиска совпадений регистр символов будет игнорироваться.

В случае обнаружения совпадения считается, что данные прошли проверку, и во всплывающем окне выводится сообщение "Верно!".

Листинг 14.3. Сценарий для проверки формата адреса электронной почты

```
<html>
<head>
  <title>Проверка адреса электронной почты</title>
  <script>
    window.addEventListener('load',loader,false);
    function loader(e){
      e.preventDefault();
      document.getElementById('emailinput').
        addEventListener('submit',validateEmail,false);
    }

    function validateEmail(e) {
      var re = /\b[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}\b/i;
      if (re.test(e.target.yourEmail.value)) {
        alert("Верно!");
      } else {
        alert("Неверно!");
      }
    }

  </script>
</head>
<body>
  <form id="emailinput">
    <label>Введите адрес электронной почты:
      <input type="text" id="yourEmail">
    </label>
    <input type="submit" value="Проверить" id="validate">
  </form>
</body>
</html>
```

Результат выполнения листинга 14.3 в браузере представлен на рис. 14.4.

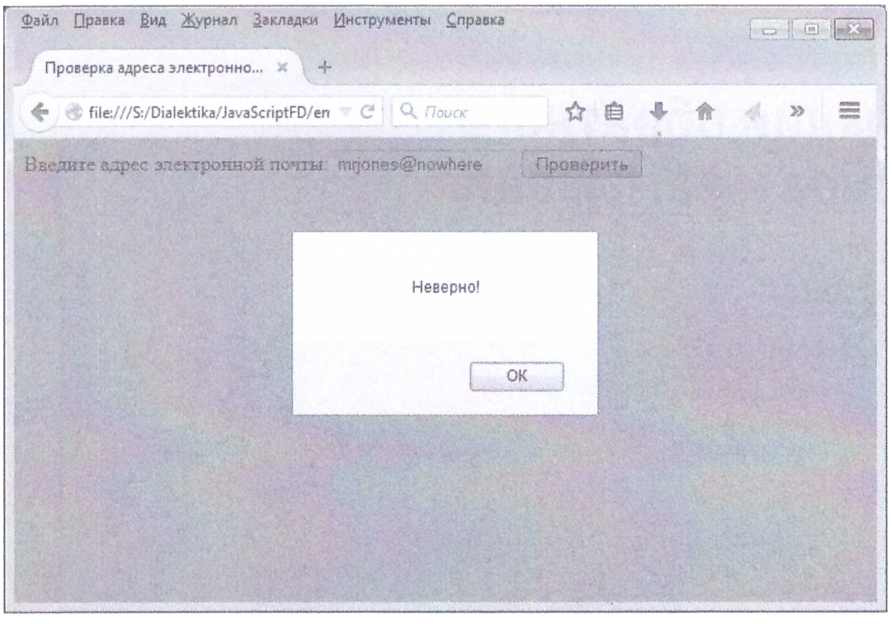


Рис. 14.4. Использование регулярного выражения в сценарии для проверки адреса электронной почты

Глава 15

Функции обратного вызова и замыкания

В этой главе...

- Функции обратного вызова
- Использование функций обратного вызова
- Создание замыканий

"Вели позвать вчерашний день обратно".
Уильям Шекспир

Функции обратного вызова (callbacks) и замыкания (closures) — одни из наиболее полезных и широко используемых в JavaScript программных инструментов. В этой главе вы узнаете о том, как передавать функции в качестве аргументов другим функциям и какие преимущества это дает.

Что такое функции обратного вызова

Как и любой другой объект, функции могут присваиваться переменным, передаваться в качестве аргументов другим функциям, а также создаваться и возвращаться другими функциями.

Функции в роли аргументов

Функция обратного вызова — это функция, которая передается в качестве аргумента другой функции. Эта техника возможна в JavaScript в силу того факта, что функции — это объекты.

Объекты функций содержат строку с кодом функции. Вызов функции, осуществляемый путем указания ее имени, за которым следует пара круглых скобок `()`, инициирует выполнение ее кода. Если же вы просто указываете имя функции или передаете его в качестве аргумента другой функции, не используя круглые скобки в конце, то функция не выполняется.

С примерами использования функций обратного вызова вы уже сталкивались в главе 11 в процессе применения метода addEventListener():

```
document.addEventListener('click',doSomething,false);
```

Данный метод получает в качестве аргументов событие (click) и объект Function (doSomething). Функция обратного вызова выполняется не сразу. Метод addEventListener() выполняет ее, когда происходит указанное событие.

Написание функций, использующих функции обратного вызова

В качестве примера рассмотрим функцию doMath(), принимающую функцию обратного вызова в качестве аргумента.

```
function doMath(number1,number2,callback) {
  var result = callback(number1,number2);
  document.write ("Результат: " + result);
}
```

Это обобщенная функция, которая возвращает результат выполнения любой операции, требующей двух операндов. Фактическая операция, которая должна быть выполнена, определяется функцией обратного вызова, передаваемой в качестве аргумента.

Вызов функции doMath() осуществляется путем передачи ей двух числовых аргументов и дополнительного третьего аргумента, определяющего функцию обратного вызова.

```
doMath(5,2,function(number1,number2){
  var calculation = number1 * number2 / 6;
  return calculation;
});
```

В листинге 15.1 приведен полный код веб-страницы, содержащий функцию doMath(), которая затем вызывается несколько раз с различными функциями обратного вызова.

Листинг 15.1. Вызов функции doMath() с различными функциями обратного вызова

```
<html>
<head>
  <title>Тестирование функции doMath</title>
  <script>
    function doMath(number1,number2,callback){

      var result = callback(number1,number2);
      document.getElementById("theResult").innerHTML +=
            ("Результат: " + result + "<br>");
    }

    document.addEventListener("DOMContentLoaded",
            function() {
```

```
      doMath(5,2,function(number1,number2){
        var calculation = number1 * number2;
        return calculation;
      });

      doMath(10,3,function(number1,number2){
        var calculation = number1 / number2;
        return calculation;
      });

      doMath(81,9,function(number1,number2){
        var calculation = number1 % number2;
        return calculation;
      });

    }, false);
  </script>
</head>
<body>
  <h1>Выполнение вычислений</h1>
  <div id="theResult"></div>
</body>
</html>
```

Результат выполнения листинга 15.1 в браузере представлен на рис. 15.1.

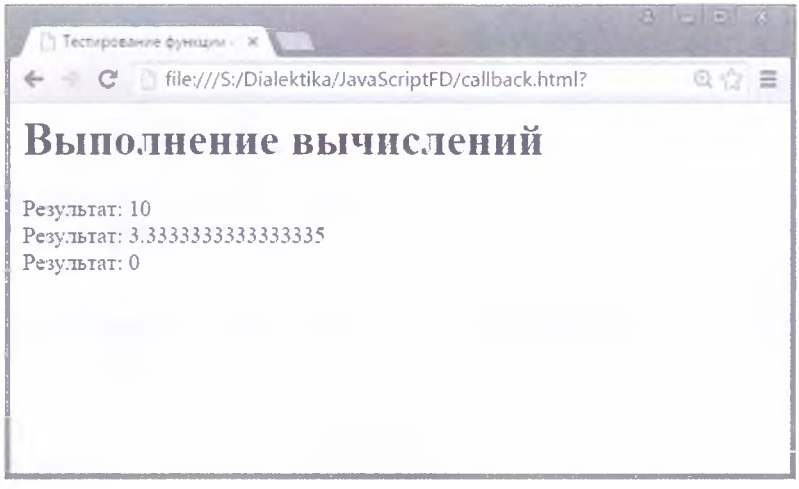


Рис. 15.1. Выполнение вычислений с использованием функций обратного вызова

Использование именованных функций обратного вызова

Во всех примерах, приведенных в предыдущем разделе, функции обратного вызова были анонимными. Существует также возможность определять именованные функции, имена которых можно затем передавать другим функциям в качестве функций обратного вызова.

Анонимная функция (см. главу 7) — это функция, которая создается без присвоения ей имени.

Использование именованных функций в качестве функций обратного вызова повышает удобочитаемость кода. Пример использования анонимной функции в качестве функции обратного вызова приведен в листинге 15.2. Кроме того, по сравнению с листингом 15.1 этот пример усовершенствован в двух отношениях:

- ✓ в функцию `doMath()` добавлена проверка, гарантирующая, что передаваемый аргумент действительно является функцией;
- ✓ прежде чем отобразить результат выполнения функции, программа выводит ее код.

Листинг 15.2. Использование именованных функций в качестве функций обратного вызова

```
<html>
<head>
  <title>Тестирование doMath с именованными функциями</title>
  <script>
    function doMath(number1,number2,callback){

      if (typeof callback === "function") {

      var result = callback(number1,number2);
      document.getElementById("theResult").innerHTML +=
          (callback.toString() + "<br><br>Результат: " +
          result + "<br><br>");
      }
    }

    function multiplyThem(number1,number2){
      var calculation = number1 * number2;
      return calculation;
    }

    function divideThem(number1,number2){
      var calculation = number1 / number2;
      return calculation;
    }
    function modThem(number1,number2){
      var calculation = number1 % number2;
      return calculation;
    }

    document.addEventListener("DOMContentLoaded",
          function() {

      doMath(5,2,multiplyThem);

      doMath(10,3,divideThem);
```

```
      doMath(81,9,modThem);

    }, false);
  </script>
</head>
<body>
  <h1>Выполнение вычислений</h1>
  <div id="theResult"</div>
</body>
</html>
```

Результат выполнения листинга 15.2 в браузере представлен на рис. 15.2.

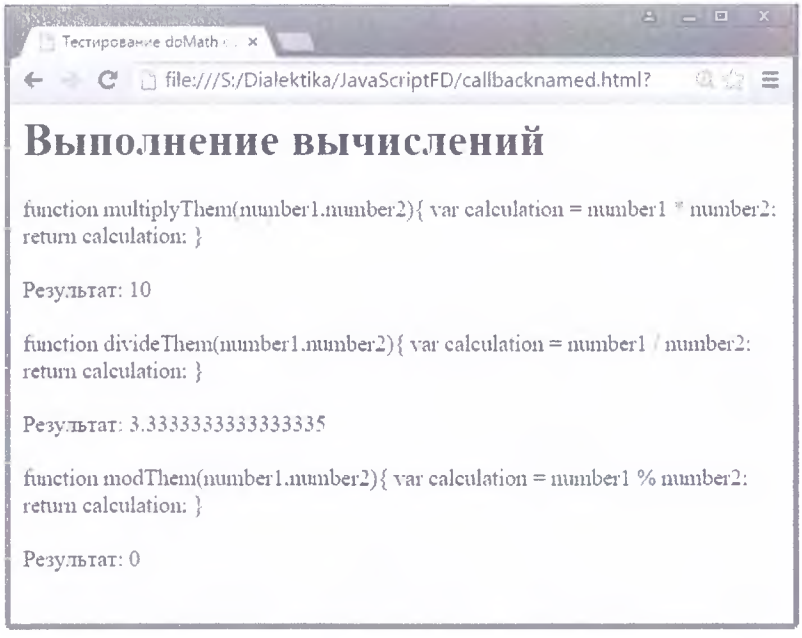


Рис. 15.2. Выполнение вычислений с использованием именованных функций обратного вызова

Использование именованных функций в качестве функций обратного вызова имеет два преимущества по сравнению с использованием анонимных функций:

- ✓ повышает удобочитаемость кода;
- ✓ именованные функции универсальны и могут использоваться не только в качестве функций обратного вызова, но и как независимые функции.

Замыкания

Замыкание — это локальная переменная функции, которая сохраняет свое значение даже после того, как из функции был выполнен возврат.

Рассмотрим пример, приведенный в листинге 15.3. Здесь в одной функции определяется другая функции. Когда внешняя функция возвращает ссылку на внутреннюю

функцию, возвращенная ссылка по-прежнему обеспечивает доступ к локальным данным из внешней функции.

В листинге 15.3 функция `greetVisitor()` возвращает создаваемую в ней функцию `sayWelcome`. Обратите внимание на отсутствие пары круглых скобок после имени функции `sayWelcome` в инструкции `return`. Именно благодаря этому возвращается не результат выполнения функции `sayWelcome()`, а ее код.

Листинг 15.3. Создание функции, использующей другую функцию

```
function greetVisitor(phrase) {
  var welcome = phrase + ". Рад видеть тебя!"; // Локальная
                                               // переменная
  var sayWelcome = function() {
    alert(welcome);
  }
  return sayWelcome;
}

var personalGreeting = greetVisitor('Привет, дружище!');
personalGreeting(); // выводит "Привет, дружище!
                    // Рад видеть тебя!"
```

В листинге 15.3 примечательно то, что функция `greetVisitor()` используется для создания новой пользовательской функции `personalGreeting()`, которая сохраняет возможность доступа к переменным, определенным в исходной функции.

Обычно, после того как функция завершает свою работу, ее локальные переменные становятся недоступными. В то же время благодаря возврату ссылки на функцию (`sayWelcome`) внутренние данные функции `greetVisitor()` становятся доступными для внешнего мира.

Для понимания замыканий важно знать, что такое область видимости переменной в JavaScript и чем вызов функции отличается от ссылки на нее. Присваивая возвращаемое значение функции `greetVisitor()` новой функции `personGreeting()`, программа фактически сохраняет код функции `sayWelcome()`. Это можно проверить, используя метод `toString()`. Если вы добавите в листинг 15.3 инструкцию для вывода значения переменной `personalGreeting` с помощью метода `toString()`, то получите результат, приведенный на рис. 15.3.

```
alert("personalGreeting.toString() \n\n" +
      personalGreeting.toString());
```

На рис. 15.3 переменная `welcome` представляет собой копию значения переменной `welcome` из исходной функции `greetVisitor()`, которое она получила при создании замыкания.

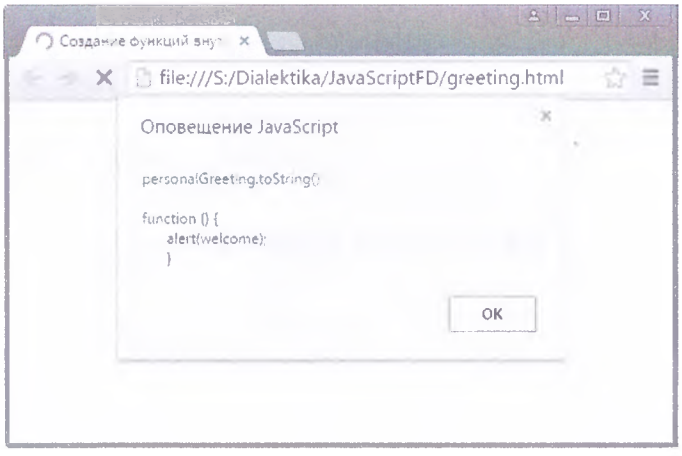


Рис. 15.3. Замыкание включает код возвращенной внутренней функции

В листинге 15.4 создается новое замыкание с использованием другого аргумента функции `greetVisitor()`. Несмотря на то что вызов функции `greetVisitor()` изменяет значение переменной `welcome`, результат вызова первой функции, `personalGreeting()`, остается прежним.

Листинг 15.4. Замыкание содержит секретные ссылки на переменные внешних функций

```
<html>
<head>
  <title>Использование замыканий</title>
  <script>
    function greetVisitor(phrase) {

      var welcome = phrase +
          ". Рад тебя видеть!<br><br>"; // Локальная переменная
      var sayWelcome = function() {
      document.getElementById("greeting").innerHTML += welcome;
}
      return sayWelcome;
      }
      // дождаться загрузки документа
      document.addEventListener("DOMContentLoaded",
          function() {

      // создать функцию
      var personalGreeting = greetVisitor("Hola Amiga");
      // создать еще одну функцию
      var anotherGreeting = greetVisitor("Howdy, Friend");

      // просмотреть код первой функции
      document.getElementById("greeting").innerHTML +=
          "personalGreeting.toString() <br>" +
           personalGreeting. toString() + "<br>";
```

```
      // выполнить код первой функции
        personalGreeting(); // выводит "Hola Amiga.
                            //          Рад тебя видеть!"

      // просмотреть код второй функции
      document.getElementById("greeting").innerHTML +=
          "anotherGreeting.toString() <br>" +
           anotherGreeting.toString() + "<br>";

      // выполнить вторую функцию
      anotherGreeting(); // выводит "Howdy, Friend.
                         //          Рад тебя видеть!"

      // проверить первую функцию
      personalGreeting(); // выводит "Hola Amiga.
                          //          Рад тебя видеть!""

      // завершить метод addEventListener()
    }, false);
  </script>
</head>
<body>
  <p id="greeting"</p>
</body>
</html>
```

Результат выполнения листинга 15.4 в браузере представлен на рис. 15.4.

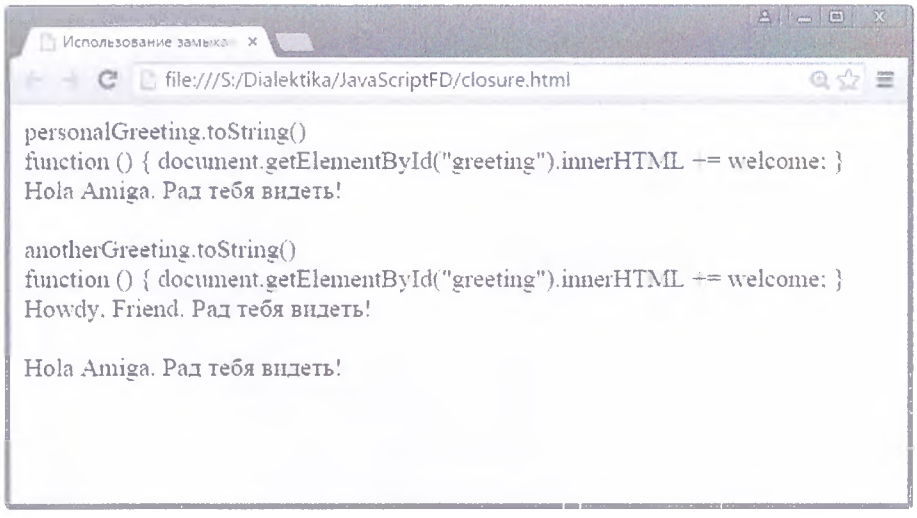


Рис. 15.4. Создание персонализированного приветствия с помощью замыканий

Вам будет несложно разобраться с замыканиями после того, как вы освоите лежащие в их основе понятия, при условии, что замыкания вам действительно нужны. Не огорчайтесь, если пока что вы чувствуете себя неуверенно, работая с ними. Вполне возможно писать программы на JavaScript,

обходясь без замыканий, но как только вы поймете их суть, они станут для вас полезным инструментом и признаком высокого уровня вашей квалификации как программиста.

Использование замыканий

Суть замыканий состоит в том, что они обеспечивают сохранение тех значений локальных переменных функции, которые они имели во время создания замыкания.

В веб-программировании замыкания часто используют для того, чтобы избежать дублирования кода, а также для сохранения тех значений, которые еще будут повторно использоваться в программе, вместо того чтобы вычислять их каждый раз заново.

Еще одна область применения замыканий — это создание специализированных версий функций для специфического использования.

В листинге 15.5 замыкания используются для создания функций с сообщениями об ошибках, специфичными для конкретных ситуаций, которые могут возникать в программе. Все сообщения об ошибках создаются с использованием одной и той же функции.

Функция, назначением которой является создание других функций, называется *фабрикой функций*.

Листинг 15.5. Использование функции для создания других функций

```
<html>
<head>
  <title>Фабрика функций</title>
  <script>

  function createMessageAlert(theMessage){
    return function() {
      alert (theMessage);
    }
  }

  var badEmailError =
      createMessageAlert("Неизвестный адрес электронной почты!");
  var wrongPasswordError =
      createMessageAlert("Неправильный пароль!");

  window.addEventListener('load', loader, false);
  function loader(){
    document.login.yourEmail.addEventListener("change", badEmailError);
    document.login.yourPassword.addEventListener("change",
wrongPasswordError);
  }

</script>
</head>
```

```
<body>
  <form name="login" id="loginform">
    <p>
      <label>Введите адрес электронной почты:
        <input type="text" name="yourEmail">
      </label>
    </p>
    <p>
      <label>Введите пароль:
        <input type="text" name="yourPassword">
      </label>
    </p>
    <button>Отправить</button>
</body>
</html>
```

В листинге 15.5 ключевую роль играет фабрика функций.

```
unction createMessageAlert(theMessage){
  return function() {
    alert (theMessage);
  }
}
```

Чтобы использовать фабрику функций, следует присвоить возвращаемое ею значение переменной, как в приведенной ниже инструкции:

```
var badEmailError = createMessageAlert("Неизвестный адрес!");
```

Предыдущая инструкция создает замыкание, которое можно использовать в любом месте программы, вызвав переменную `badEmailError` как функцию:

```
document.login.yourEmail.addEventListener('change', badEmailError);
```

Глава 16

Приветствуем AJAX и JSON

В этой главе...

- Чтение и запись данных в формате JSON
- Что такое AJAX
- Использование AJAX

"Веб-пространство объединяет не только компьютеры, оно объединяет людей".

Тим Бернерс-Ли

AJAX — это технология повышения динамичности веб-страниц за счет возможности отправки и получения данных в фоновом режиме, в то время как пользователь взаимодействует со страницами. Формат JSON стал стандартным способом представления данных, используемых AJAX-приложениями. В этой главе вы узнаете о том, как с помощью AJAX-технологии сделать так, чтобы ваш сайт заиграл всеми красками!

Закулисная работа AJAX

AJAX (асинхронный JavaScript + XML) — термин, описывающий технологию совместного использования JavaScript, DOM, HTML и объекта `XMLHttpRequest` для обновления частей веб-страницы актуальными данными без обновления всей страницы. В крупномасштабном варианте технология AJAX была впервые реализована на сайте Google Gmail в 2004 году, а свое нынешнее название получила от Джесси Джеймса Гарретта в 2005 году.

HTML DOM динамически изменяет страницы. Важнейшей инновацией, которую привнесла технология AJAX, cnfkj использование объекта `XMLHttpRequest` для асинхронного получения данных от сервера (в фоновом режиме) без блокирования выполнения остального кода JavaScript на веб-странице.

Несмотря на то что на первом этапе технология AJAX базировалась на данных в формате XML (что и объясняет появление буквы "X" в названии), в наши дни в AJAX-приложениях гораздо чаще используется формат данных JSON (JavaScript Object Notation). Большинство людей по-прежнему называют приложения, получающие JSON-данные от сервера в асинхронном режиме, AJAX-приложениями, хотя правильнее было бы использовать другую аббревиатуру — AJAJ.

Примеры применения AJAX

Как только веб-разработчики начали впервые использовать AJAX, эта технология моментально стала визитной карточкой нового направления, получившего название Web 2.0. До появления AJAX самым распространенным способом динамического отображения данных на веб-страницах была загрузка с сервера новой веб-страницы. Взгляните, например, на сайт `craigslist.org`, представленный на рис. 16.1.

Для навигации по категориям списков или результатам поиска на Craigslist пользователь должен щелкнуть на соответствующей ссылке, что приводит к полному обновлению всей страницы, необходимому для отображения запрошенного содержимого.

Несмотря на то что этот способ все еще широко используется, обновление всей страницы лишь для того, чтобы отобразить новые данные в некоторой ее части, неоправданно замедляет работу сайта и приводит к ухудшению удобства взаимодействия с ним.

Сравните навигацию в стиле Craigslist с более характерными для современных веб-приложений средствами навигации сайта Google Plus (рис. 16.2), на котором загрузка нового содержимого в отдельные части страницы осуществляется с помощью AJAX, в то время как панель навигации остается статичной.

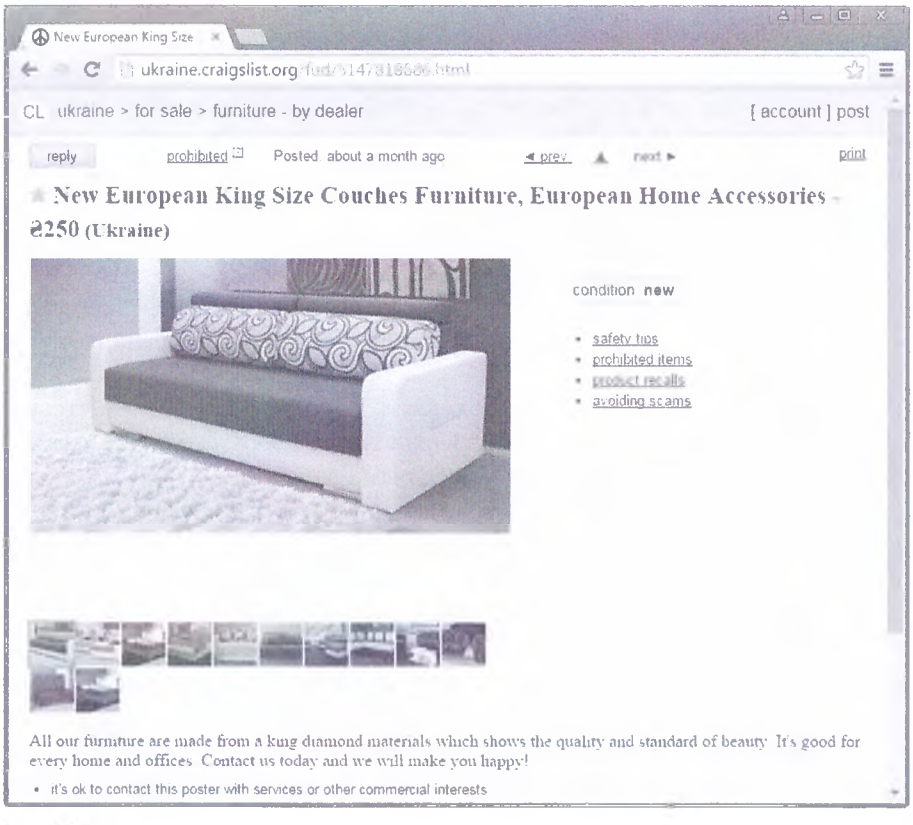


Рис. 16.1. Большое спасибо сайту `craigslist.org`, который чувствует себя вполне счастливым, работая в рамках Web 1.0

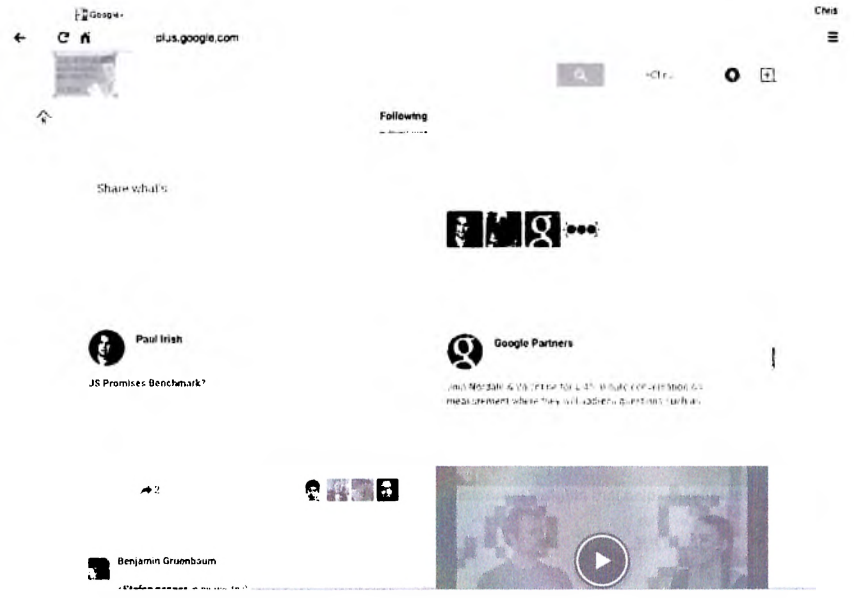


Рис. 16.2. Для обеспечения удобства взаимодействия сайт Google Plus использует технологию AJAX

AJAX не только обеспечивает плавность межстраничных переходов, но и может весьма эффективно использоваться для создания элементов, отображающих оперативные данные. До появления AJAX, если вы хотели отображать в режиме реального времени данные, диаграммы или обновляемое представление папки входящих сообщений электронной почты, вам либо приходилось использовать плагины (например, Adobe Flash), либо задавать автоматическое обновление веб-страницы.

В случае AJAX можно периодически получать новые данные посредством фонового асинхронного процесса, а затем обновлять лишь те элементы страницы, которые нуждаются в обновлении.

В главе 10 рассказывалось о том, как обновлять HTML- и CSS-код веб-страницы, используя методы и свойства HTML DOM. В технологии AJAX та же техника применяется для отображения веб-страниц с обновленными данными.

Сайт Weather Underground's Wundermap (рис. 16.3) отображает карту погоды, постоянно изменяя и обновляя данные. Необходимые для этого данные пересылаются с удаленных серверов с использованием AJAX.

Детальное ознакомление с работой AJAX

Чтобы ближе познакомиться с тем, как работает AJAX, воспользуемся вкладкой Network (Сеть) окна Developer Tools (Инструменты разработчика) браузера Google Chrome. На этой вкладке отображается вся сетевая активность, связанная с текущей веб-страницей. В процессе загрузки страницы здесь фиксируются запросы к серверу наряду с загрузкой кода HTML, CSS, JavaScript и изображений. После того как страница загрузится, на вкладке отображаются также асинхронные HTTP-запросы и ответы, на которых зиждется технология AJAX.

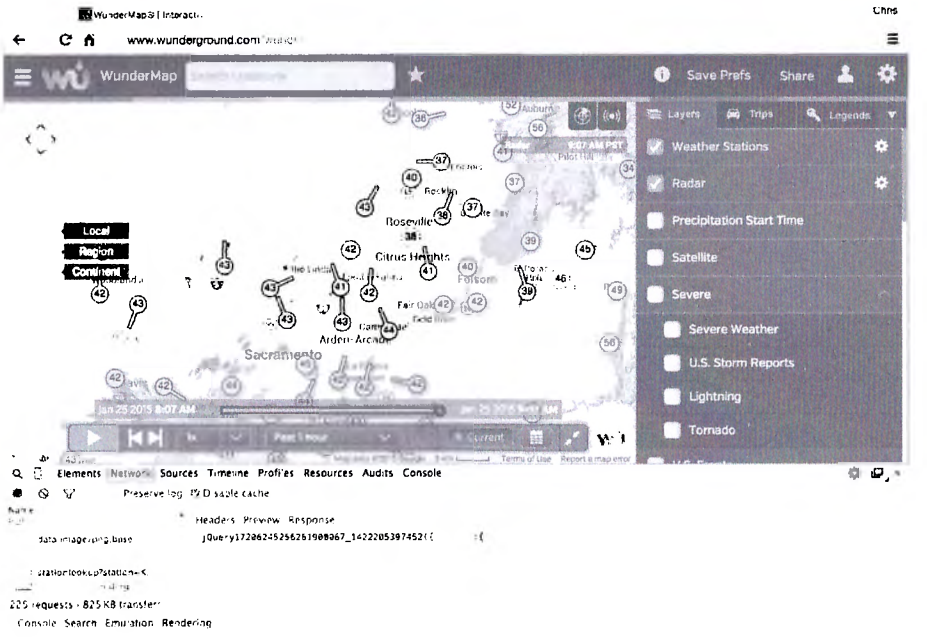


Рис. 16.3. На сайте Wundermap технология AJAX используется для отображения оперативной карты погоды

Чтобы просмотреть в Chrome запросы и ответы AJAX, выполните следующие действия.

1. **Откройте браузер Chrome и выполните переход на сайт `www.wunderground.com/wundermap`.**
2. **Откройте панель инструментов разработчика, нажав комбинацию клавиш <Command+Option+I> (Mac) или <Ctrl+Shift+I> (Windows).**
3. **Откройте вкладку Network (Сеть).**

   Окно инструментов разработчика должно выглядеть примерно так, как показано на рис. 16.4. Возможно, вам понадобится немного расширить это окно, перетащив его верхнюю границу. Не беспокойтесь, если размер области содержимого окна браузера уменьшится настолько, что ее использование будет значительно затруднено. На данном этапе нам важно лишь то, что происходит в окне разработчика.

   Заметьте, что на вкладке Network периодически появляются новые элементы. Это запросы и ответы AJAX. Одни из них — это изображения, возвращаемые с сервера, другие — данные, предназначенные для клиентского JavaScript-кода.

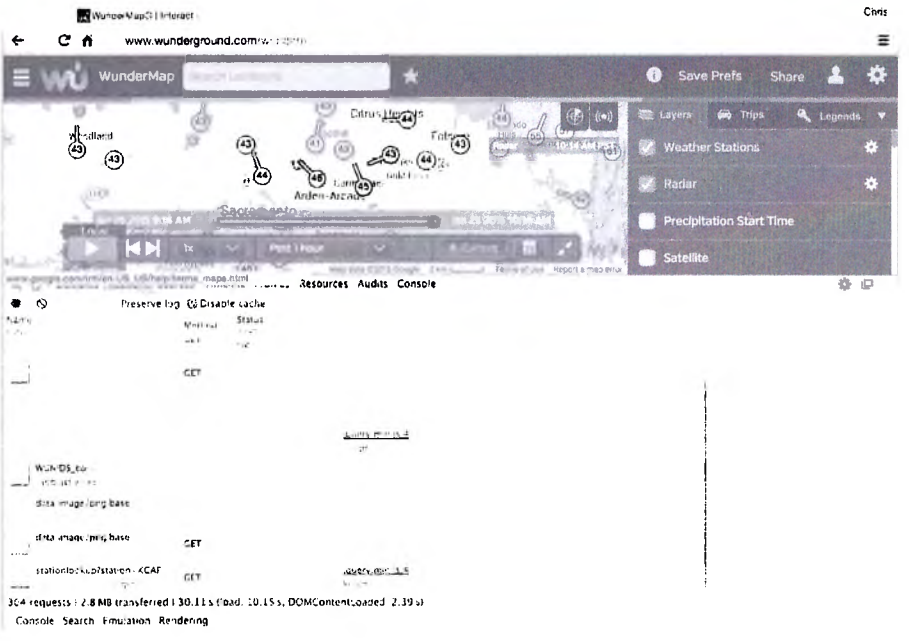


Рис. 16.4. Вкладка `Network` панели инструментов разработчика

4. **Щелкните на одной из строк в столбце Name (Имя) вкладки Network.**

   На вкладке отобразятся данные, касающиеся конкретного элемента (рис. 16.5).

5. **Пощелкайте на ярлыках вкладок (Headers, Preview, Response и др.) в панели детализированного отображения данных и исследуйте эти данные.**

   На первой вкладке, Headers (Заголовки), отображается HTTP-запрос, отправленный на удаленный сервер. Просмотрите содержимое элемента `Request URL`. Это стандартный адрес сайта, вместе с которым данные передаются на удаленный сервер.

6. **Выделите и скопируйте значение `Request URL` одного из просматриваемых элементов.**

7. **Откройте новую вкладку в своем браузере и вставьте скопированное значение в адресную строку.**

   Откроется страница, содержащая данные или изображение (рис. 16.6).

8. **Сравните результаты открытия новой вкладки по адресу, взятому из элемента `Request URL`, с результатами, отображаемыми на вкладке Response (Ответ) панели инструментов разработчика.**

   Они должны быть похожи, хотя могут и не быть идентичными, поскольку были получены в разное время.

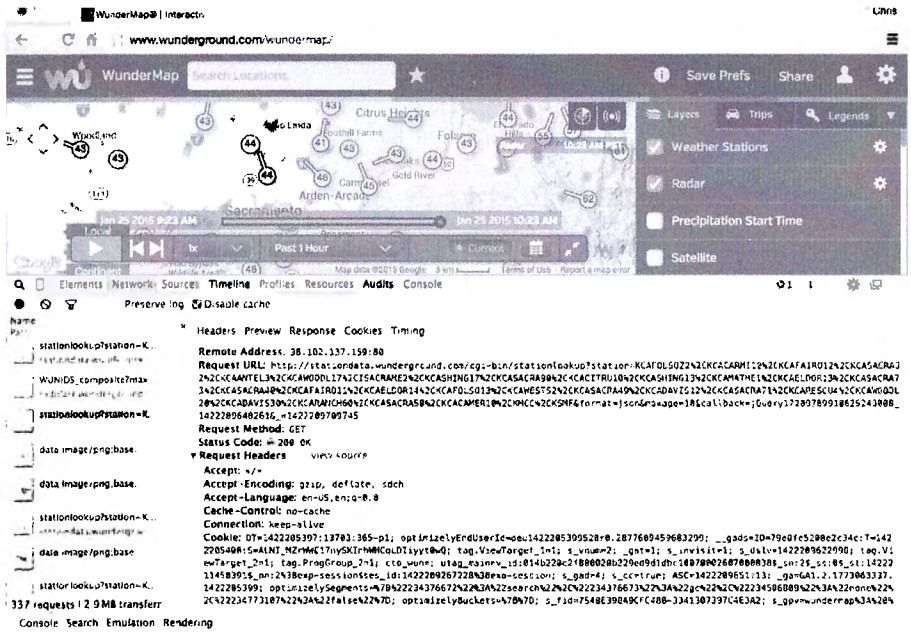


Рис. 16.5. Просмотр дополнительной информации о конкретной записи на вкладке `Network`

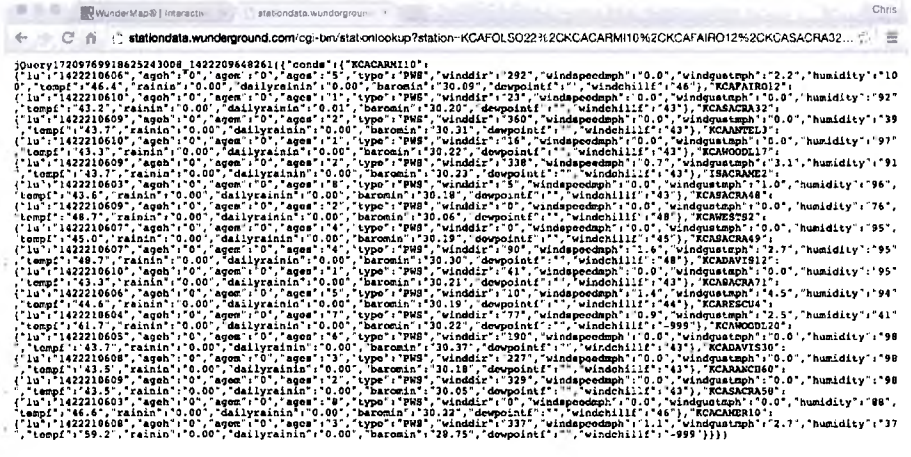


Рис. 16.6. Результат перехода по адресу, скопированному из элемента `Request URL` *HTTP-заголовка на вкладке* **Network**

Как видите, в действительности в AJAX нет ничего таинственного. Все сводится к тому, что JavaScript-код веб-страницы просто запрашивает и получает данные с сервера. Все детали закулисной работы доступны для изучения с помощью панели инструментов разработчика Chrome (или же других аналогичных инструментальных средств, которые предлагаются большинством современных браузеров).

Использование объекта `XMLHttpRequest`

С помощью объекта `XMLHttpRequest` браузеры могут запрашивать данные от сайта с определенным URL без обновления страницы.

Первоначально объект `XMLHttpRequest` был создан и реализован в браузере Internet Explorer компанией Microsoft и с тех пор стал веб-стандартом, которого придерживаются все производители современных браузеров.

Вы можете использовать методы и свойства объекта `XMLHttpRequest` для получения данных от удаленного или локального сервера. Несмотря на свое название, объект `XMLHttpRequest` позволяет получать не только XML-разметку, но и данные других типов, причем с использованием не только протокола HTTP, но и других протоколов.

В листинге 16.1 приведен пример использования объекта `XMLHttpRequest` для загрузки содержимого внешнего текстового документа, включающего HTML-разметку, в текущий HTML-документ.

Листинг 16.1. Использование объекта `XMLHttpRequest` для загрузки внешних данных

```
<html>
<head>
  <title>Загрузка внешних данных</title>
  <script>
    window. AddEventListener("load",init,false);
    function init(e){
      document.getElementById("myButton").
          addEventListener("click",documentLoader,false);
    }

    function reqListener () {
      console.log(this.responseText);
      document.getElementById("content").innerHTML = this.
          responseText;
    }

    function documentLoader(){
      var oReq = new XMLHttpRequest();
      oReq.onload = reqListener;
      oReq.open("get", "loadme.txt", true);
      oReq.send();
    }
  </script>
</head>
<body>
  <form id="myForm">
```

```
    <button id="myButton" type="button">Загрузить</button>
  </form>
  <div id="content"></div>
</body>
</html>
```

Ключевой составляющей этого документа является функция `documentLoader()`.

```
function documentLoader(){
  var oReq = new XMLHttpRequest();
  oReq.onload = reqListener;
  oReq.open("get", "loadme.txt", true);
  oReq.send();
}
```

В первой строке кода создается новый объект `XMLHttpRequest`, которому присваивается имя `oReq`:

```
var oReq = new XMLHttpRequest();
```

Объект `oReq` обеспечивает доступ к методам и свойствам объекта `XMLHttpRequest`.

Вторая строка кода назначает функцию-обработчик `reqListener()` событию `event` объекта `oReq`:

```
oReq.onload = reqListener;
```

В третьей строке метод `open()` используется для создания запроса:

```
oReq.open("get", "loadme.txt", true);
```

В данном случае эта функция загружает файл `loadme.txt`, используя HTTP-метод GET. Ее третий параметр — это аргумент `async`. Он позволяет определить, должен ли запрос выполняться как асинхронный. Если он имеет значение `false`, то возврат из метода `send()` произойдет лишь после того, как выполнится запрос. Если же он установлен в `true`, то уведомление о выполнении запроса поступает через слушателя событий. Поскольку для слушателя событий задано событие `load`, то асинхронный запрос — это как раз то, что нужно.

Весьма маловероятно, что у вас возникнут ситуации, когда вы захотите установить для аргумента `async` значение `false`. В действительности, учитывая то, что синхронные запросы негативно влияют на взаимодействие с пользователем, некоторые браузеры начали игнорировать значение `false` и в любом случае обрабатывают запрос как асинхронный.

Последняя строка кода функции `documentLoader()` осуществляет фактическую отправку запроса, созданного с помощью метода `open()`:

```
oReq.send();
```

Метод `open()` получит самую последнюю версию запрошенного файла. В приложениях, создаваемых на основе технологии Live Data и ориентированных на обработку данных в режиме реального времени, для формирования повторных запросов на получение обновленных данных с сервера с помощью AJAX часто используют циклы.

Политика одинакового источника

Сохранив HTML-документ, приведенный в листинге 16.1, на своем компьютере и открыв его в браузере, вы, вероятнее всего, не получите ожидаемого результата. Если вы загрузите документ со своего компьютера, а затем откроете его в консоли JavaScript браузера Chrome, то увидите сообщения об ошибке наподобие тех, которые приведены на рис. 16.7.

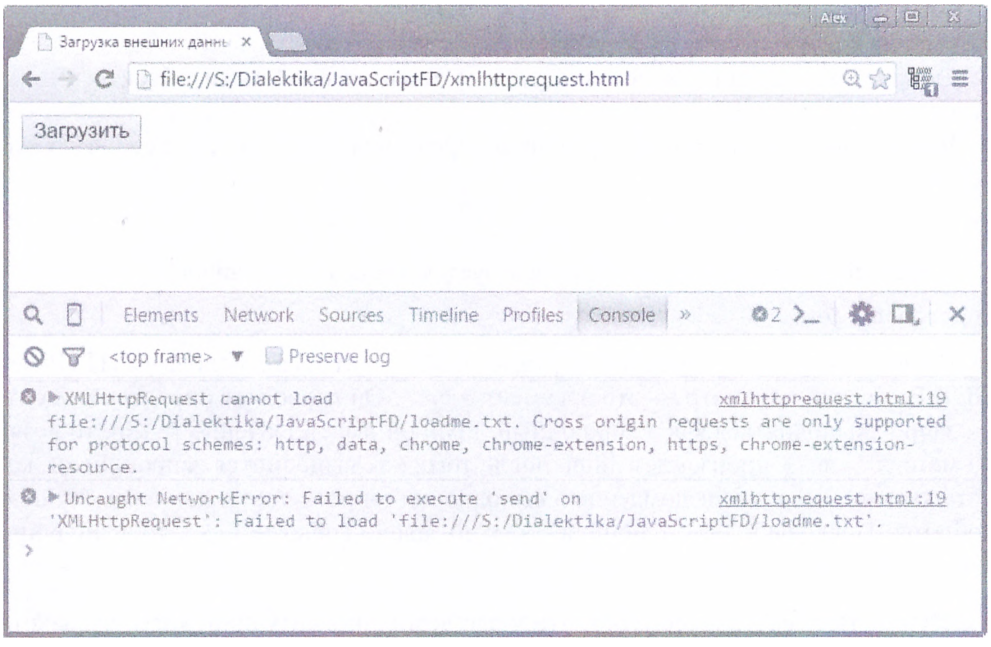


Рис. 16.7. Ошибки, возникающие при попытке использования объекта `XMLHttpRequest`. для запроса локального файла

Суть проблемы состоит в том, что здесь срабатывает так называемая *политика одного источника* (same-origin policy). Во избежание непреднамеренного выполнения пользователем AJAX-запросов, которые могут приводить к загрузке на веб-страницу вредоносного кода, браузеры по умолчанию возвращают сообщение об ошибке всякий раз, когда сценарий пытается загрузить данные с домена, не совпадающего с тем, с которого была загружена запрашивающая страница. Если вы загрузите веб-страницу с сайта `www.example.com` и сценарий на этой странице попытается получить данные с

сервера `www.watzthis.com`, то браузер не даст этому запросу выполниться и выведет сообщения об ошибке, аналогичные тем, которые приведены на рис. 16.7.

Политика одного источника применяется также к файлам, находящимся на локальном компьютере. Если бы этого не делалось, то запросы, выполняемые с помощью объекта `XMLHttpRequest`, могли бы быть использованы в ущерб безопасности компьютера.

У вас нет никаких причин беспокоиться относительно того, что приведенные в книге примеры могут нанести вред вашему компьютеру. Однако для того, чтобы примеры, приведенные в этой главе, работали корректно, необходимо каким-то образом обойти правило одного источника.

Первый способ состоит в том, чтобы расположить HTML-файл, содержащий функцию `documentLoader()`, и текстовый файл на одном сервере. Можете посмотреть, как работает этот пример, открыв в браузере следующую страницу:

```
http://www.codingjsfordummies.com/code/ch16/listing16-1.html
```

Второй способ обхода ограничений состоит во временном отключении политики одного источника при запуске браузера.

Приведенные ниже инструкции позволяют тестировать файлы лишь на локальном компьютере. Не пытайтесь посещать другие сайты в то время, когда политика одного источника в браузере отключена. Тем самым вы оставляете свой компьютер беззащитным перед вредоносным кодом.

Чтобы отключить политику одного источника на компьютере, работающем под управлением Mac OS, выполните следующие действия.

1. **Закройте браузер Chrome.**
2. **Откройте приложение Terminal и запустите Chrome с помощью следующей команды:**

```
/Applications/Google\ Chrome.app/Contents/MacOS/
          Google\ Chrome --disable-web-security
```

Чтобы отключить политику одного источника на компьютере, работающем под управлением Windows, выполните следующие действия.

1. **Закройте браузер Chrome.**
2. **Откройте окно командной строки и перейдите к папке, в которой установлено приложение Chrome.**
3. **Запустите браузер, введя следующую команду:**

```
Chrome.exe --disable-web-security
```

После этого вы сможете выполнять файлы, содержащие локальные AJAX-запросы, вплоть до закрытия браузера. Когда браузер будет закрыт, а затем повторно открыт, ограничения политики одного источника автоматически восстановятся.

Результат выполнения листинга 16.1 в браузере, на этот раз не сопровождающийся выводом сообщений об ограничениях, налагаемых политикой одного источника, приведен на рис. 16.8.

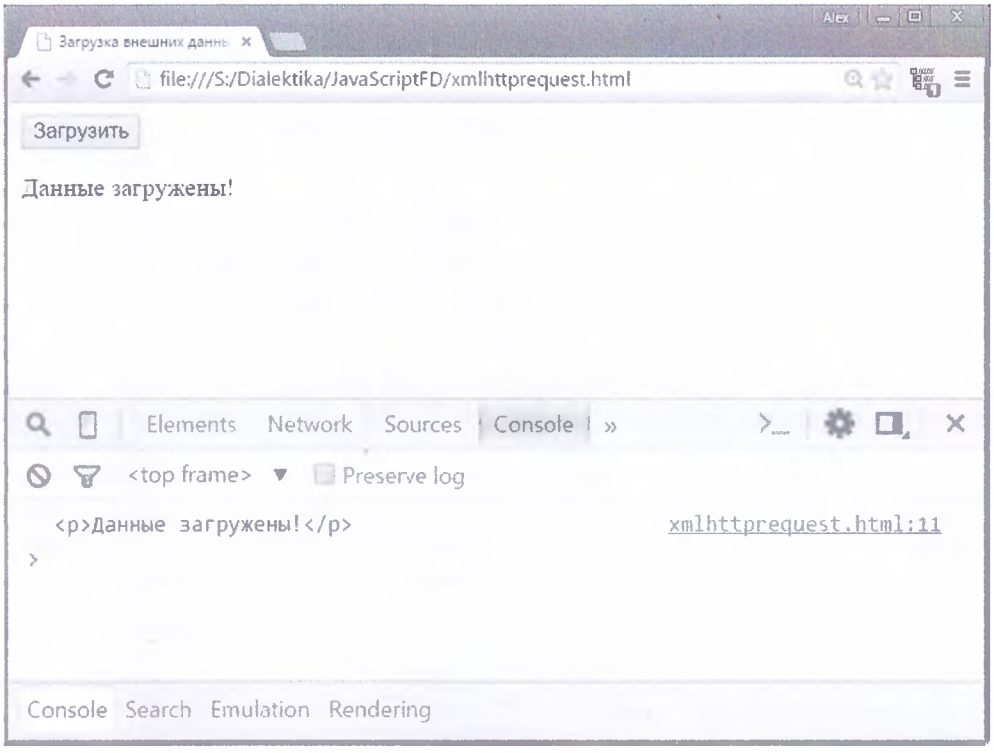


Рис. 16.8. Результат выполнения листинга 16.1 в браузере с отключенной политикой одного источника

CORS — серебряная пуля AJAX-запросов

В веб-приложениях довольно часто встречаются ситуации, когда для получения данных требуется выполнять запросы к различным серверам. Например, карты Google бесплатно доступны для использования в приложениях независимых разработчиков.

Для повышения надежности транзакций, осуществляемых между серверами, были созданы механизмы, позволяющие браузерам и серверам улаживать расхождения и устанавливать взаимное доверие.

В настоящее время наилучшим методом урегулирования прав доступа к ресурсам между серверами является спецификация *CORS* (Cross-Origin Resource Sharing — совместное использование ресурсов между разными источниками).

Чтобы увидеть, как работает технология CORS, посетите сайт Weather Underground's Wundermap и просмотрите данные на вкладке **Network** панели инструментов разработчика Chrome. Щелкните на одном из запросов, начинающихся следующим URL:

```
http://stationdata.wunderground.com/cgi-bin/stationlookup
```

Щелкните на вкладке Headers (Заголовки), и вы увидите следующий текст в HTTP-заголовке:

```
Access-Control-Allow-Origin: *
```

Это заголовок ответа CORS, для отправки которого сконфигурирован данный сервер. Символ "звездочка" (*) после двоеточия указывает на то, что данный сервер будет принимать запросы от любого источника. Если бы владельцы сайта wunderground.com хотели ограничить доступ к данным в этом сценарии только конкретными серверами или авторизованными пользователями, то они могли бы это сделать с помощью CORS.

Передача объектов с помощью JSON

В листинге 16.1 технология AJAX использовалась для открытия и отображения текстового документа, содержащего фрагмент HTML. Другая распространенная область применения AJAX — это запрос и получение данных для их последующей обработки браузером.

Например, на сайте gasbuddy.com карты Google используются совместно с данными о ценах на бензин для предоставления понятной и постоянно обновляемой картины распределения цен по различным регионам (рис. 16.9).

Если вы просмотрите данные, касающиеся сайта gasbuddy.com, на вкладке Network, то увидите, что ответы на некоторые запросы выглядят как код примерно того вида, который представлен в листинге 16.2.

Листинг 16.2. Часть ответа на запрос, отправленный на сайт gasbuddy.com

```
([{id:"tuwtvtuvvvv",base:[351289344,822599680],zrange:[
        11,11],layer:"m@288429816",features:[{id:"172
        43857463485476481",a:[0,0],bb:[-8,-8,7,7,-47-
        ,7,48,22,-41,19,41,34],c:"{1:{title:\"Folsom
        Lake State Recreation Area\"},4:{type:1}}"}]},
        {id:"tuwtvtuvvvw",zrange:[11,11],layer:"m@2884
        29816"},{id:"tuwtvtuvvwv",base:[351506432,8242
        91328],zrange:[11,11],layer:"m@288429816",feat
        ures:[{id:"8748558518353272790",a:[0,0],bb:[-
        8,-8,7,7,-41,7,41,22],c:"{1:{title:\"Deer Creek
        Hills\"},4:{type:1}}"}]},{id:"tuwtvtuvvww",zran
        ge:[11,11],layer:"m@288429816"}])
```

Если взять из этого блока небольшую часть данных и немного переформатировать ее, то вы получите нечто уже знакомое вам (листинг 16.3).

Листинг 16.3. Переформатированные данные ответа, полученного с сайта gasbuddy.com

```
{id:"tuwtvtuvvvv",
base:[351289344,822599680],
zrange:[11,11],
layer:"m@288429816",
features:[{
```

```
id:"17243857463485476481",
a:[0,0],
bb:[-8,-8,7,7,-47,7,48,22,-41,19,41,34],
c:"{
1:{title:\"Folsom Lake State Recreation Area\"},
4:{type:1}
}"}
]}
}
```

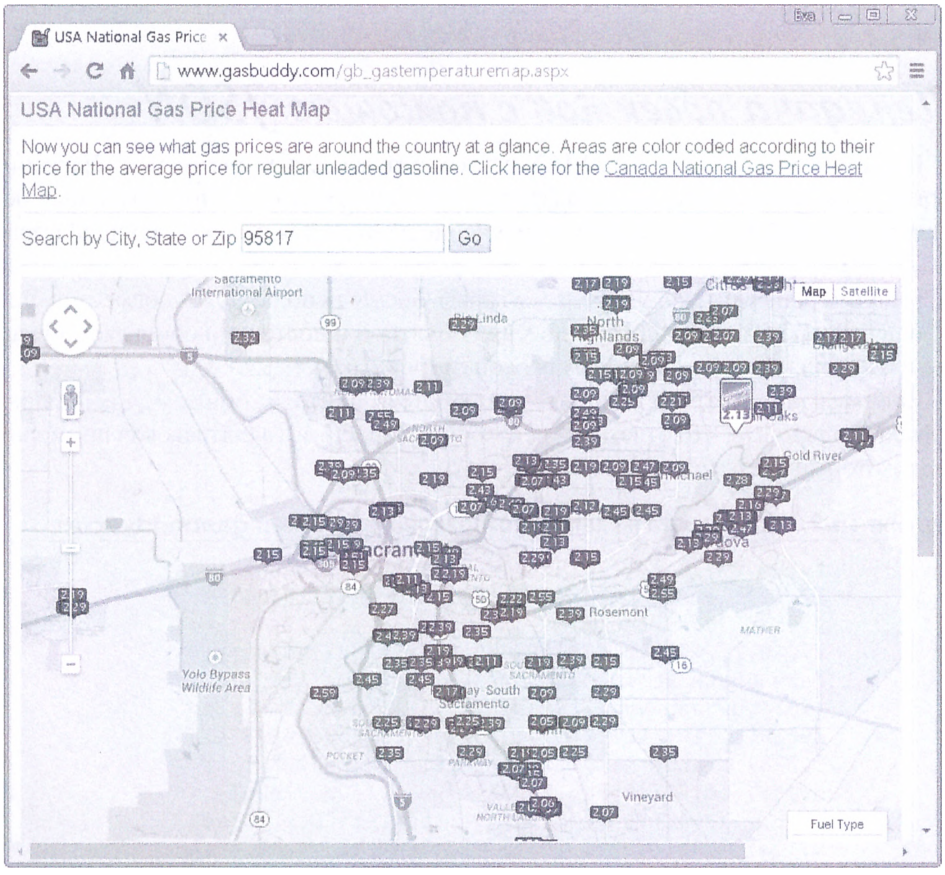


Рис. 16.9. На сайте `gasbuddy.com` технология AJAX используется для отображения цен на бензин в различных регионах

Внимательно изучив формат данных, вы увидите, что он подозрительно напоминает формат *имя:значение*, используемый для описания литеральных объектов JavaScript (см. главу 8).

Основной причиной необычайной простоты использования формата JSON является тот факт, что он уже является рабочим форматом JavaScript, вследствие чего отпадает необходимость в преобразовании данных. Например, в листинге 16.4 представлен файл, содержащий информацию об этой книге.

Листинг 16.4. Данные в формате JSON, описывающие книгу *JavaScript для чайников*

```
{
  "book_title": "JavaScript для чайников",
  "book_author": "Крис Минник и Ева Холланд",
  "summary": "Все, что надо знать начинающим для того, чтобы
              приступить к написанию программ на JavaScript!"
}
```

В листинге 16.5 показано, как загрузить эти данные на веб-страницу с помощью JavaScript для последующего их отображения в формате HTML.

Листинг 16.5. Отображение JSON-данных с помощью JavaScript

```
<html>
<head>
  <title>Отображение JSON-данных</title>
  <script>
    window.addEventListener('load',init,false);
    function init(e){
      document.getElementById('myButton').
              addEventListener('click',documentLoader,false);
    }

    function reqListener () {
      // преобразовать строку из файла в объект с помощью
      // метода JSON.parse()
      var obj = JSON.parse(this.responseText);

      // отобразить данные объекта обычным способом
      document.getElementById('book_title').innerHTML =
              obj.book_title;
      document.getElementById('book_author').innerHTML =
              obj.book_author;
      document.getElementById('summary').innerHTML =
              obj.summary;
    }

    function documentLoader(){
      var oReq = new XMLHttpRequest();
      oReq.onload = reqListener;
      oReq.open("get", "Листинг16-4.json", true);
      oReq.send();
    }
  </script>
</head>
<body>
  <form id="myForm">
    <button id="myButton" type="button">Щелкните для
            загрузки</button>
  </form>
  <h1>Название книги</h1>
  <div id="book_title"></div>
```

```
  <h2>Авторы</h2>
  <div id="book_author"></div>
  <h2>Резюме</h2>
  <div id="summary"></div>
</body>
</html>
```

Ключевым моментом при отображении любых JSON-данных, поступающих в JavaScript из внешнего источника, является их преобразование из строки в объект с помощью метода `JSON.parse()`. Сделав это, вы сможете получить доступ к значениям, хранящимся в файле JSON, используя точечную или скобочную нотацию, как вы это делаете, обращаясь к свойствам любого JavaScript-объекта.

На рис. 16.10 представлены результаты выполнения листинга 16.5 в браузере, получаемые после щелчка на кнопке для загрузки JSON-данных.

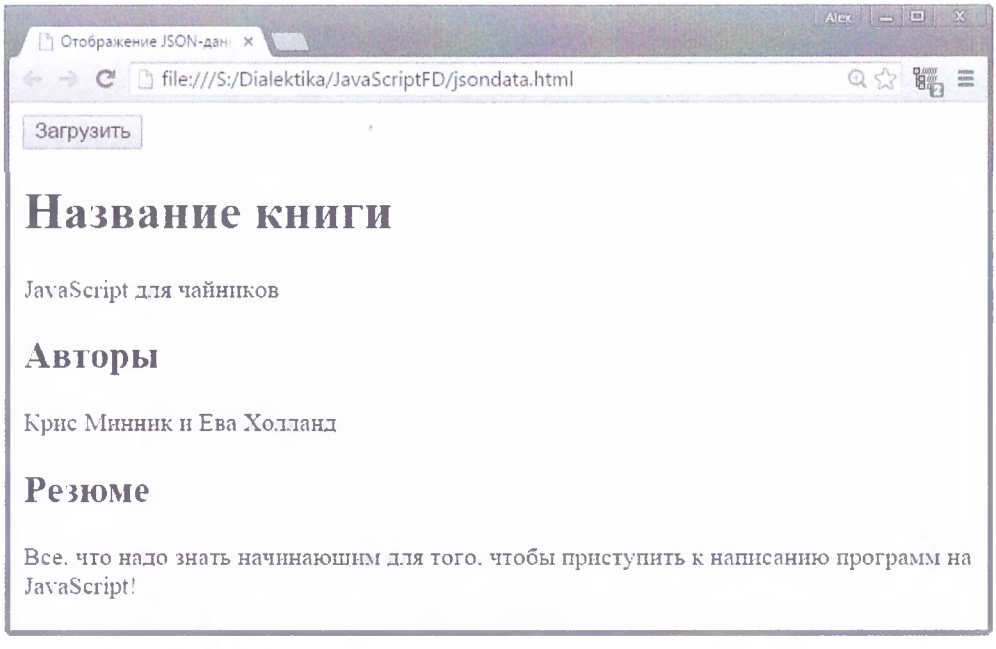


Рис. 16.10. Отображение JSON-данных на HTML-странице

Часть V

JavaScript и HTML5

В этой части...

- ✓ Программные интерфейсы HTML5
- ✓ Основы jQuery

Глава 17

Программные интерфейсы HTML5

В этой главе...

- Знакомство с программными интерфейсами HTML5
- Использование возможностей геолокации
- Доступ к мультимедийным данным

"Язык — это инопланетный вирус".
Уильям С. Берроуз

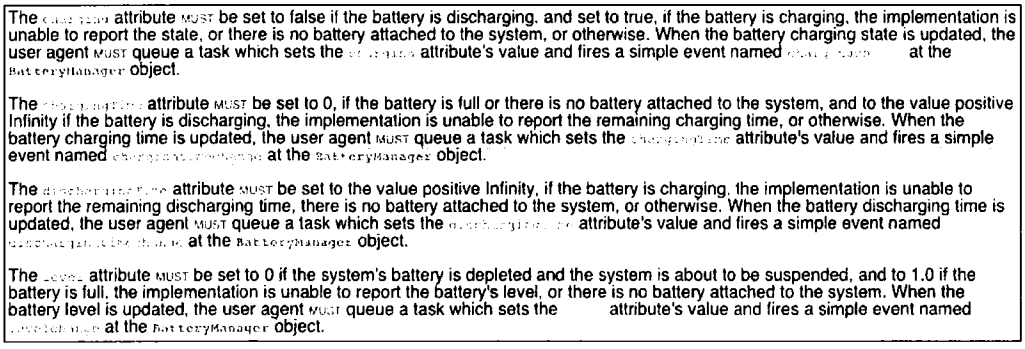
Интерфейсы прикладного программирования (Application Programming Interface, API) в HTML5 обеспечивают доступ к расширенной функциональности компьютеров и мобильных устройств. Эта глава познакомит вас как со всевозможными API, так и с используемыми в них стандартными методами и подходами. Работа с некоторыми из наиболее интересных API будет проиллюстрирована соответствующими примерами.

Как работают API

Интерфейсы прикладного программирования — это наборы программ и стандартов, предоставляющих программистам доступ к возможностям приложений. Это средства, с помощью которых одни компьютерные программы могут взаимодействовать с другими. В тех случаях, когда браузер разрешает программе на JavaScript взаимодействовать с ним, это делается посредством API.

К примеру, стандарт Battery Status API консорциума W3C описывает структуру данных отчета о текущем состоянии батареи устройства (скажем, смартфона или планшета). Другие программы (например, JavaScript-программы, включенные в веб-страницы) могут получать доступ к методам Battery Status API, чтобы определить уровень заряда батареи и выяснить, сколько времени осталось до ее полного разряда. Затем эта информация может быть использована в вашей программе.

Язык описания API отличается точностью и детально определяет, каким образом тот или иной API должен реализовываться браузерами. На рис. 17.1 приведена выдержка из самой последней версии спецификации Battery Status API.



The charging attribute MUST be set to false if the battery is discharging, and set to true, if the battery is charging, the implementation is unable to report the state, or there is no battery attached to the system, or otherwise. When the battery charging state is updated, the user agent MUST queue a task which sets the charging attribute's value and fires a simple event named chargingchange at the BatteryManager object.

The chargingTime attribute MUST be set to 0, if the battery is full or there is no battery attached to the system, and to the value positive Infinity if the battery is discharging, the implementation is unable to report the remaining charging time, or otherwise. When the battery charging time is updated, the user agent MUST queue a task which sets the chargingTime attribute's value and fires a simple event named chargingtimechange at the BatteryManager object.

The dischargingTime attribute MUST be set to the value positive Infinity, if the battery is charging, the implementation is unable to report the remaining discharging time, there is no battery attached to the system, or otherwise. When the battery discharging time is updated, the user agent MUST queue a task which sets the dischargingTime attribute's value and fires a simple event named dischargingtimechange at the BatteryManager object.

The level attribute MUST be set to 0 if the system's battery is depleted and the system is about to be suspended, and to 1.0 if the battery is full, the implementation is unable to report the battery's level, or there is no battery attached to the system. When the battery level is updated, the user agent MUST queue a task which sets the attribute's value and fires a simple event named levelchange at the BatteryManager object.

Рис. 17.1. Выдержка из спецификации Battery Status API

Обычно описания API оформляются в виде спецификаций, содержащих подробные определения свойств и методов, доступных программистам, аргументов, передаваемых методам, и типов значений, возвращаемых свойствами.

Во многих случаях, когда API разрабатываются специальными комитетами стандартизации наподобие World Wild Web Consortium (консорциум W3C), их описания ориентируются на изложение принципов взаимодействия программ с браузерами, а не рекомендаций, касающихся фактической реализации этих взаимодействий. Решения о способах реализации стандарта API разработчики браузеров принимают самостоятельно.

В документации любого API содержится информация относительно того, как программисты могут взаимодействовать с программами и как программы должны отвечать на их запросы. При чтении документации API, описывающей взаимодействие с браузерами, очень важно иметь в виду, что из самого факта существования API вовсе не следует, что программисты действительно могут воспользоваться соответствующими средствами. Было предложено и написано множество API, которые до сих пор остались нереализованными в браузерах или реализованы лишь частично.

Проверка браузерной поддержки программных интерфейсов HTML5

Наилучший источник информации о поддержке браузерами какого-либо конкретного стандарта — сайт `www.caniuse.com`. На этом сайте публикуются сведения относительно всех элементов и API HTML5 вместе с таблицами, отражающими степень поддержки каждого из них различными браузерами. На рис. 17.2 представлена информация о состоянии браузерной поддержки стандарта IndexedDB на момент написания данной книги.

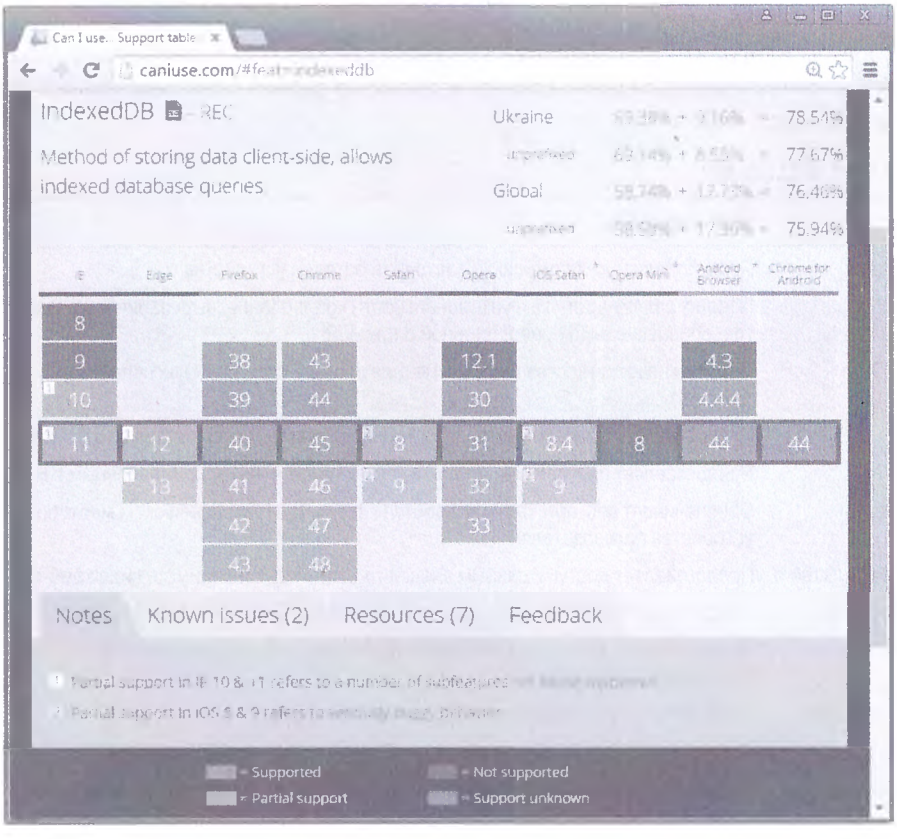


Рис. 17.2. Приведенная на сайте `caniuse.com` информация о браузерной поддержке стандарта IndexedDB

Знакомство с программными интерфейсами HTML5

Стандартом HTML5 определен и документирован ряд API, которые JavaScript-программисты могут использовать для доступа к возможностям браузеров стандартизированным способом.

Многие из API, описанных в HTML5, все еще не завершены и не реализованы во всех браузерах. Уже реализованные API чрезвычайно полезны и расширяют границы возможного в веб-приложениях.

Список этих API постоянно изменяется и расширяется. В табл. 17.1 приведен перечень наиболее популярных и надежно поддерживаемых API, определенных в HTML5 на момент написания книги.

Исчерпывающий список HTML5 API можно найти в шпаргалке в конце книге.

В каждом из API, описанных в спецификации HTML5, в точности определено, каким образом программисты должны взаимодействовать с функциональностью веб-браузеров и компьютеров. Однако путь от идеи до ее реализации может быть болезненно долгим и сложным.

Таблица 17.1. HTML5 API

API	Использование
Battery Status	Предоставляет информацию о состоянии батареи устройства
Clipboard	Предоставляет доступ к функциональности копирования, вырезания и вставки объектов, обеспечиваемой операционной системой
Drag and Drop	Поддерживает средства перетаскивания элементов в пределах окна и между окнами браузера
File	Предоставляет программам безопасный доступ к файловой системе устройства
Forms	Предоставляет программам доступ к новым типам данных, определенным в HTML5
Geolocation	Обеспечивает веб-приложениям доступ к данным о географическом местоположении устройства пользователя
getUserMedia/Stream	Предоставляет доступ к данным внешнего устройства (например, видео веб-камеры)
Indexed database	Создает простую клиентскую систему управления базой данных в браузере
Internationalization	Предоставляет доступ к средствам форматирования и сравнения строк с учетом региональных особенностей
Screen Orientation	Определяет ориентацию экрана (книжная/альбомная)
Selection	Поддерживает выбор элементов в JavaScript с помощью CSS-селекторов
Server-sent events	Разрешает серверу загружать данные в браузер без предварительного ожидания запроса
User Timing	Предоставляет программистам доступ к высокоточным временным меткам для измерения производительности приложений
Vibration	Разрешает доступ к возможностям обработки данных о вибрации устройства
Web Audio	Предоставляет средства обработки и синтеза аудиоданных
Web Speech	Предоставляет средства речевого ввода и преобразования текста в речь
Web storage	Позволяет сохранять в браузере пары "ключ–значение"
Web sockets	Открывает сеанс интерактивного обмена данными между браузером и сервером
Web workers	Позволяет JavaScript выполнять сценарии в фоновом режиме
XMLHTTPRequest2	Улучшает возможности `XMLHTTPRequest`, устраняя необходимость в использовании обходных путей для устранения ошибок, связанных с нарушением политики одного источника, и обеспечивая возможность использования `XMLHTTPRequest` с новыми средствами

Некоторые из вышеперечисленных API были подвергнуты множеству тестов и проверок, обязательных для стандартов, претендующих на широкую поддержку. В наибольшей степени это относится к интерфейсу Geolocation API.

Использование HTML5 Geolocation API

Geolocation API предоставляет программам доступ к геолокационной функциональности браузера, которая может сообщать программе информацию о географическом местоположении устройства.

Этот API, относящийся к числу тех, которые отличаются наиболее надежной поддержкой, реализован примерно в 90% настольных и мобильных браузеров, в том числе во всех основных браузерах, за исключением Opera Mini.

Что такое геолокация

В Geolocation API описано, каким образом JavaScript может взаимодействовать с объектом `navigator.geolocation` для получения данных о текущем местоположении устройства, включая следующие данные.

- ✓ **Широта** (Latitude). Широта в десятичных градусах.
- ✓ **Долгота** (Longtitude). Долгота в десятичных градусах.
- ✓ **Высота** (Altitude). Высота над уровнем моря в метрах.
- ✓ **Направление** (Heading). Направление движения устройства.
- ✓ **Скорость** (Speed). Скорость устройства в метрах за секунду.
- ✓ **Точность** (Accuracy). Точность указания широты и долготы в метрах.

Получая эти данные частично или полностью, JavaScript-приложение, выполняющееся в браузере, может указать местоположение пользователя на карте, запрашивать такие источники, как Google Maps, относительно расположения различных достопримечательностей или ресторанов, ближайших к пользователю, и т.п.

Как работает геолокация

Когда JavaScript инициирует запрос местоположения устройства посредством объекта `Geolocation`, получению информации о позиции предшествует выполнение ряда этапов.

Прежде всего браузер должен убедиться в том, что пользователь разрешил данному приложению получать доступ к геолокационной информации. Для запроса разрешения у пользователя разные браузеры используют разные способы, но обычно это делается посредством всплывающего окна или уведомления.

Например, в браузере Chrome при поступлении от сайта запроса геолокационных данных под адресной строкой отображается значок геолокации и выводится соответствующее сообщение (рис. 17.3).

После предоставления сайту разрешения на доступ к геолокационным данным он пытается обнаружить ваше устройство. Для этого используется целый ряд способов, начиная с наиболее точных и заканчивая менее точными.

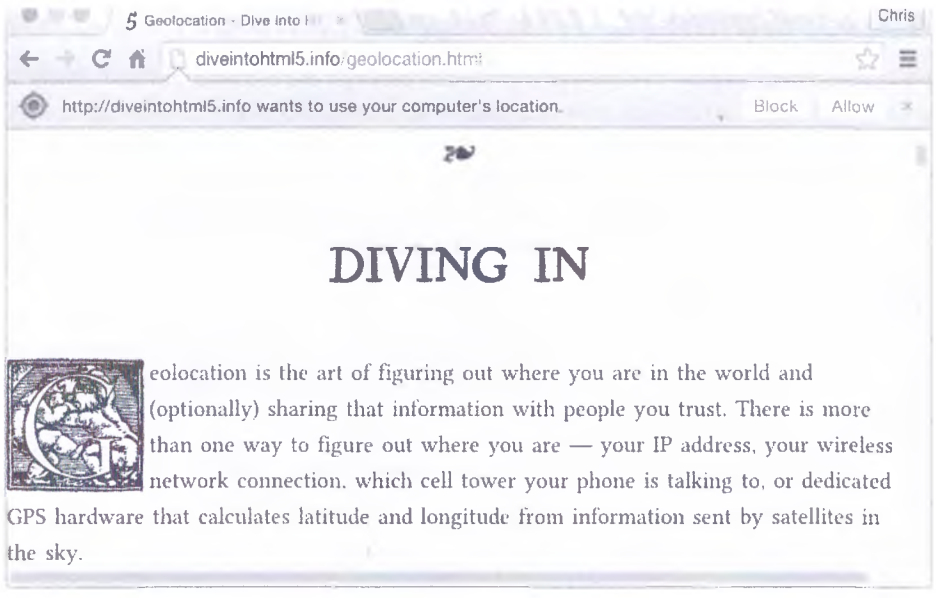


Рис. 17.3. Отображение запроса геолокационных данных под адресной строкой в браузере Chrome

Если программа указывает на необходимость получения высокоточных данных, то будет сделана попытка получения точной GPS-информации, что займет больше времени. В противном случае браузер попытается достигнуть разумного компромисса между точностью и скоростью получения данных, используя любой из следующих источников информации по мере их доступности:

- ✓ спутниковое GPS-позиционирование;
- ✓ ваше беспроводное сетевое соединение;
- ✓ базовая станция, с которой соединен ваш телефон или устройство;
- ✓ IP-адрес вашего устройства или компьютера.

Применение геолокации

Ключевую роль в использовании геолокации играет метод `getCurrentPosition()` объекта `navigator.geolocation`, который может получать три аргумента.

- ✓ `success`. Функция обратного вызова, которой в случае успешной попытки геолокации передается объект `Position`.
- ✓ `error`. Необязательная функция обратного вызова, которой в случае неудачной попытки геолокации передается объект `PositionError`.
- ✓ `options`. Необязательный объект `PositionOptions`, который может использоваться для управления несколькими аспектами геолокационного поиска.

Объект `Position`, возвращаемый методом `getCurrentPosition()`, содержит два свойства.

- ✓ `Position.coords`. Содержит объект `Coordinates`, описывающий местоположение.
- ✓ `Position.timestamp`. Время получения данных о местоположении.

В листинге 17.1 показано, как использовать метод `getCurrentPosition()` для получения объекта `Position` и организовать цикл для просмотра значений, возвращаемых свойством `Position.coords`.

Листинг 17.1. Получение информации о местоположении и ее отображение в браузере

```
<html>
<head>
  <title>Объект Position</title>
  <script>
    var gps = navigator.geolocation.getCurrentPosition(

    function (position) {
      for (key in position.coords) {
        document.write(key+': '+ position.coords[key]);
        document.write ('<br>');
      }
    });
  </script>
</head>
<body>

</body>
</html>
```

Если устройство, на котором вы выполняете этот код, поддерживает геолокацию и браузер может определить ваше местоположение, то вы получите результаты, аналогичные тем, которые представлены на рис. 17.4.

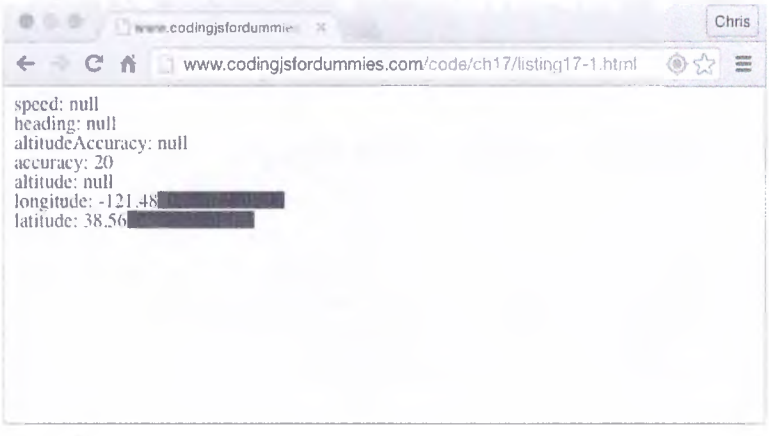


Рис. 17.4. Вывод возвращенных значений объекта `Position`

Обратите внимание на то, что некоторые из свойств объекта `Coordinates`, отображенных на рис. 17.4, имеют значение `null`. Это объясняется тем, что приведенные результаты были получены на настольном компьютере, для которого некоторые координаты недоступны. Результат выполнения того же сценария в мобильном браузере, установленном на смартфоне, представлен на рис. 17.5.

Заметьте, что мобильный браузер дополнительно отображает данные для высоты над уровнем моря, тогда как направление и скорость движения по-прежнему имеют нулевые значения. Разумеется, это объясняется тем, что во время загрузки страницы устройство оставалось неподвижным.

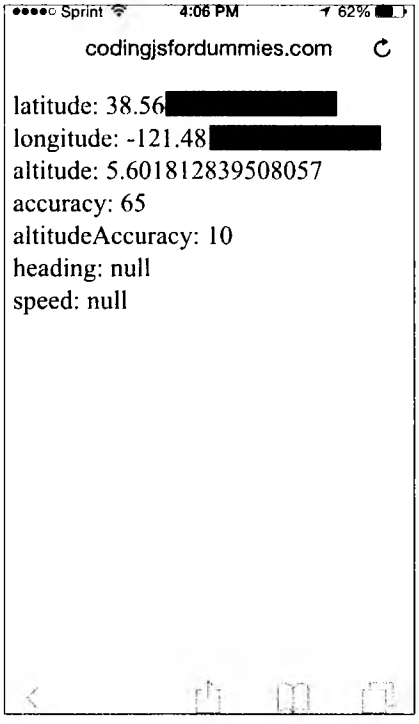


Рис. 17.5. Получение геолокационных данных с помощью браузера смартфона

Сочетание геолокации с картами Google

Одна из часто встречающихся задач, которую приходится решать программистам, работающим с геолокационными данными, — это отображение местоположения устройства на карте. Прежде всего для этого необходимо получить широту и долготу географического местоположения устройства. На данном этапе мы уже решили эту задачу. Но как отобразить карту и определить, где именно на ней находится точка, соответствующая полученным значениям широты и долготы? По-видимому, эта задача не из легких.

К счастью, нашлись люди, которые уже сделали это за вас и создали API для взаимодействия с программным обеспечением, поддерживающим карты. Всемирно известная компания Google бесплатно предоставляет доступ к этому программному обеспечению всем желающим через программный интерфейс Google Maps API (в большинстве случаев даже для коммерческих целей).

Прежде чем вы сможете воспользоваться Google Maps API, необходимо выполнить описанную ниже процедуру.

1. **Перейдите к консоли Google APIs по адресу `http://code.google.com/apis/console` и выполните вход, используя свою учетную запись Google.**
2. **Если после выполнения входа вам будет предложено согласиться с условиями использования консоли, примите условия.**
3. **Щелкните на ссылке Enable API (Включить API).**

   На экране отобразится список API и окно поиска.
4. **Введите в окне поиска текст `Google Maps JavaScript API v3`.**

   Отобразится ссылка для указанного API.
5. **Щелкните на кнопке с подписью OFF, которая находится под заголовком окна состояния.**

   Тем самым вы активизируете данный API.

   Об активизации Google Maps API можно судить по появлению надписи ON зеленого цвета рядом с именем API (рис. 17.6).
6. **Щелкните на ссылке Credentials (Параметры доступа) в левой панели навигации.**

   Отобразится экран API Access.
7. **Щелкните на ссылке Create New Key (Создать новый ключ).**

   Откроется диалоговое окно Create a New Key.
8. **Щелкните на кнопке Browser key (Ключ браузера) в диалоговом окне Create New Key.**

   Откроется окно с полем `Accept requests from these HTTP Referrers` для ввода текста.
9. **Оставьте поле `Accept requests from these HTTP referrers` пустым и щелкните на кнопке Create (Создать).**

   Диалоговое окно закроется, и для вас будет создан ключ API.

   В разделе `Public API access` в поле `Key for browser applications` вы увидите длинную строку, состоящую из букв и цифр, которая является вашим ключом API.

Ключ API — это все, что вам требуется для получения доступа ко всей функциональности Google Maps API.

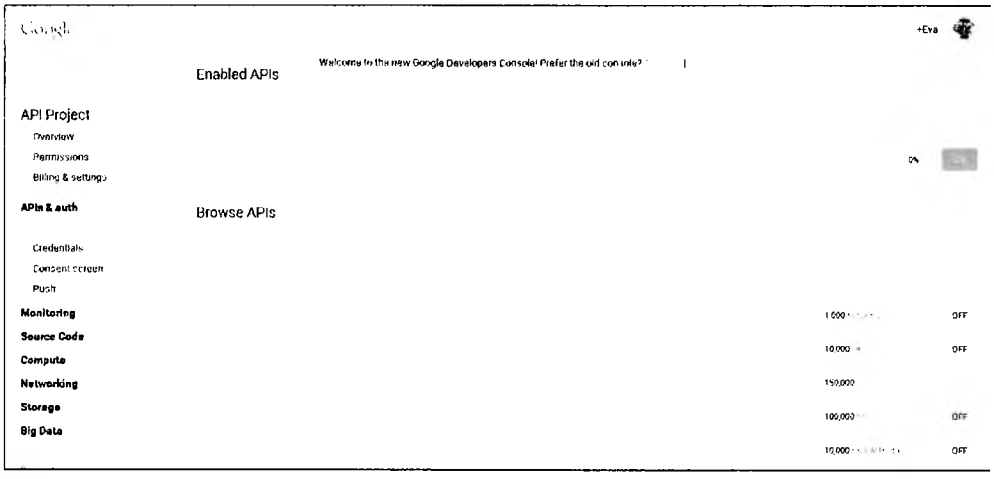


Рис. 17.6. Активизация Google Maps API

Теперь, когда вы располагаете доступом к Google Maps API, можете испытать новые возможности на практике. Приведенная в листинге 7.2 веб-страница определяет местоположение вашего компьютера с помощью объекта `navigator.geolocation` и передает эту информацию на сайт Google Maps для получения карты. Обратите внимание на выделенную область кода, в которую вы должны вставить свой ключ API.

Листинг 17.2. Отображение местоположения на карте с помощью программных интерфейсов `Geolocation API` и `Google Maps API`

```
<html>
<head>
  <title>Отображение вашего местоположения на карте</title>
  <style type="text/css">
    html, body, #map-canvas
        { height: 100%; margin: 0; padding: 0;}
  </style>

  <script type="text/javascript"
      src="https://maps.googleapis.com/maps/api/js?
          key=ВАШ_КЛЮЧ_API">
  </script>

  <script>

  // выполнить функцию initialize после загрузки карты
  google.maps.event.addDomListener(window, "load", initialize);

  function initialize() {

    // получить объект Position и отправить его функции
    // обратного вызова
    var gps = navigator.geolocation.getCurrentPosition(
```

```
    // функция обратного вызова
    function (position) {

      // установить параметры Google Map, используя значения
      // широты и долготы из объекта Position
      var mapOptions = {
        center: { lat: position.coords.latitude,
            lng: position.coords.longitude}, zoom: 8};

      // создать карту и загрузить ее в элемент map-canvas div
      // элемента <body>
      var map = new google.maps.
          Map(document.getElementById("map-canvas"), mapOptions);
    }
  );
};
</script>

</head>
<body>
  <div id="map-canvas"></div>
</body>
</html>
```

Чтобы этот сценарий работал корректно, вы должны заменить текст `ВАШ_КЛЮЧ_API` своим ключом API, полученным от Google.

Результат выполнения листинга 17.2 в браузере представлен на рис. 17.7.

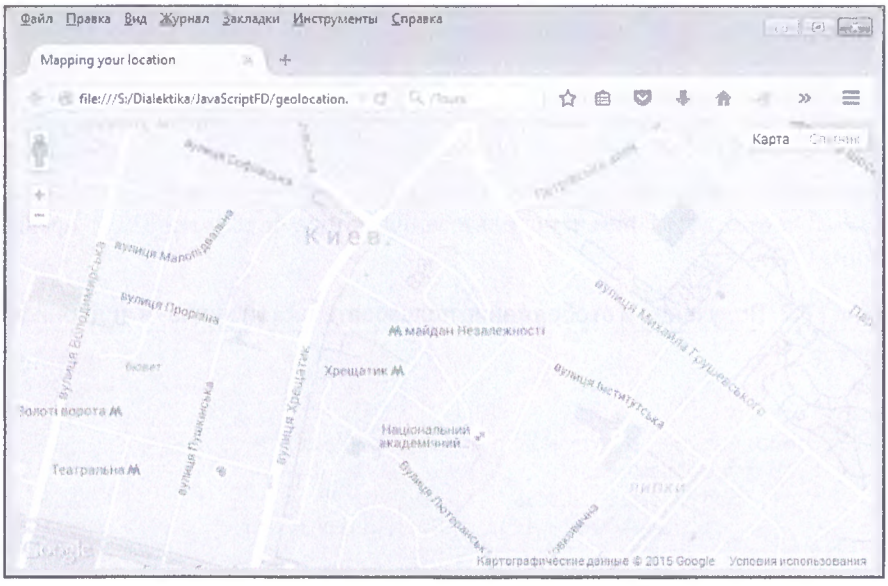


Рис. 17.7. Определение своего местоположения на карте

Работа со звуком и видео

До появления HTML5 единственным способом отображения на веб-страницах видеоинформации, поступающей от подключенной или встроенной видеокамеры компьютера, было использование плагинов, таких как Flash.

Одна из главных задач HTML5 — устранение необходимости в привлечении плагинов, нуждающихся в постоянных обновлениях и создающих бреши в системе безопасности компьютера. С момента выпуска первого варианта набора спецификаций HTML5 неоднократно делались попытки определить стандарт, регламентирующий использование видеокамер.

Самая последняя версия API, обеспечивающего обмен аудио- и видеоинформацией между браузерами в режиме реального времени, носит название WebRTC (Web Real Time Communications).

В стандарте WebRTC ключевая роль отводится методу `navigator.getUserMedia()`, который делает именно то, о чем говорит его название, т.е. получает мультимедийную информацию (звук и видео) от пользователя (или, если говорить точнее, от пользовательского устройства).

В настоящее время метод `navigator.getUserMedia()` поддерживается браузерами Chrome, Opera и Firefox. Его использование в других браузерах, таких как Safari или Internet Explorer, требует принятия специальных мер (технология полизаполнения), о которых можно прочитать на сайте книги:

`http://www.dummies.com/extras/codingwithjavascript`

Первым параметром метода `getUserMedia()` является объект со свойствами, указывающими на тип мультимедийных данных, к которым необходимо получить доступ. Например, если вы хотите получить доступ как к аудио-, так и к видеоданным, то в качестве первого параметра должен использоваться следующий объект:

```
{video: true, audio: true}
```

Остальные параметры метода `getUserMedia()` — функции обратного вызова `success()` и `error()`. Пример использования метода `getUserMedia()` представлен в листинге 17.3.

Листинг 17.3. Получение и отображение пользовательской аудио- и видеоинформации

```
<!DOCTYPE html>
<html>
  <head>
    <title>Получение мультимедийных данных</title>
      <style type="text/css">
        html, body, #map-canvas
            { height: 100%; margin: 0; padding: 0;}
      </style>

    <script>
```

```
    window.addEventListener('DOMContentLoaded', function() {
    var v = document.getElementById('v');
    navigator.getUserMedia = (navigator.getUserMedia ||
                              navigator.webkitGetUserMedia ||
                              navigator.mozGetUserMedia ||
                              navigator.msGetUserMedia);
    if (navigator.getUserMedia) {
       // Запросить доступ только к видео
       navigator.getUserMedia(
          {
             video:true,
             audio:false
          },
          function(stream) {
             var url = window.URL || window.webkitURL;
             v.src = url ? url.createObjectURL(stream) : stream;
             v.play();
          },
          function(error) {
             alert('Что-то пошло не так. (код ошибки '
                 + error.code + ')');
             return;
          }
       );
    }
    else {
       alert('Извините, ваш браузер не поддерживает getUserMedia');
       return;
    };
});
    </script>
  </head>
  <body>
    <video id = "v" autoplay></video>
  </body>
</html>
```

Рассмотрим последовательно все ключевые строки кода, приведенного в листинге 17.3.

```
window.addEventListener('DOMContentLoaded', function() {
```

Это слушатель событий, задача которого — дождаться загрузки DOM, прежде чем запустить остальной код.

```
var v = document.getElementById('v');
```

В этой строке создается новая переменная `v`, хранящая ссылку на элемент `video` с идентификатором `id = "v"`.

```
navigator.getUserMedia = (navigator.getUserMedia ||
          navigator.webkitGetUserMedia ||
          navigator.mozGetUserMedia ||
          navigator.msGetUserMedia);
```

Метод `getUserMedia()` — экспериментальная технология, которая еще не полностью стандартизирована. В силу этого ее версии, реализованные в разных браузерах, различаются, и для их распознавания используют префиксы поставщиков. В приведенной выше инструкции стандартному объекту `getUserMedia` присваивается значение, которое соответствует версии, поддерживаемой браузером пользователя. Поэтому, если вы используете Firefox и вызываете метод `navigator.getUserMedia()`, то фактически вызываете метод `navigator.mozGetUserMedia`.

```
if (navigator.getUserMedia) {
```

В этой строке проверяется, поддерживает ли браузер пользователя метод `getUserMedia()`.

```
navigator.getUserMedia(
```

Вызов метода `getUserMedia()` с параметрами выглядит так.

```
{
  video:true,
  audio:false
}
```

Первый параметр — объект, содержащий информацию о том, к какому типу мультимедийных данных вы хотите получить доступ.

```
function(stream) {
```

Это функция обратного вызова, которая вызывается в случае успешного выполнения запроса с помощью метода `getUserMedia()`. Она принимает единственный аргумент.

```
var url = window.URL || window.webkitURL;
v.src = url ? url.createObjectURL(stream) : stream;
```

Предыдущие две строки устраняют различия в способах обработки объекта `stream` различными браузерами. Во второй строке используется наш хороший друг — тернарный оператор! В этой инструкции свойству `src` присваивается либо значение `url.createObjectUrl(stream)`, либо значение `stream`, в зависимости от того, какой из этих методов поддерживается браузером.

```
v.play();
```

Наконец, воспроизведение видео! Если ваш компьютер поддерживает метод `getUserMedia()` и у вас установлена видеокамера, то в этот момент вы должны увидеть самого себя (или то, что попадает в поле зрения видеокамеры).

```
function(error) {
  alert('Что-то пошло не так. (код ошибки ' + error.code + ')');
  return;
}
```

Предыдущий код — это функция обратного вызова, обрабатывающая ошибки. Если браузер поддерживает метод `getUserMedia()`, но пользователь отказал браузеру в доступе к камере, то она выведет соответствующее сообщение.

```
else {
      alert('Извините, ваш браузер не поддерживает getUserMedia');
      return;
   };
```

Этот код представляет ветвь `else` условной инструкции. Если метод `getUserMedia()` не поддерживается браузером пользователя, то отобразится приведенное выше сообщение.

Если же браузер пользователя поддерживает метод `getUserMedia()`, видеокамера установлена и пользователь разрешил приложению доступ к ней, то приложение отобразит видео, как в примере на рис. 17.8.

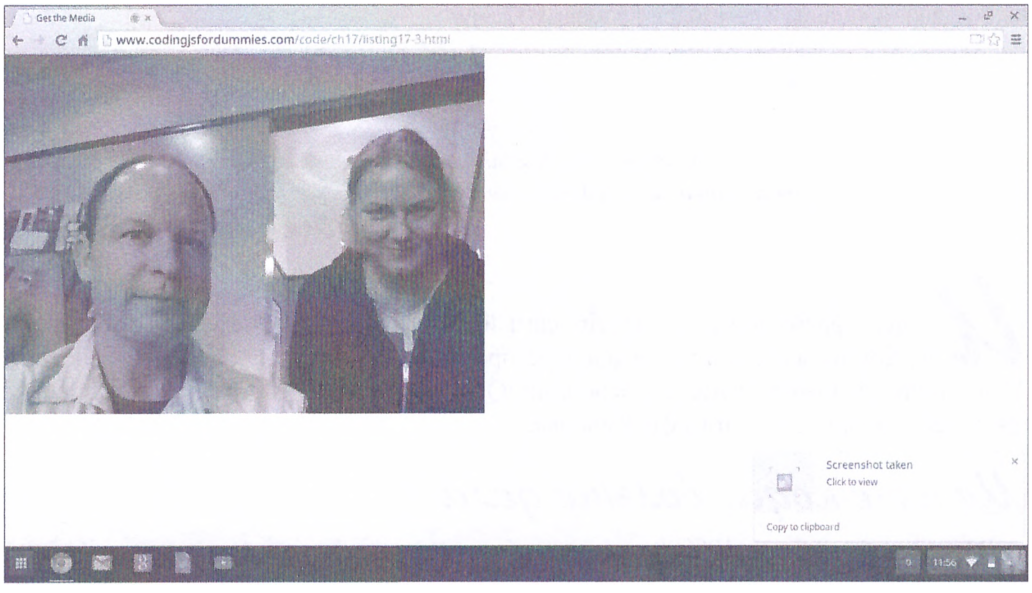


Рис. 17.8. Успех! Браузер отображает видео в режиме реального времени без использования плагинов

Глава 18

Библиотека jQuery

В этой главе...

- Что представляет собой библиотека jQuery
- Выбор элементов
- Создание анимации и переходов с помощью jQuery

"Лучше носить с собой все инструменты. А то попадется что-то неожиданное, и будешь смотреть на него, как баран на новые ворота".

Стивен Кинг

Из всех фреймворков JavaScript самый популярный — jQuery. Почти все программисты используют его для ускорения и упрощения процесса разработки. В этой главе вы познакомитесь с основами jQuery и поймете, что именно обусловило такой небывалый успех этого фреймворка.

Меньше кода, больше дела

В настоящее время из 100000 ведущих веб-сайтов доля тех, где используется jQuery, превышает 61%. Популярность этого фреймворка настолько велика, что многие считают его одним из важнейших инструментальных средств JavaScript-программистов.

Библиотека jQuery сглаживает некоторые острые углы JavaScript, такие как кроссбраузерная совместимость, и значительно упрощает выбор и изменение отдельных частей HTML-документа, позволяя выполнять больший объем работы при меньшем количестве кода. Кроме того, jQuery включает некоторые инструменты, которые можно использовать для создания анимации и повышения интерактивности веб-страниц.

Приступаем к работе с jQuery

Первое, что нужно сделать, приступая к работе с библиотекой jQuery, — это добавить ее в документ. Для этого проще всего воспользоваться одной из публично доступных сетей распространения контента (Content Delivery Network — CDN). Другой способ подключения библиотеки jQuery предполагает загрузку библиотеки с сайта jQuery и размещение ее на своем сервере. В листинге 18.1 представлена простая веб-страница, включающая jQuery.

На сайте Google хранятся версии многих библиотек JavaScript. Ссылки на них и фрагменты HTML-разметки для их включения в документы вы найдете по следующему адресу:

`http://developers.google.com/speed/libraries`

Найдя ссылку на нужную версию библиотеки, хранящуюся в CDN, поместите ее между тегами <head> и </head> на каждой из страниц, в которых используется функциональность jQuery.

В настоящее время существуют две линейки версий jQuery: линейка 1.*x* и линейка 2.*x*. Различие между самой последней версией из линейки 1.*x* и самой последней версией из линейки 2.*x* состоит в том, что версии 1.*x* работают с браузерами Internet Explorer 6-8, тогда как в версиях 2.*x* поддержка этих устаревших браузеров полностью отсутствует.

Листинг 18.1. Ваша первая страница с jQuery

```
<html>
<head>
  <title>Привет, jQuery</title>
  <style>
    #helloDiv {
      background: #333;
      color: #fff;
      font-size: 24px;
      text-align: center;
      border-radius: 3px;
      width: 200px;
      height: 200px;
      display: none;
    }
  </style>
  <script src="http://code.jquery.com/jquery-
          1.11.2.min.js"></script>
</head>
<body>
  <button id="clickme">Щелкни на мне!</button>
  <div id="helloDiv">Привет, jQuery!</div>

  <script>
  $( "#clickme" ).click(function () {
    if ( $( "#helloDiv" ).is( ":hidden" ) ) {
      $( "#helloDiv" ).slideDown( "slow" );
    } else {
      $( "div" ).hide();
    }
  });
  </script>
</body>
</html>
```

Объект jQuery

В основе всей функциональности jQuery лежит объект jQuery. На этот объект можно ссылаться двояким образом: с помощью ключевого слова `jQuery` или псевдонима `$`. Оба способа приводят к одному и тому же результату. Запись с помощью псевдонима `$` выгодно отличается лишь краткостью, и именно поэтому она пользуется большей популярностью среди программистов, работающих с jQuery.

Базовый синтаксис jQuery выглядит примерно так:

```
$("selector").method();
```

Первая часть (в скобках) указывает, на какие элементы вы хотите воздействовать, а вторая — что именно следует сделать с этими элементами.

На практике команды jQuery часто выполняют сразу несколько действий над набором выбранных элементов за счет объединения методов в *цепочки*, в которых методы разделяются символом точки. Например, в листинге 18.2 цепочка методов используется для того, чтобы сначала выбрать единственный элемент (с идентификатором `pageHeader`), а затем применить к нему стиль.

Листинг 18.2. Использование цепочки методов

```
<html>
<head>
  <title>Объединение методов jQuery в цепочки</title>
  <script src="http://code.jquery.com/jquery-
          1.11.2.min.js"></script>
</head>
<body>
  <div id="pageHeader"/>
  <script type="text/javascript">
    $("#pageHeader").text("Привет, мир!").css("color",
          "red").css("font-size",
          "60px");
  </script>
</body>
</html>
```

Цепочки методов jQuery могут быть довольно длинными, и уже после объединения всего лишь двух методов чтение кода затрудняется. Следует, однако, иметь в виду, что JavaScript терпимо относится к пробелам. Поэтому цепочку, фигурирующую в листинге 18.2, можно переформатировать, придав ей более удобочитаемый вид:

```
$("#pageHeader")
  .text("Привет, мир!")
  .css("color","red")
  .css("font-size","60px");
```

Готов ли документ к работе?

Для индикации того, что все необходимые компоненты загружены и готовы к работе, в jQuery предусмотрено специальное событие `ready`. Чтобы избежать ошибок, связанных с тем, что к моменту выполнения сценария процесс загрузки DOM или jQuery еще не был завершен, очень важно использовать событие `ready` в тех сценариях с инструкциями jQuery, которые не находится в самом конце документа (как это было сделано в листингах 18.1 и 18.2).

Вот как выглядит соответствующий синтаксис.

```
$(document).ready(function(){

  // Сюда помещаются вызовы методов jQuery...

});
```

Любой код jQuery, который должен выполняться лишь после загрузки страницы, следует помещать в инструкцию `(document).ready(...);`. Разумеется, именованные функции могут объявляться вне функции `ready()`, поскольку они не выполняются до тех пор, пока не будут вызваны.

Использование селекторов jQuery

В отличие от сложных и в то же время ограниченных по своим возможностям средств JavaScript, предназначенных для выбора элементов, jQuery чрезвычайно упрощает эту задачу. В jQuery программисты могут использовать те же способы выбора элементов, что и в CSS. Перечень наиболее часто используемых селекторов jQuery и CSS приведен в табл. 18.1.

В дополнение к базовым селекторам можно изменять или комбинировать отдельные части набора выбранных элементов множеством других способов. Например, чтобы выбрать первый из элементов `p`, встречающихся в документе, можно использовать следующую запись:

```
$('p:first')
```

Таблица 18.1. Часто используемые селекторы jQuery/CSS

Селектор	Пример HTML	Пример jQuery
`element`	`<p></p>`	`$('p').css('font-size','12')`
`.class`	`<p class="redtext"></p>`	`$('.redtext').css('color','red')`
`#id`	`<p id="intro"></p>`	`$('#intro').fadeIn('slow)`
`[attribute]`	`<p data-role="content"></p>`	`$('[data-role]').show()`

Приведем некоторые примеры. Выбор последнего элемента `p`:

```
$('p:last')
```

Выбор четных (в отношении порядка следования) элементов:

```
$('li:even')
```

Выбор нечетных (в отношении порядка следования) элементов:

```
$('li:odd)
```

Можно комбинировать несколько селекторов, разделяя их запятыми. Например, следующий селектор выбирает все элементы `p`, `h1`, `h2` и `h3`, содержащиеся в документе:

```
$('p,h1,h2,h3')
```

По сравнению с обычным JavaScript в jQuery гораздо больше возможностей выбора элементов. Полный список селекторов jQuery вы найдете в шпаргалке в конце книги.

Изменение документа с помощью jQuery

Следующей по важности после выбора элементов задачей является изменение выбранного набора. Средства jQuery позволяют изменять атрибуты, стили CSS и элементы.

Получение и установка значений атрибутов

Метод `attr()` обеспечивает доступ к значениям атрибутов. Все, что для этого требуется, — предоставить методу имя атрибута, значение которого вы хотите получить или установить. В приведенном ниже коде метод `attr()` используется для изменения значения атрибута `href` элемента с идентификатором `"homepage-link"`.

```
$('a#homepage-link').attr('href') =
          "http://www.codingjsfordummies.com/";
```

В результате выполнения этой инструкции атрибуту `href` выбранного элемента будет присвоено новое значение в DOM. Если пользователь щелкнет на измененной ссылке, то браузер откроет страницу, находящуюся по указанному адресу, а не по тому, который первоначально был определен в атрибуте.

Изменения элементов с помощью jQuery затрагивают лишь представление элементов в DOM (а значит, и изображение на экране пользователя). При этом сама веб-страница на сервере остается неизменной, и если вы просмотрите ее исходный код, то не заметите в нем никаких изменений.

Изменение стилей CSS

Изменение CSS-стилей средствами jQuery весьма напоминает технику изменения свойств объекта `Style`, описанную в главе 13. Управлять стилевыми свойствами с помощью jQuery гораздо проще, чем стандартными средствами JavaScript, и при этом стилевые свойства имеют те же названия, что и в CSS.

В листинге 18.3 изменение активных свойств CSS-стилей в сочетании с событиями формы предоставляет пользователю возможность управлять размером шрифта отображаемого текста.

Листинг 18.3. Манипулирование стилями с помощью jQuery

```
<html>
<head>
  <title>JQuery CSS</title>
  <script src="http://code.jquery.com/jquery-1.11.2.min.js">
  </script>
  <script type="text/javascript">
    $(document).ready(function(){

    $('#sizer').change(function() {
      $('#theText').css('font-size',$('#sizer').val());
    });
    });
  </script>
</head>
<body>
  <div id="theText">Привет!</div>
  <form id="controller">
    <input type="range" id="sizer" min="10" max="100">
  </form>
</body>
</html>
```

Результат выполнения листинга 18.1 в браузере представлен на рис. 18.1.

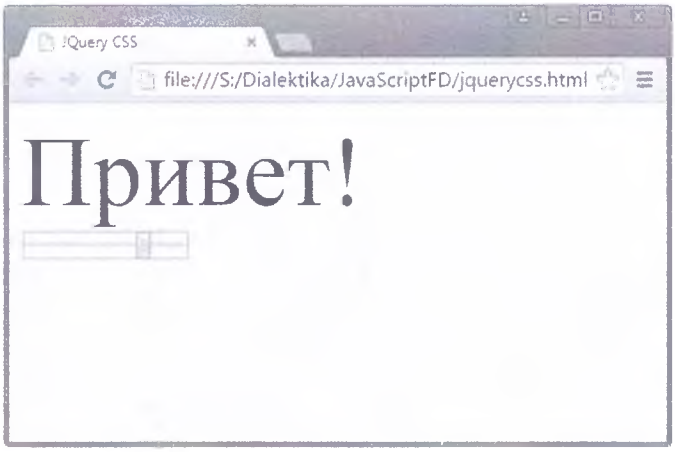


Рис. 18.1. Изменение CSS-стиля с помощью элемента `input`

Манипулирование элементами в DOM

Библиотека jQuery располагает рядом методов, с помощью которых можно изменять содержимое элементов, а также перемещать, добавлять, масштабировать элементы и выполнять над ними много других операций. В табл. 18.2 приведены все методы, предназначенные для манипулирования элементами в пределах DOM.

Таблица 18.2. Манипулирование элементами в пределах DOM

Метод	Описание	Пример
`text()`	Получает объединенное текстовое содержимое или устанавливает текстовое содержимое элементов выбранного набора	`$('p').text(Привет!')`
`html()`	Получает значение первого выбранного элемента или устанавливает значение каждого выбранного элемента	`$('div').html('<p>hi</p>')`
`val()`	Получает значение первого выбранного элемента или устанавливает значение всех выбранных элементов	`$('select#choices').val()`
`append()`	Вставляет содержимое в качестве последних дочерних элементов во все выбранные элементы	`$('div #closing'). append('<p>Благодарим вас</p>')`
`prepend()`	Вставляет содержимое в качестве первых дочерних элементов во все выбранные элементы	`$('div #introduction'). prepend('<p>Всем заинтересованным лицам:</p>')`
`before()`	Вставляет содержимое перед каждым выбранным элементом	`$('#letter').before(header)`
`after()`	Вставляет содержимое после каждого выбранного элемента	`$('#letter').after(footer)`
`remove()`	Удаляет выбранные элементы	`$('.phonenumber').remove()`
`empty()`	Удаляет все дочерние узлы выбранных элементов	`$('.blackout').empty()`

События

В главе 11 мы обсуждали различные методы регистрации обработчиков событий в JavaScript, которые отлично работают и в jQuery. Однако в jQuery имеется собственный синтаксис для регистрации слушателей событий и обработки событий.

Метод `on()` в jQuery берет на себя обеспечение кросс-браузерной совместимости и гарантирует одинаковую обработку событий во всех браузерах, а кроме того, требует гораздо меньшего объема ввода по сравнению с решениями, основанными исключительно на JavaScript.

Использование метода `on()` для подключения событий

Метод `on()` в jQuery работает во многом аналогично методу `addEventListener()`. Он принимает событие и определение функции в качестве аргументов. Когда происходит событие, связанное с выбранным элементом (или элементами), выполняется указанная функция. В листинге 18.4 метод `on()` и селектор `jQuery` используются для изменения цвета строк таблицы через одну после щелчка на кнопке.

Листинг 18.4. Изменение цвета строк таблицы щелчком на кнопке

```
<html>
<head>
  <title>jQuery CSS</title>
  <style>
    td {
      border: 1px solid black;
    }
  </style>
  <script src="http://code.jquery.com/jquery-1.11.2.min.js">
  </script>
  <script type="text/javascript">
    $(document).ready(function(){

      $('#colorizer').on('click',function() {
        $('#things tr:even').css('background','yellow');
      });

    });

  </script>
</head>
<body>
  <table id="things">
    <tr>
      <td>элемент1</td>
      <td>элемент2</td>
      <td>элемент3</td>
    </tr>
    <tr>
      <td>яблоки</td>
      <td>апельсины</td>
      <td>лимоны</td>
    </tr>
    <tr>
      <td>мерло</td>
      <td>мальбек</td>
      <td>каберне совиньон</td>
    </tr>
  </table>
  <form id="tableControl">
    <button type="button" id="colorizer">Колоризовать</button>
  </form>
</body>
</html>
```

Вид таблицы с чередованием цвета строк, полученным после щелчка на кнопке, представлен на рис. 18.2.

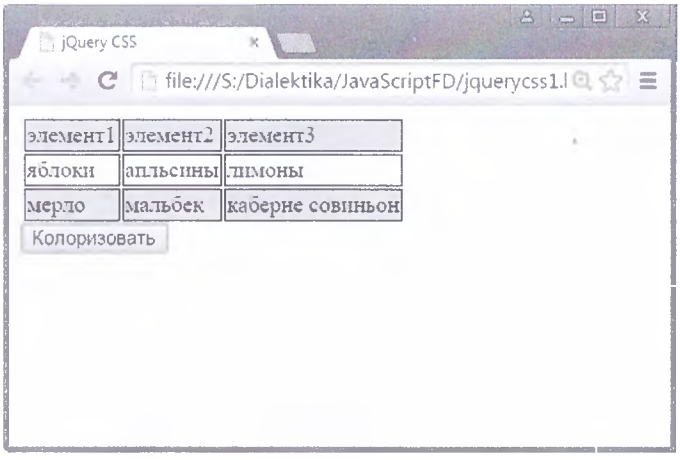


Рис. 18.2. Чередование цвета строк таблицы

Вы заметили нечто странное в колоризации строк на рис. 18.2? Окрашенными оказались первая и третья строки, хотя мы задали колоризацию четных (`even`) строк. Все объясняется очень просто. В данном случае понятие четности основано на числовых индексах элементов `tr`, индексация которых всегда начинается с нуля. Поэтому окрашиваются первая (индекс 0) и третья (индекс 2) строки.

Открепление событий с помощью метода `off()`

Метод `off()` позволяет отменить регистрацию ранее установленного слушателя событий. Например, если вы захотите отключить функциональность кнопки в листинге 18.4 (возможно, до тех пор, пока пользователь не внесет плату за использование этого средства), это можно сделать с помощью следующего кода:

```
$('#colorizer').off('click');
```

Также существует возможность удалить все слушатели событий, связанные с данным элементом, опустив аргументы при вызове метода:

```
$('#colorizer').off();
```

Привязка событий к еще не существующим элементам

В силу динамической природы современного Интернета иногда возникает необходимость в регистрации событий для элементов, которые динамически создаются уже после того, как загрузится HTML-код.

Для добавления слушателей событий в динамически создаваемые элементы методу `on()` можно передавать селекторы элементов, создание которых будет отслеживаться. Например, если вы хотите, чтобы все существующие строки таблицы, а также строки, которые будут созданы впоследствии, реагировали на щелчки, используйте следующий код.

```
$(document).on('click','tr',function(){
  alert("Спасибо за щелчок!");
}
```

Другие методы для работы с событиями

Помимо метода `on()`, в jQuery предусмотрен простой сокращенный синтаксис подключения слушателей событий к выбранным элементам. В соответствии с этим синтаксисом вам достаточно передать обработчик события методу, имя которого совпадает с именем события. Например, обе приведенные ниже инструкции делают одно и то же.

```
$('#myButton').on('click',function() {
  alert('Спасибо!');
}
$('#myButton').click(function() {
  alert('Спасибо!');
}
```

Существуют и другие сокращенные методы:

- ✓ `change()`
- ✓ `click()`
- ✓ `dbclick()`
- ✓ `focus()`
- ✓ `hover()`
- ✓ `keypress()`
- ✓ `load()`

Чтобы ознакомиться с полным списком методов для работы с событиями, посетите сайт jQuery по следующему адресу:

`http://api.jquery.com/category/events`

Эффекты

Библиотека jQuery значительно облегчает жизнь JavaScript-программистам. Она упрощает даже создание анимации и эффектов.

В силу простоты создания эффектов jQuery ими часто злоупотребляют. После того как вы ознакомитесь со всеми эффектами и поэкспериментируете с каждым из них, было бы, пожалуй, неплохо, чтобы вы создали веб-приложение, в котором эффекты используются всякий раз, когда происходит какое-либо событие. После этого смело удалите этот файл, и пусть приобретенный опыт умерит ваше желание использовать эффекты даже там, где в них нет особой необходимости.

Базовые эффекты

К базовым мы относим эффекты jQuery, позволяющие управлять видимостью элементов. Сюда входят следующие эффекты.

- ✓ `hide()`. Метод `hide()` скрывает элементы выбранного набора.
- ✓ `show()`. Метод `show()` отображает элементы выбранного набора.
- ✓ `toggle()`. Метод `toggle()` переключает режим видимости элементов выбранного набора. Если выбранный элемент скрыт, то метод `toggle()` сделает его видимым. Если же он отображается, то метод `toggle()` скроет его.

Эффекты затухания

Эффекты затухания (растворения) обеспечивают отображение или сокрытие элементов за счет плавного изменения непрозрачности (или, если вам так больше нравится, прозрачности) элементов. К ним относятся следующие эффекты.

- ✓ `fadeIn()`. Метод `fadeIn()` отображает элементы выбранного набора (делает их полностью непрозрачными) в течение заданного промежутка времени.
- ✓ `fadeOut()`. Метод `fadeOut()` скрывает элементы выбранного набора (делает их полностью прозрачными) в течение заданного промежутка времени.
- ✓ `fadeTo()`. Метод `fadeTo()` изменяет непрозрачность элементов выбранного набора до заданного уровня в течение заданного промежутка времени.
- ✓ `fadeToggle()`. Метод `fadeToggle()` увеличивает или уменьшает непрозрачность элементов выбранного набора в течение заданного промежутка времени.

Эффекты скольжения

Эффекты скольжения позволяют управлять отображением и сокрытием элементов выбранного набора за счет плавного изменения их высоты. К ним относятся следующие эффекты.

- ✓ `slideDown()`. Метод `slideDown()` отображает элементы выбранного набора путем их плавного раскрытия на странице в направлении вниз.
- ✓ `slideUp()`. Метод `slideUp()` скрывает элементы выбранного набора путем их плавного свертывания на странице в направлении вверх.

- ✓ `slideToggle()`. Метод `slideToggle()` изменяет состояние видимости элементов выбранного набора путем их плавного свертывания в направлении вверх и раскрытия в направлении вниз.

Задание аргументов анимационных методов

Для каждого из анимационных методов jQuery предусмотрен набор необязательных аргументов, позволяющих управлять детальными характеристиками анимации.

Методы базовых эффектов, а также эффектов затухания и скольжения принимают следующие аргументы.

- ✓ `duration`. Числовой параметр, задающий длительность периода анимации (в миллисекундах).
- ✓ `easing`. Строковое значение, указывающее тип функции, используемой для сглаживания анимационного перехода. Функция сглаживания определяет скорость анимации элемента на различных стадиях. Например, анимационный переход может начинаться медленно, а затем ускоряться, или же начинаться быстро, а затем замедляться. В jQuery имеются две встроенные функции сглаживания.
  - `swing` (значение по умолчанию). На начальном и конечном этапах анимация выполняется несколько медленнее, чем на средних.
  - `swing`. Анимация выполняется с постоянной скоростью.
- ✓ `complete`. Задает функцию, которая должна быть выполнена по завершении анимации.

Создание пользовательских анимационных эффектов с помощью метода `animate()`

Метод `animate()` обеспечивает пользовательскую анимацию CSS-свойств. Вид анимации определяется набором свойств, передаваемых методу `animate()` в качестве аргументов. В процессе выполнения метода анимация движется к значениям, установленным вами для каждого свойства. Например, чтобы анимировать увеличение ширины и изменение цвета элемента `div`, можно использовать следующий код.

```
('div #myDiv').animate(
{
  width: 800,
  color: 'blue'
}, 5000);
```

Кроме аргумента, определяющего требуемые CSS-свойства, метод `animate()` принимает те же необязательные аргументы, что и любой другой анимационный метод.

Пример выполнения анимации средствами jQuery

В листинге 18.5 реализованы некоторые из анимационных эффектов jQuery. Проведите собственные эксперименты, изменяя значения аргументов методов, и понаблюдайте, к чему это приводит[1].

Листинг 18.5. Пример анимации средствами jQuery

```
<html>
<head>
  <title>JQuery CSS</title>
  <style>
  td {
    border: 1px solid black;
  }
  </style>
  <script src="http://code.jquery.com/jquery-
          1.11.2.min.js"></script>
  <script type="text/javascript">
    // дождаться готовности DOM
    $(document).ready(function(){
      // начать выполнение по щелчку на кнопке
      $('#animator').on('click',function() {
        $('#items').fadeToggle(200);
        $('#fruits').slideUp(500);
        $('#wines').toggle(400,'swing',function(){
          $('#wines').toggle(400,'swing');
        });
        $('h1').hide();
        $('h1').slideDown(1000).animate({
          'color': 'red',
          'font-size': '100px'},1000);
      });
    });
</script>
</head>
<body>
<h1>Целая куча всего интересного!</h1>
<table id="things">
  <tr id="items">
    <td>элемент1</td>
    <td>элемент2</td>
    <td>элемент3</td>
  </tr>
  <tr id="fruits">
    <td>яблоки</td>
    <td>апельсины</td>
    <td>лимоны</td>
```

[1] В ходе таких экспериментов не пытайтесь искать в своем коде ошибки, из-за которых вам не удается выполнить анимацию цвета. Стандартные средства jQuery позволяют анимировать лишь свойства, выражаемые простыми числовыми значениями. Анимация цвета с помощью метода `animate()` требует привлечения дополнительных средств. См., например, статью по адресу `http://xiper.net/collect/js-plugins/effects/jquery-color`. — *Примеч. ред.*

```
  </tr>
  <tr id="wines">
    <td>мерло</td>
    <td>мальбек</td>
    <td>каберне совиньон</td>
  </tr>
</table>
<form id="tableControl">
  <button type="button" id="animator">Анимировать!</button>
</form>
</body>
</html>
```

AJAX

Одной из наиболее полезных особенностей библиотеки jQuery является то, что она значительно упрощает работу с AJAX и внешними данными.

Технология AJAX, обеспечивающая загрузку новых данных в веб-страницу без ее полного обновления, обсуждалась в главе 16, в которой также рассматривалось использование JSON-данных в JavaScript.

Использование метода `ajax()`

За предоставление возможностей AJAX в jQuery отвечает метод `ajax()`, обеспечивающий низкоуровневый обмен данными с внешними файлами. В своей простейшей форме метод `ajax()` может принимать имя файла или URL-адрес в качестве аргумента и загружает содержимое указанного файла. После этого можно присвоить загруженное содержимое переменной в своем сценарии.

Кроме того, можно задать дополнительные аргументы, детализирующие особенности загрузки внешних данных с указанного URL-адреса, а также указать функции, которые должны выполняться в случае удачного или неудачного завершения запроса.

Полный перечень дополнительных параметров метода `ajax()` вы найдете по адресу `http://api.jquery.com/jQuery.ajax`.

Приведенный в листинге 18.6 сценарий открывает текстовый файл, содержащий абзац текста, и отображает этот текст в элементе `div`.

Листинг 18.6. Загрузка и отображение внешнего файла с помощью jQuery и AJAX

```
<html>
<head>
  <title>Динамическое приветствие</title>
  <script src="http://code.jquery.com/jquery-1.11.2.min.js">
  </script>
  <script>
    // дождаться окончания загрузки документа
    $(document).ready(function(){
      // запускается по щелчку на кнопке
      $('#loadIt').on('click',function(){
        // получить значение выбранного элемента и присоединить
        // к нему строку .txt
```

```
        var fileToLoad = $('#intros').val() + '.txt';
        // открыть файл с этим именем
        $.ajax({url:fileToLoad,success:function(result){
          // в случае удачного открытия файла отобразить
          // его содержимое
            $('#introtext').html(result);
        }});
      });
    });
  </script>
</head>
<body>
  <h1>Выберите желаемый тип приветствия:</h1>
  <form id="intro-select">
    <select id="intros">
      <option value="none">Пожалуйста, выберите</option>
      <option value="formal">Официальное</option>
      <option value="friendly">Дружеское</option>
      <option value="piglatin">Фамильярное</option>
    </select>
    <button id="loadIt" type="button">Загрузить!</button>
  </form>
  <div id="introtext"></div>
</body>
</html>
```

Если захотите выполнить листинг 18.6 в своем браузере, то у вас это не получится из-за ограничений безопасности, носящих название *политики одного источника*, которые предотвращают загрузку данных по механизму AJAX, если их источник и веб-страница принадлежат к разным доменам (см. главу 16). Чтобы выполнить этот пример, загрузите его на свой веб-сервер, посетив следующий сайт:

```
http://www.codingjsfordummies.com/code/
```

Или отключите в браузере ограничения безопасности.

Сокращенные методы для работы с AJAX

В jQuery имеется несколько специализированных (сокращенных) методов для работы с AJAX. Синтаксис этих методов более компактен, поскольку они ориентированы на выполнение конкретных задач. Ниже приведен перечень этих методов.

- ✓ `get()`. Загружает данные с сервера, используя GET-запрос HTTP.
- ✓ `getJSON()`. Загружает JSON-данные с сервера, используя GET-запрос HTTP.
- ✓ `getScript()`. Загружает JavaScript-файл с сервера, используя GET-запрос HTTP, а затем выполняет его.
- ✓ `post()`. Загружает данные с сервера и помещает возвращенный HTML-документ в соответствующий элемент.

Используя сокращенный метод, вы передаете ему URL-адрес и необязательный обработчик, который запускается в случае удачного выполнения запроса. Например, для получения файла с сервера с использованием метода `get()` и его последующей вставки в веб-страницу воспользуйтесь следующим кодом.

```
$.get( "getdata.html", function( data ) {
  $( ".result" ).html( data );
});
```

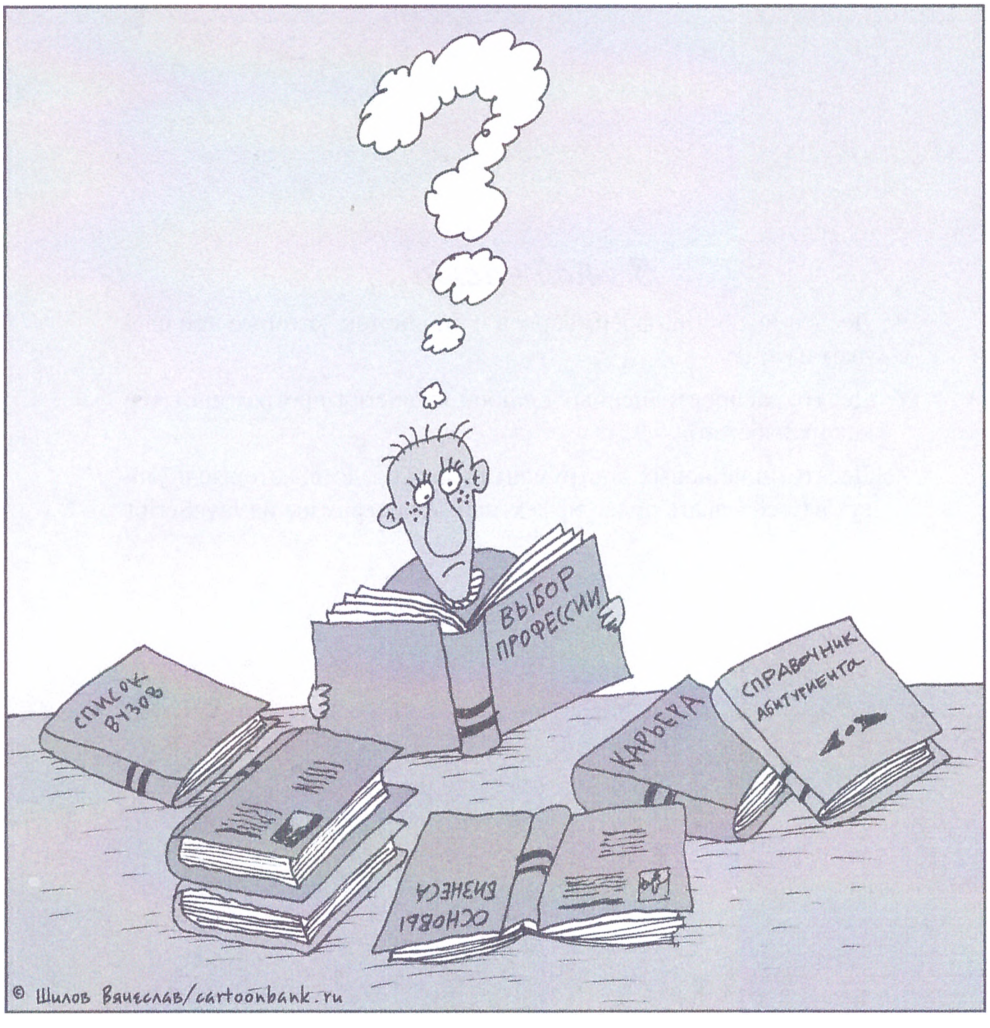
Предыдущий пример эквивалентен выполнению следующей инструкции, в которой используется полная форма метода `ajax()`.

```
$.ajax({
  url: getdata.html,
  success: function( data ) {
  $( ".result" ).html( data );
  }
});
```

В этом примере экономия усилий не очень заметна. Однако в случае более сложных AJAX-запросов использование сокращенной формы метода `ajax()` приводит к созданию более понятного и компактного кода.

Часть VI

Великолепные десятки



В этой части...

- ✓ Десять JavaScript-фреймворков и библиотек, которые вам следует изучить
- ✓ Десять распространенных ошибок JavaScript-программистов и как их избежать
- ✓ Десять онлайновых инструментальных средств, которые помогут вам создавать более эффективные программы на JavaScript

Глава 19

Десять JavaScript-фреймворков и библиотек, которые вам следует изучить

В этой главе...

- Популярные JavaScript-фреймворки и библиотеки
- Знакомство с сайтами, использующими фреймворки и библиотеки JavaScript

"Я бьюсь головой о стены, и стены поддаются".

Густав Малер

Ваше путешествие в увлекательный мир JavaScript только началось. Целая вселенная инструментов, фреймворков и библиотек JavaScript, с помощью которых вы сможете создавать лучшие программы, огромна и быстро расширяется.

В этой главе вы познакомитесь с десятью нашими любимыми фреймворками и библиотеками[1]. Вам необязательно досконально знать все эти платформы, но общее знакомство с ними и свободное владение хотя бы двумя из них окажет вам неоценимую помощь в занятиях веб-программированием.

У каждого из описанных ниже средств имеется свой круг пользователей, почитателей и людей, принимающих участие в его дальнейшем развитии и поддержке. Для каждого средства мы приводим примеры известных сайтов, на которых оно используется.

AngularJS

AngularJS (обычно просто — Angular) — JavaScript-фреймворк с открытым исходным кодом (рис. 19.1). Этот фреймворк, который многие считают библиотекой из-за сравнительно небольшого размера кода, поддерживается компанией Google и сообществом разработчиков.

[1] Библиотека — это универсальный набор программ близкой функциональности, которые могут использоваться в программных продуктах любой архитектуры. Фреймворк (или каркас приложений) — это программная система, включающая несколько библиотек, которая предоставляет начальный каркас приложения, подлежащий последующему расширению пользователем, но при этом диктует правила построения архитектуры приложений. — *Примеч. ред.*

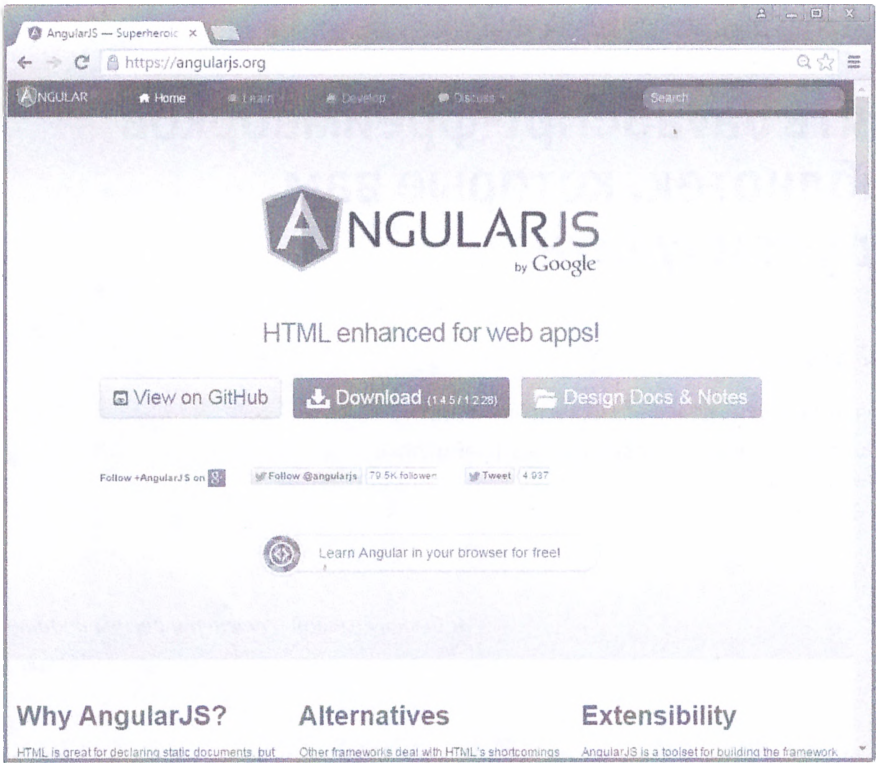


Рис. 19.1. Сайт `http://angularjs.com`

Данный фреймворк адаптирует и дополняет традиционные средства HTML для поддержки динамического содержимого с использованием двухсторонней привязки данных, которая обеспечивает автоматическую синхронизацию модели (данные) и представления (веб-страницы). Следствием этого является отделение DOM-манипуляций от бизнес-логики приложения, что улучшает тестируемость и производительность кода.

При проектировании AngularJS ставились следующие основные цели:

- ✓ улучшить тестируемость кода за счет отделения DOM-манипуляций от логики приложения;
- ✓ уравнять значимость тестирования и написания кода, т.е. предложить технологию разработки, в которой тестирование кода считается не менее важной задачей, чем его создание;
- ✓ отделить клиентскую часть приложения от серверной;
- ✓ предоставить инфраструктуру, поддерживающую весь процесс создания приложений — от проектирования пользовательского интерфейса, через написание бизнес-логики, до тестирования.

Где используется этот фреймворк? Сайты `YouTube.com`, `Lynda.com`, `Netflix.com` и `freelancer.com`.

Backbone.js

Backbone.js (рис. 19.2) — это JavaScript-библиотека с открытым исходным кодом на основе MVC, предназначенная для создания одностраничных приложений[2]. Библиотека Backbone обеспечивает разработчика структурой приложения и стимулирует следование весьма действенному принципу, суть которого состоит в том, что обмен данными с сервером должен осуществляться посредством RESTful API.

В результате использования библиотеки Backbone ваш код приобретает более выраженную модульную структуру, что обеспечивает эффективную организацию процесса создания и сопровождения очень сложных веб-приложений при минимальном объеме кода.

Библиотека Backbone имеет лишь одну зависимость (`underscore.js`) и налагает лишь самую минимальную дополнительную нагрузку на веб-приложение.

Где используется эта библиотека? Сайты `reddit.com`, `bitbucket.org`, `tumblr.com`, `pintrest.com` и `linkedin.com`.

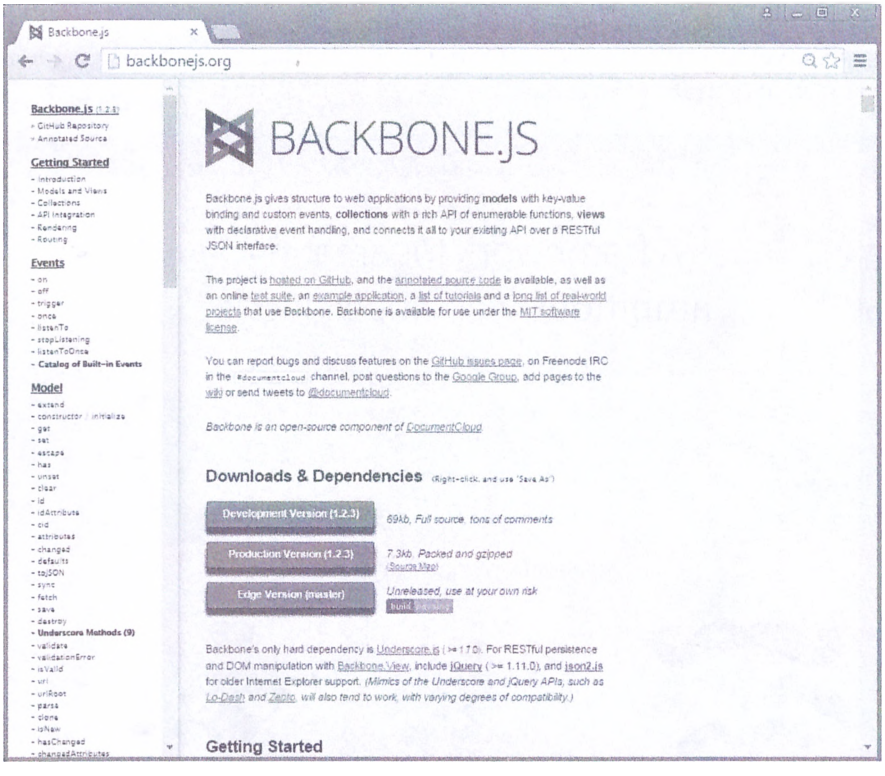


Рис. 19.2. Сайт `http://backbonejs.org`

[2] Одностраничное приложение — это веб-приложение, которое умещается на одной странице и выполняется непосредственно на стороне клиента. Страница загружается один раз вместе со всем необходимым кодом JavaScript и CSS, не обновляется в процессе работы с ней и не перенаправляет пользователя на другую страницу, что не исключает возможности динамической дозагрузки кода в ответ на действия пользователя. — *Примеч. ред.*

Ember.js

Ember.js — один из старейших JavaScript-фреймворков на основе MVC, появившийся в далеком 2007 году. Как говорят сами его создатели, Ember — "фреймворк, предназначенный для создания амбициозных веб-приложений" (рис. 19.3). Подобно многим другим фреймворкам, описанным в этой главе, он основан на шаблоне MVC (Model-View-Controller). Как и библиотека Backbone, он спроектирован для создания одностраничных веб-приложений.

Ember имеет репутацию непростой в изучении программной среды. Однако стоит вам разобраться с ним, как вы поймете, что он обладает массой достоинств. Ember проектировался с той целью, чтобы на первый план вышли соглашения, а не конфигурационные настройки. Для разработчиков, использующих Ember, это означает, что если код пишется в соответствии с принятой в Ember практикой программирования, то фреймворк самостоятельно сделает необходимые заключения относительно конфигурации приложения, не заставляя разработчика детально определять его конфигурацию вручную. Это позволяет экономить массу времени.

Где используется этот фреймворк? Сайты `digitalocean.com`, `vine.co`, `nbcnews.com`, `twitch.tv` и `mediabistro.com`.

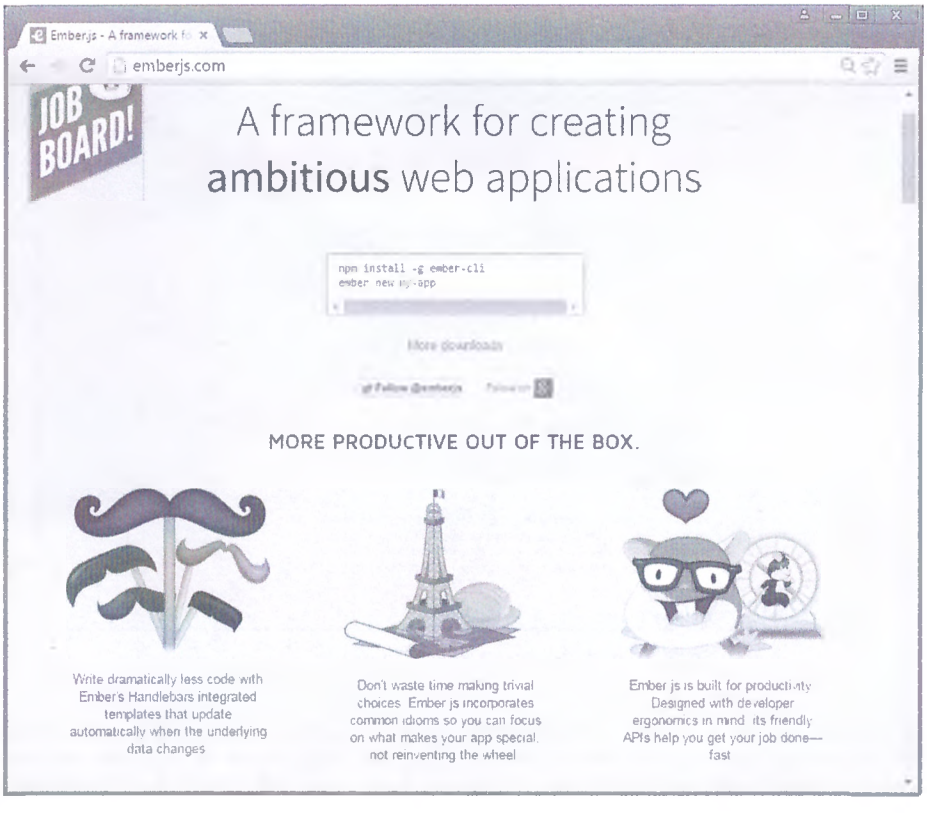


Рис. 19.3. Сайт `http://emberjs.com`

Famo.us

Famo.us (рис. 19.4) — JavaScript-фреймворк с открытым исходным кодом, предназначенный для создания сложных пользовательских интерфейсов, ориентированных на любой экран. Он имеет встроенный движок 3D-рендеринга, предоставляющий разработчикам возможность писать JavaScript-код, который может перемещать объекты в окне браузера, имитируя трехмерное движение, а также создавать эффекты и интерфейсы, которые ранее были доступны в платформозависимых приложениях. В результате веб-приложения, созданные с помощью Famo.us, могут работать намного быстрее и более плавно перемещать изображения, чем те, которые создаются только средствами HTML5, CSS3 и JavaScript.

Где используется этот фреймворк? Сайты InkaBinka.com, SuperStereo, Requested App и Japan Today.

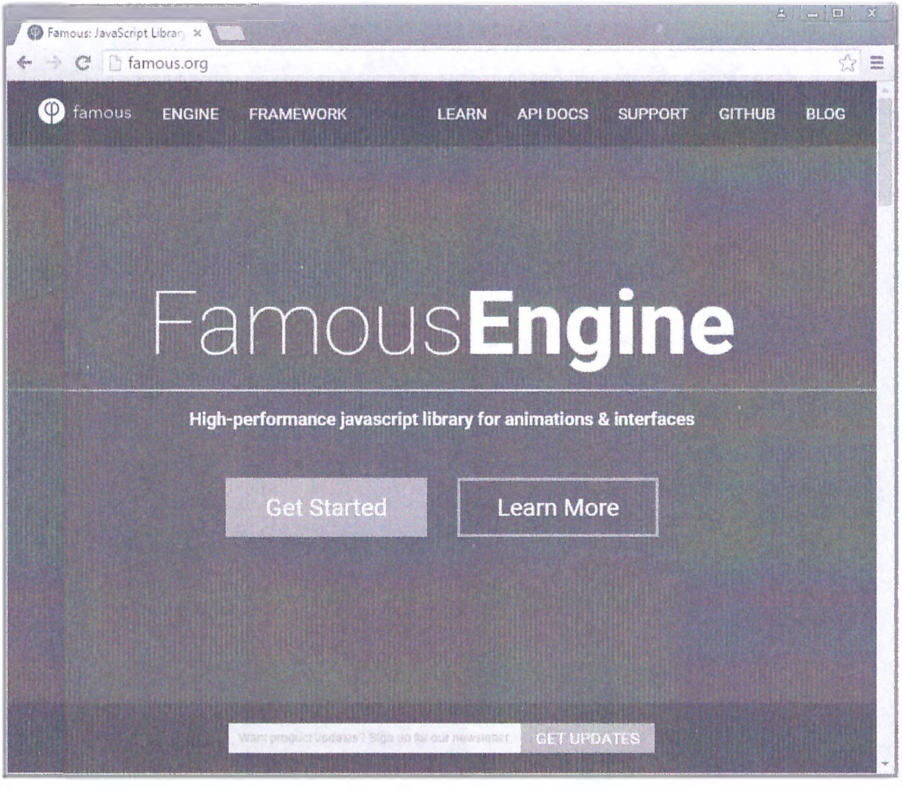


Рис. 19.4. Сайт `http://famous.org`

Knockout

Knockout (рис. 19.5) — JavaScript-фреймворк с открытым исходным кодом, предназначенный для упрощения разработки динамических пользовательских интерфейсов JavaScript. В нем используется шаблон Model-View-View-Model.

Фреймворк включает:

- ✓ декларативное связывание;
- ✓ автоматическое обновление пользовательского интерфейса при изменении данных;
- ✓ отслеживание зависимостей;
- ✓ шаблонизация.

Где используется этот фреймворк? Сайты `mlb.com`, `ancestry.com`, `Eventbrite.com` и `ameritrade.com`.

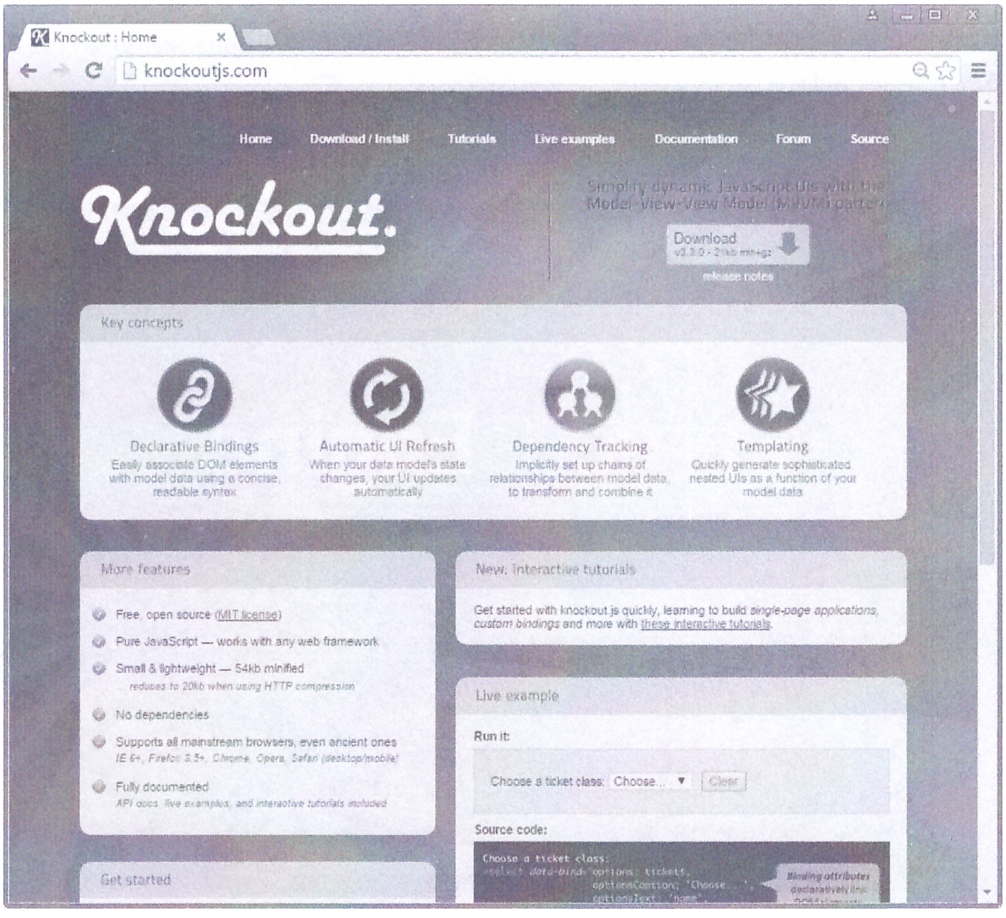


Рис. 19.5. Сайт `http://knockoutjs.com`

QUnit

QUnit (рис. 19.6) — фреймворк, позволяющий писать модульные тесты для JavaScript. Он используется во многих JavaScript-проектах с открытым кодом, включая

jQuery, и с его помощью можно тестировать любой типичный JavaScript-код. Этот фреймворк известен своими мощными возможностями и простотой использования.

Где используется этот фреймворк? Библиотека jQuery, плагины jQuery UI и jQuery Mobile, сайт `sinepoint.com` и многие другие.

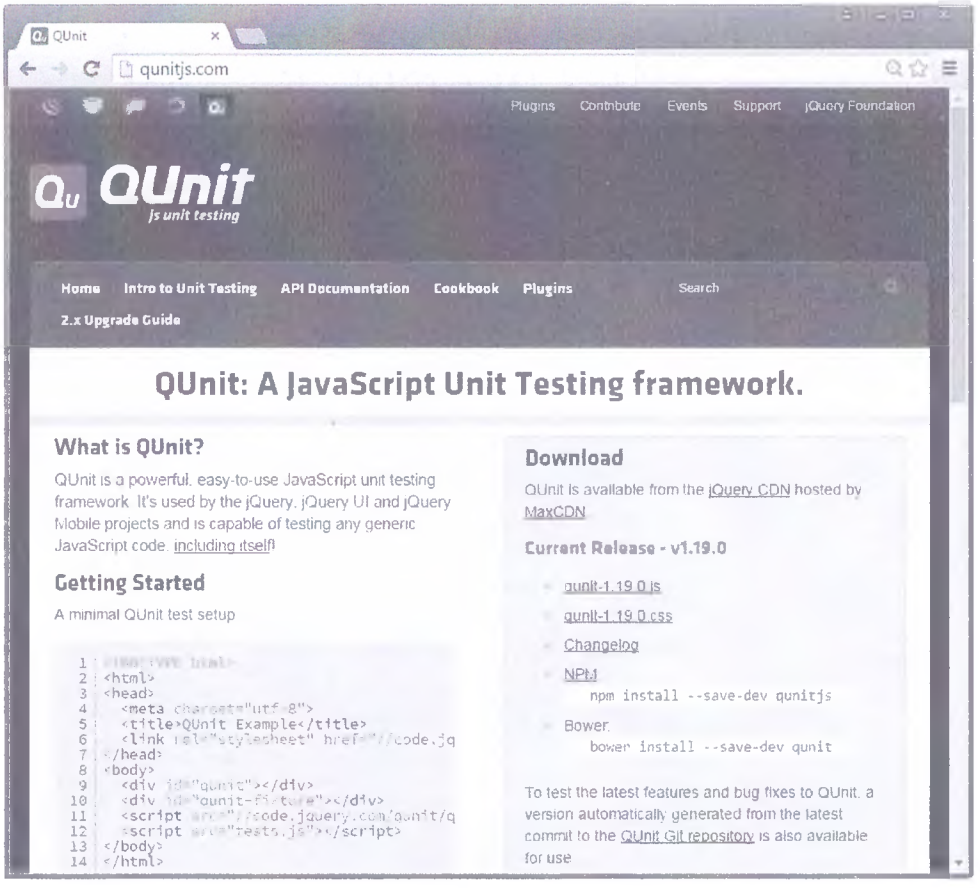


Рис. 19.6. Сайт `http://qunitjs.com`

Underscore.js

Underscore (рис. 19.7) — JavaScript-библиотека, предоставляющая программистам множество полезных вспомогательных функций. Стоит вам освоить возможности, предлагаемые библиотекой Underscore, и вы сами удивитесь тому, что ранее могли обходиться без нее.

В качестве примера вспомогательных функций Underscore можно привести функции `sortBy` (сортировка списков), `groupBy` (группирование коллекций в наборы), `contains` (возвращает `true`, если список содержит заданное значение), `shuffle` (возвращает копию списка с перемешанными элементами) и около 100 других функций, многие из которых должны были бы с самого начала быть встроенными в JavaScript.

Где используется эта библиотека? Сайты `dropbox.com`, `lifehacker.com`, `theverge.com`, `att.com` и `gawker.com`.

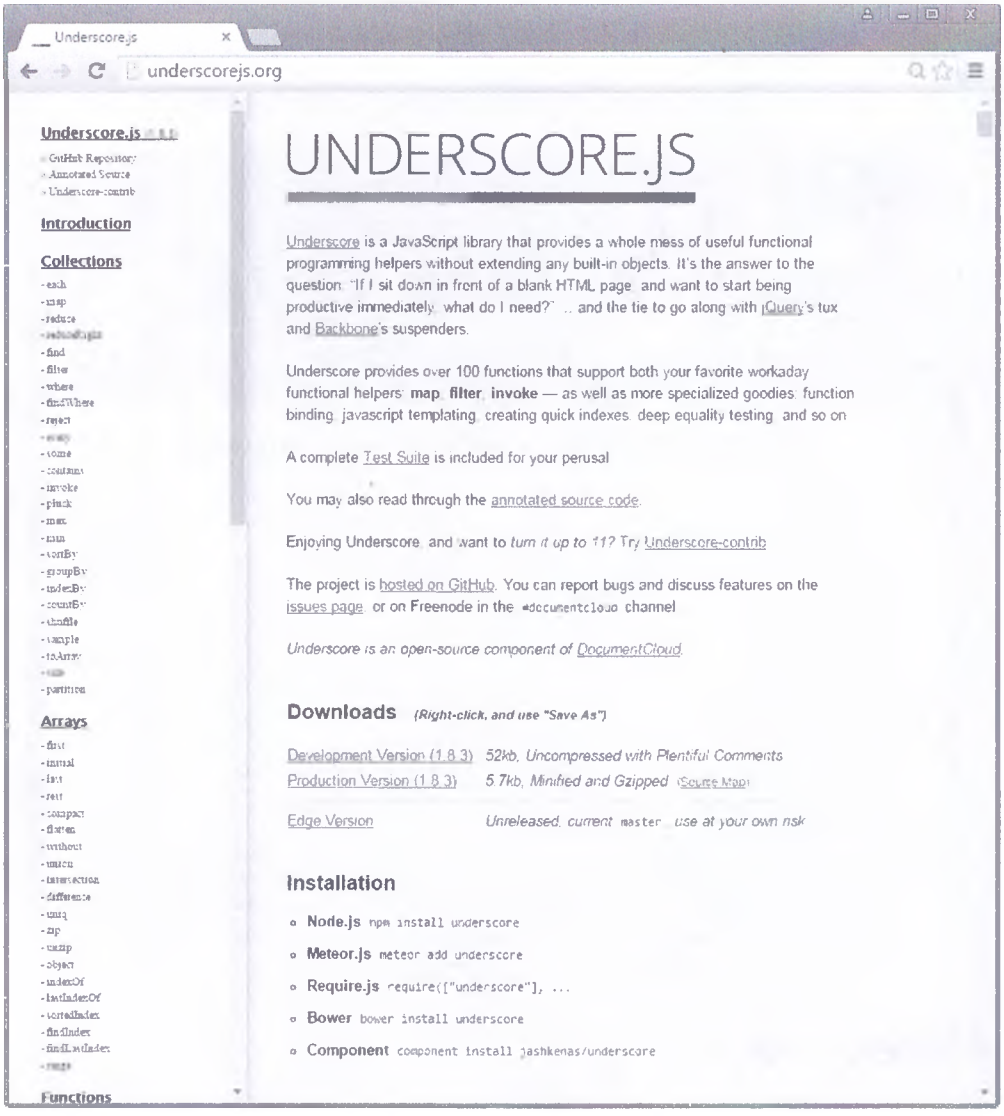


Рис. 19.7. Сайт `http://underscorejs.com`

Modernizr

Modernizr (рис. 19.8) — JavaScript-библиотека, предназначенная для выявления возможностей браузера, в котором она установлена. Чаще всего ее используют в качестве простого и удобного средства проверки того, может ли браузер пользователя выполнить какой-то конкретный фрагмент JavaScript-кода или использовать определенный

API, прежде чем пытаться задействовать это средство. Библиотеку Modernizr часто применяют в сочетании с полизаполнениями (Polyfills), которые предоставляют альтернативные способы использования новых возможностей, доступных в современных браузерах, в менее мощных устройствах и устаревших браузерах.

Где используется эта библиотека? Сайты `go.com`, `about.com`, `hostgator.com`, `addthis.com` и `usatoday.com`.

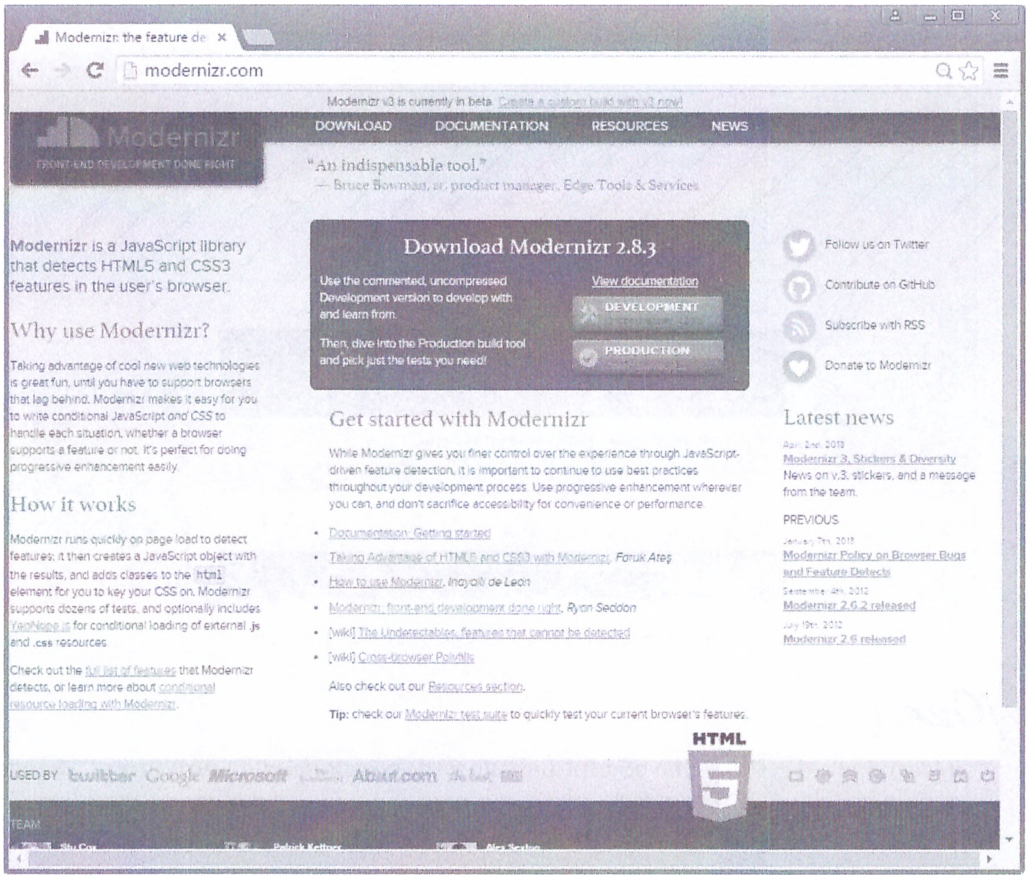


Рис. 19.8. Сайт `http://underscorejs.com`

Handlebars.js

Handlebars (рис. 19.9) — это шаблонизатор JavaScript. Он позволяет программистам вставлять шаблоны в HTML-страницы, синтаксический разбор которых будет проводиться с использованием актуальных данных, передаваемых функции Handlebars.js.

Где используется этот шаблонизатор? Сайты `meetup.com`, `mashable.com`, `flickr.com`, `wired.com` и `overstock.com`.

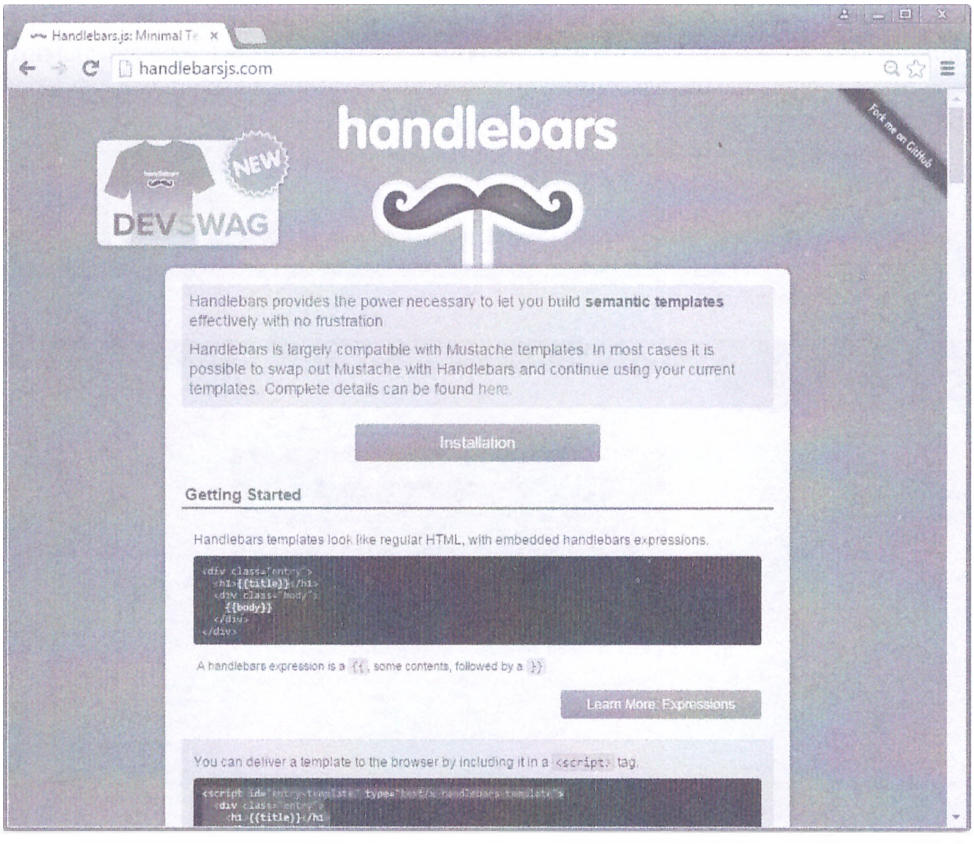


Рис. 19.9. Сайт `http://underscorejs.com`

jQuery

jQuery (рис. 19.10) — JavaScript-библиотека, созданная под лозунгом "Меньше кода, больше дела". Свыше 60% наиболее популярных сайтов используют эту библиотеку, которая стала незаменимым инструментом для большинства JavaScript-программистов. Достаточно вспомнить хотя бы о таких возможностях, предлагаемых в jQuery, как DOM-манипуляции, обработка событий, анимация и AJAX.

Помимо всего прочего, архитектура jQuery рассчитана на работу с подключаемыми модулями (плагинами), что позволяет другим разработчикам достраивать основную функциональность библиотеки для создания новых библиотек и фреймворков.

К числу наиболее популярных плагинов относятся jQuery UI, jQuery Mobile, многочисленные эффекты, средства для работы с изображениями и многие другие. Полный список доступных плагинов jQuery вы найдете по адресу `http://plugins.jquery.com`.

Где используется jQuery? Сайты WordPress.com, Pinterest, Amazon, Microsoft.com, Etsy и множество других.

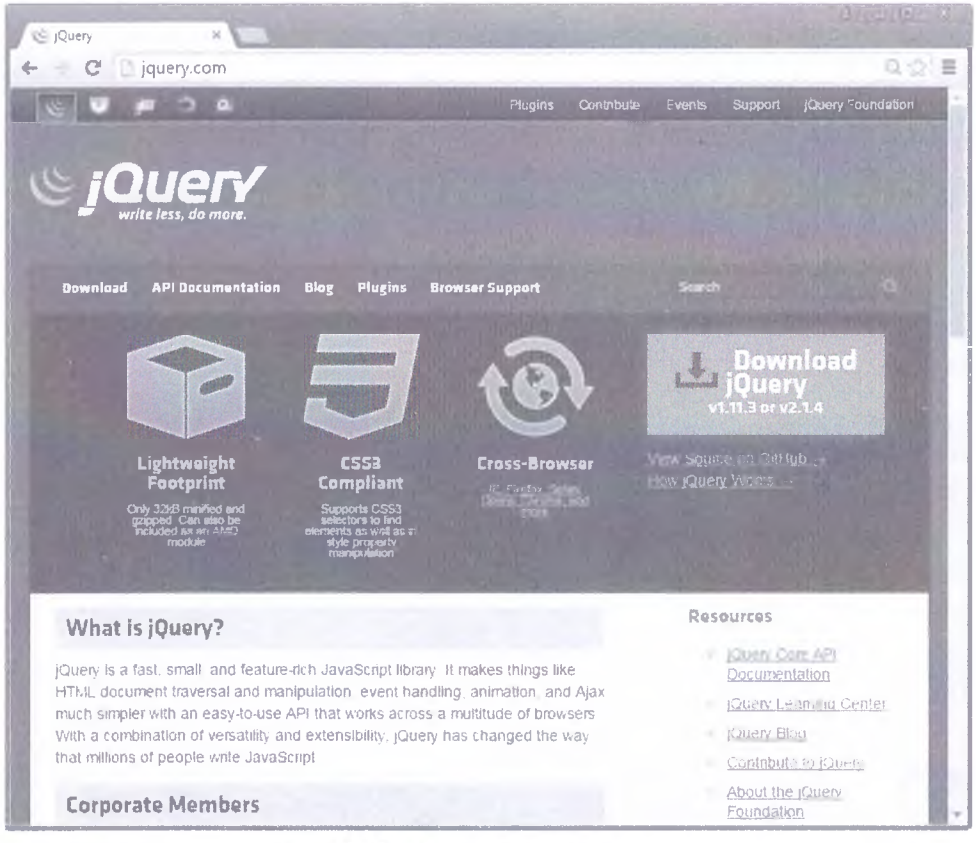


Рис. 19.10. Сайт `http://jquery.com`

Глава 20

Десять самых распространенных ошибок в JavaScript-программах и как их избежать

В этой главе...

- Непарные скобки
- Некорректные знаки препинания
- Исправление ошибок
- Неправильное использование имен переменных

"Не бойтесь совершенства — вы его не достигнете".
Сальвадор Дали

Даже самые квалифицированные JavaScript-программисты допускают ошибки. Одни ошибки приводят к тому, что программа не позволяет получать те результаты, которых от нее ожидают, тогда как другие делают невозможным само выполнение программы. Любая проблема, являющаяся причиной того, что программа не выполняется или выполняется не так, как должна, называется *дефектом* (жарг. "баг, глюк") или логической ошибкой. На протяжении всей книги мы давали вам советы относительно того, как находить и устранять возникающие в программе ошибки, и знакомили с инструментальными средствами, упрощающими эту задачу.

Умение распознавать потенциальные источники дефектов и своевременно их устранять является одной из составных частей мастерства программиста. По мере приобретения опыта вы станете замечать, что допускаете все меньше и меньше ошибок и что строки безошибочного, работоспособного кода буквально слетают с кончиков ваших пальцев. Это будет верным признаком вашего постепенного превращения в эксперта JavaScript.

В этой главе мы расскажем о десяти наиболее распространенных ошибках, допускаемых JavaScript-программистами любого уровня квалификации. Мы также дадим рекомендации относительно того, как избежать появления ошибок такого рода.

Путаница с оператором сравнения

Равны ли x и y? Имеет ли x значение true? Вопросы равенства занимают центральное место в JavaScript, но иногда с ними случается путаница. Как правило, вопросы такого рода касаются трех аспектов JavaScript: условных инструкций и операторов (if, && и др.), оператора равенства (==) и оператора строгого равенства, или тождественности (===).

Словно специально для того, чтобы еще больше усложнить нашу жизнь, оператор присваивания (=) подозрительно похож на то, что большинство из нас назвали бы знаком равенства. Однако не дайте обвести себя вокруг пальца! Приведенные ниже примеры помогут вам лучше понять, при каких обстоятельствах следует использовать каждый из этих операторов.

Избегайте неправильного использования оператора присваивания

С помощью оператора присваивания значение операнда, находящегося справа, присваивается операнду, находящемуся слева. Например:

```
var a = 3;
```

В этой инструкции новой переменной с именем a присваивается значение 3.

В программе в роли *операнда* может выступать все, что угодно. Можете считать, что роль операнда в инструкции аналогична роли существительного в обычном языке, тогда как операторы (+, -, *, / и т.д.) можно сравнить с глаголами.

Справа от операторов присваивания могут находиться выражения (иногда довольно сложные), которые предварительно вычисляются, после чего полученный результат присваивается переменной, находящейся слева.

Обычная ошибка начинающих — использование оператора присваивания вместо оператора сравнения. Например:

```
if (a = 4) {.   .}
```

Этот код не будет выполняться так, как ожидается, если в ваши намерения входит сравнить значение переменной a с числом 4.

Как избежать подводных рифов сравнений

Оператор равенства (==) и его антипод оператор неравенства (!=) могут быть довольно гибкими, но одновременно и коварными. Мы рекомендуем использовать их как можно реже или вообще отказаться от их использования. Вот пример, поясняющий это утверждение:

```
0 == '0';
```

Каждому, у кого за плечами имеется хотя бы небольшой опыт программирования, известно, что число, заключенное в кавычки, это на самом деле не число. Однако оператор == считает их эквивалентными, поскольку, прежде чем сравнивать между собой

две величины, он приводит их к одному типу. Это может порождать проблемы, установить причину которых бывает очень трудно.

Если вы хотите сравнить строку с числом и получить результат `true` при их видимом совпадении, то гораздо безопаснее выполнить явное приведение типов:

```
parseInt(0) === parseInt("0")
```

Эта инструкция также даст значение `true`, но, в отличие от предыдущей, она не включает никакой магии. Она возвращает нас к нашим добрым друзьям — операторам строгого равенства (`===`) и строгого неравенства (`!==`). Эти операторы делают именно то, что вы от них и ожидаете. Как вы думаете, к какому результату приведет вычисление следующего выражения?

```
0 === '0'
```

Поскольку операнды имеют совершенно разные типы, результат будет `false`.

Непарные скобки

По мере усложнения программы, особенно когда вы работаете с объектами JavaScript, в ней появляется все больше скобок. И тогда вы вдруг замечаете, что программа ведет себя как-то странно, или начинаете получать в консоли JavaScript загадочные сообщения об ошибках.

Вот пример объекта JavaScript, в котором имеются непарные скобки.

```
{
    "status": "OK",
    "results": [{
        "id": 12,
        "title": "Coding JavaScript For Dummies",
        "author": "Chris Minnick and Eva Holland",
        "publication_date": "",
        "summary_short": "",
        "link": {
            "type": "review",
            "url": "",
            "link_text": "Read the New York Times Review
            of Coding JavaScript For Dummies"
        },
        "awards": [{
            "type": "Nobel Prize",
            "url": "",
          }]
}
```

Видите ли вы здесь источники проблем? Для этого вам придется подсчитать количество открывающих и закрывающих скобок и проверить, согласуются ли они между собой. Если вам не удастся найти ошибку, то вы столкнетесь с серьезными проблемами. В подобной ситуации хороший редактор кода может оказаться для вас просто бесценным! Редактор Sublime Text располагает средством, которое выделяет для вас

парную скобку (по крайней мере, ту, которую редактор считает парной), когда вы подводите курсор либо к открывающей, либо к закрывающей скобке (рис. 20.1).

Рис. 20.1. Подсветка парных скобок в редакторе кода Sublime Text

Несоответствие кавычек

Для определения строк JavaScript разрешает использовать двойные или одинарные кавычки. Однако JavaScript придерживается правила, в соответствии с которым вы обязаны заканчивать строку кавычкой того же типа, которой она начинается. Кроме того, следите за правильным использованием апострофа, который передается тем же символом, что и одинарные кавычки, окружающие строку. Вот соответствующие примеры.

```
var movieName = "Popeye'; // ошибка!
var welcomeMessage = 'Thank you,' + firstName + ', let's
                     learn JavaScript!' // ошибка!
```

Отсутствующие скобки

Эта ошибка чаще всего вкрадывается в условные инструкции, особенно те, которые объединяют несколько условий. Рассмотрим следующий пример/

```
if (x > y) && (y < 1000) {
...
}
```

Нам нужно проверить, что каждое из этих условий истинно. Фактически мы имеем здесь три условия, и каждое из них нуждается в скобках. Чего тут не хватает, так это скобок вокруг длинного условия `&&`, которое гласит, что код в фигурных скобках сможет выполниться лишь в тех случаях, когда истинными будут оба других условия.

Вот правильная запись этой инструкции.

```
if ((x > y) && (y < 1000)) {
...
}
```

Отсутствие точки с запятой

Инструкции JavaScript всегда должны заканчиваться точкой с запятой. Но если вы поместите каждую инструкцию в собственной строке, не ставя точку с запятой в конце, то код все равно выполнится, как если бы точка с запятой была на месте. Несмотря на то что код будет выполняться, пропуск точки с запятой может породить проблемы, если впоследствии вы реорганизуете код или если две инструкции каким-либо образом окажутся в одной строке.

Лучший способ избежать этой ошибки — это всегда ставить точку с запятой в конце инструкции.

Ошибки, связанные с неправильным использованием регистра букв

Язык JavaScript чувствителен к регистру. Это означает, что написание имен переменных во всей программе должно быть в точности таким, как при их создании. То же самое относится и к функциям (включая встроенные функции JavaScript).

Одна из наиболее часто встречающихся ситуаций, в которых допускают ошибку такого рода, связана с использованием метода `getElementById()` объекта `Document`. Многие записывают этот вызов как `getElementByID()`, что неправильно!

Ссылки на код, не успевший загрузиться

Обычно код JavaScript загружается и выполняется в том порядке, в каком он появляется в документе (разумеется, это не относится к определениям функций). Это может создать проблемы, если вы будете ссылаться на HTML-элементы, объявляемые в документе, из сценария, находящегося в заголовке. В листинге 20.1 приведен пример сценария, в котором делается попытка изменить текст в элементе, тогда как на рис. 20.2 показано, что выполнение этого сценария не приводит к желаемому результату.

Листинг 20.1. Избегайте использования ссылок на код или разметку до того, как они загрузятся

```
<html>
<head>
  <script>
  document.getElementById("myDiv").innerHTML =
        "Это МОЙ элемент div";
  </script>
</head>
```

```
<body>
  <div id = "myDiv">Это ваш элемент div</div>

</body>
</html>
```

Этот код работает не так, как ожидается, поскольку во время выполнения JavaScript-сценария браузеру еще ничего не известно об элементе `div` с идентификатором `id="myDiv"`, который появится на странице позже.

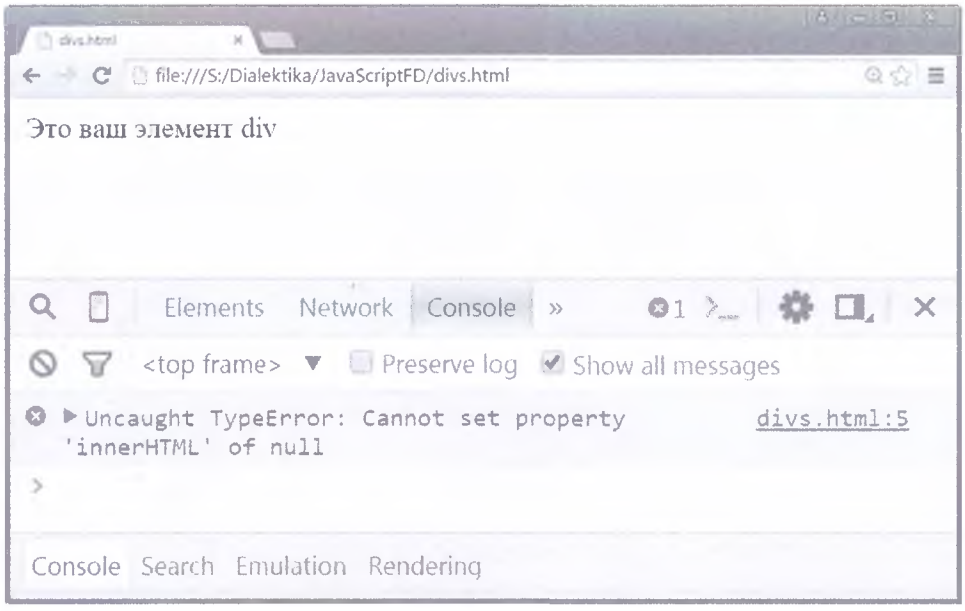


Рис. 20.2. Ссылка на HTML-элемент до его загрузки приводит к ошибке

Вот две рекомендации, которые помогут вам избежать возникновения подобных проблем.

- ✓ Поместите свой JavaScript-код в конце документа, непосредственно перед конечным тегом `</body>`.
- ✓ Поместите свой JavaScript-код в функцию. Вы сможете вызвать эту функцию, используя атрибут события `onload` в начальном теге `<body>`.

Проблема, продемонстрированная в листинге 20.1, разрешена в листинге 20.2 с помощью второго метода. Результат выполнения этого сценария в браузере показан на рис. 20.3.

Листинг 20.2. Выполнение сценария после завершения загрузки страницы

```
<html>
<head>
  <script>
  function nameMyDiv() {
    document.getElementById("myDiv").innerHTML =
       "Это МОЙ элемент div";
  }
  </script>
</head>
<body onload = "nameMyDiv();">
  <div id = "myDiv">Это ваш элемент div</div>
</body>
</html>
```

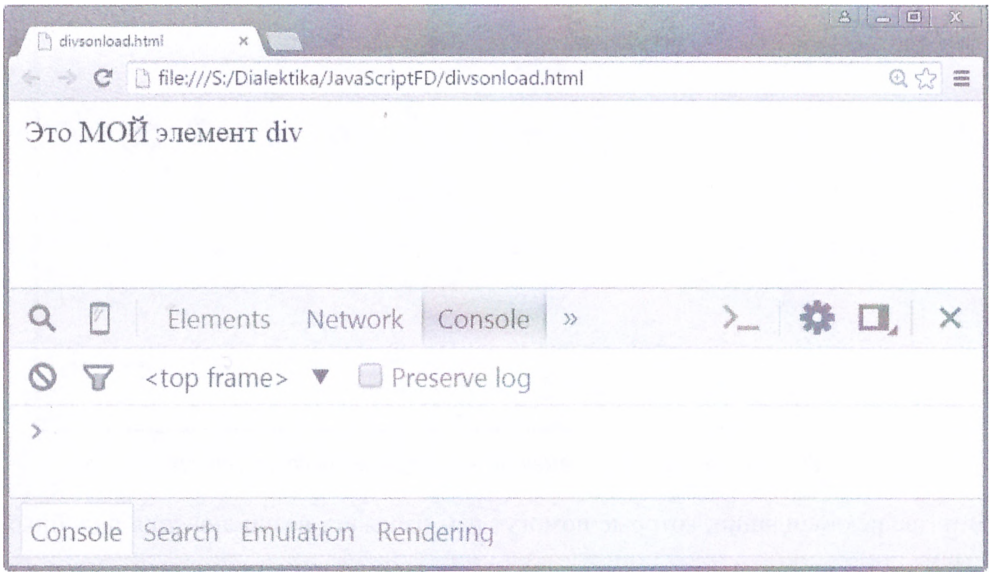


Рис. 20.3. Выполнение сценария после завершения загрузки страницы

Плохие имена переменных

Правила именования переменных подробно рассмотрены в главе 3. Одно из правил, нарушение которых отслеживается с трудом, касается запрета использования зарезервированных слов в качестве имен переменных.

Отметим, что в JavaScript имеется свыше 60 зарезервированных слов, а также много других, которые не могут служить именами переменных. Наилучший способ избежать ошибок именования переменных состоит не в том, чтобы пытаться запомнить все зарезервированные слова, а в том, чтобы использовать какую-либо описательную схему присвоения имен, которая сделает крайне маловероятной вероятность совпадения имен ваших переменных с зарезервированными словами.

Например, слово `name` является одним из зарезервированных слов JavaScript. Если вы выработаете привычку быть более конкретным при выборе имен переменных и использовать, например, такие имена, как `firstName`, `lastName`, `dogName` или `nameOfTheWind`, то сможете полностью исключить конфликты с зарезервированными словами.

Ошибки, связанные с неправильным использованием областей видимости переменных

В JavaScript имеются области видимости функций и глобальная область видимости. Если переменная объявлена без использования ключевого слова `var`, то она становится глобальной переменной и ее можно использовать в любом месте программы. Как было продемонстрировано в главе 3, использование глобальных переменных чревато нарушением нормальной работы программы. Чтобы избежать ошибок, связанных с неправильным использованием областей видимости, всегда создавайте новые переменные с использованием ключевого слова `var`.

Пропуск параметров при вызове функций

Всякий раз, когда вы создаете функцию, вы указываете в ее объявлении количество параметров, которые должны передаваться ей при вызове. Вызов функций с недостающим количеством параметров не всегда приводит к ошибкам в JavaScript, но может быть причиной получения неожиданных результатов, если коду функции требуются параметры, значения которых не определены.

Создавая функции, обязательно присваивайте параметрам описательные имена и каждый раз дважды проверяйте, что количество аргументов, передаваемых функции при ее вызове, соответствует требуемому.

Подсчитываем ошибки: забывчивость в отношении отсчета индексов от нуля

Если, работая с массивом, вы обращаетесь к элементу с индексом `10`, то фактически обращаетесь к одиннадцатому по счету элементу массива (рис. 20.4). Никогда не забывайте о том, что первому элементу массива соответствует индекс 0.

```
var myArray = new Array();
myArray[10] = "Список 10 наиболее распространенных ошибок";
myArray.length; // выдает 11!
```

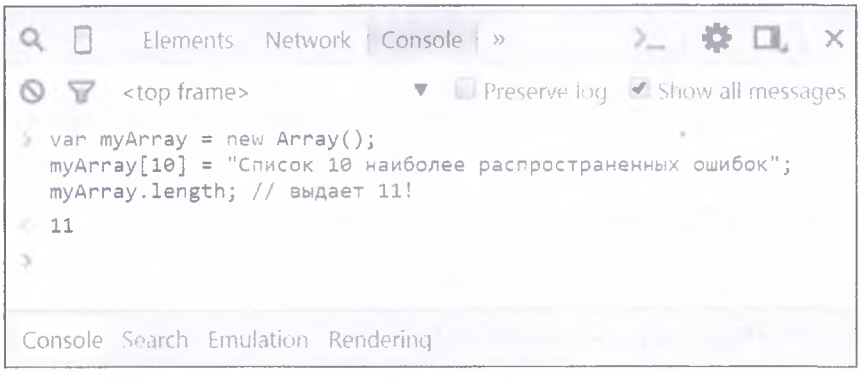


Рис. 20.4. Выполнение сценария после завершения загрузки страницы

Глава 21

Десять онлайн-инструментов, которые улучшат качество создаваемых вами программ на JavaScript

В этой главе...

- Очистка кода с помощью JSLint
- Знакомимся с JSFiddle
- Улучшение внешнего вида кода с помощью JSbeautyfier
- Уменьшение размеров JavaScript-файлов

"Не стоит недооценивать возможности простых инструментов".
Крейг Брюс

Для JavaScript создано больше библиотек, ресурсов и полезных инструментов, чем для любого другого языка программирования. В этой главе представлены десять лучших онлайн-ресурсов, которые помогут вам создавать более эффективные программы на JavaScript.

JSLint

Программа JSLint, созданная супергениальным JavaScript-программистом Дугласом Крокфордом, — это инструмент оптимизации качества кода, проверяющий соблюдение программистом стандартов кодирования и указывающий те места в коде, где эти стандарты нарушены.

JSLint (рис. 21.1) позволяет находить в коде огрехи, на которые тысячи программистов обычно не обращают внимания, за что впоследствии расплачиваются многочасовыми поисками причин того, почему их программа работает не так, как ожидается. Если ваш код успешно прошел тесты JSLint, то он, скорее всего, может считаться довольно хорошим.

Рис. 21.1. JSLint укажет на проблемные места в вашем коде

JSFiddle.net

JSFiddle (рис. 21.2) — это онлайновая программа для выполнения веб-приложений в тестовой среде. Когда вы перейдете на сайт JSFiddle.net, первое, что вы увидите, — это четыре окна, представляющих собой следующие панели:

- ✓ HTML;
- ✓ CSS;
- ✓ JavaScript;
- ✓ результаты.

Введите в любом из трех первых окон код соответствующего типа и щелкните на кнопке Run (Выполнить). Результат выполнения отобразится в панели результатов.

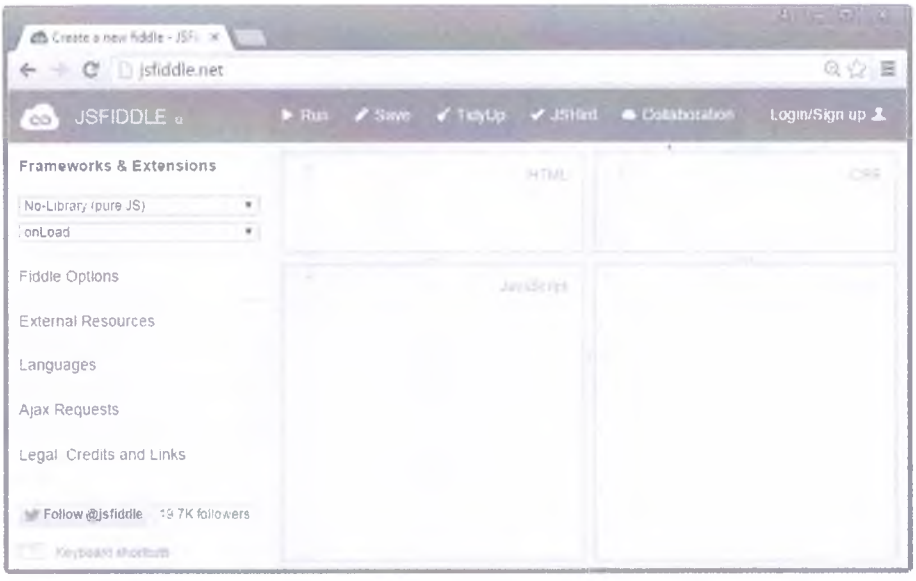


Рис. 21.2. JSFiddle — это целая игровая площадка для JavaScript

Работая с JSFiddle, вы имеете возможность сохранять содержимое панелей и отправлять соответствующие URL-адреса по электронной почте другим людям, чтобы они могли с ним ознакомиться.

JSBin

JSBin (рис. 21.3) — это онлайн-сервис, обеспечивающий разделение кода с другими пользователями, которые могут наблюдать за тем, как вы создаете программу. Независимо от того, хотите ли вы, чтобы за вашим творческим процессом наблюдали другие люди, собираетесь обучать менее квалифицированных программистов или работаете над проектом совместно с другими разработчиками, функциональные возможности JSBin очень помогут вам в отладке программ, а также в организации обратной связи и коллективного доступа к коду.

javascriptcompressor.com

Чем меньше размер ваших JavaScript-файлов, тем быстрее они загружаются. Сайт `javascriptcompressor.com` (рис. 21.4) предоставляет окно, куда можно вставить свой предварительно скопированный JavaScript-код. После того как вы щелкнете на кнопке Compress, в нижнем окне появится новая версия первоначального кода, эквивалентная ему с точки зрения функциональности, но сжатая. Сайт не только сжимает код, но и *обфусцирует* его, т.е. приводит его к виду, затрудняющему его анализ и декомпиляцию, тем самым скрывая ваши секреты от любопытных глаз.

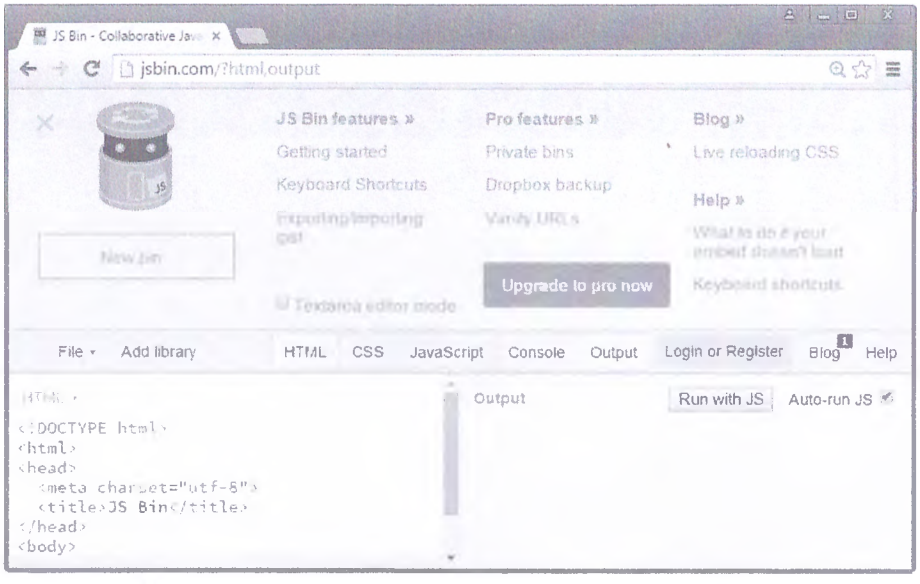


Рис. 21.3. Организация коллективной работы с помощью JSBin

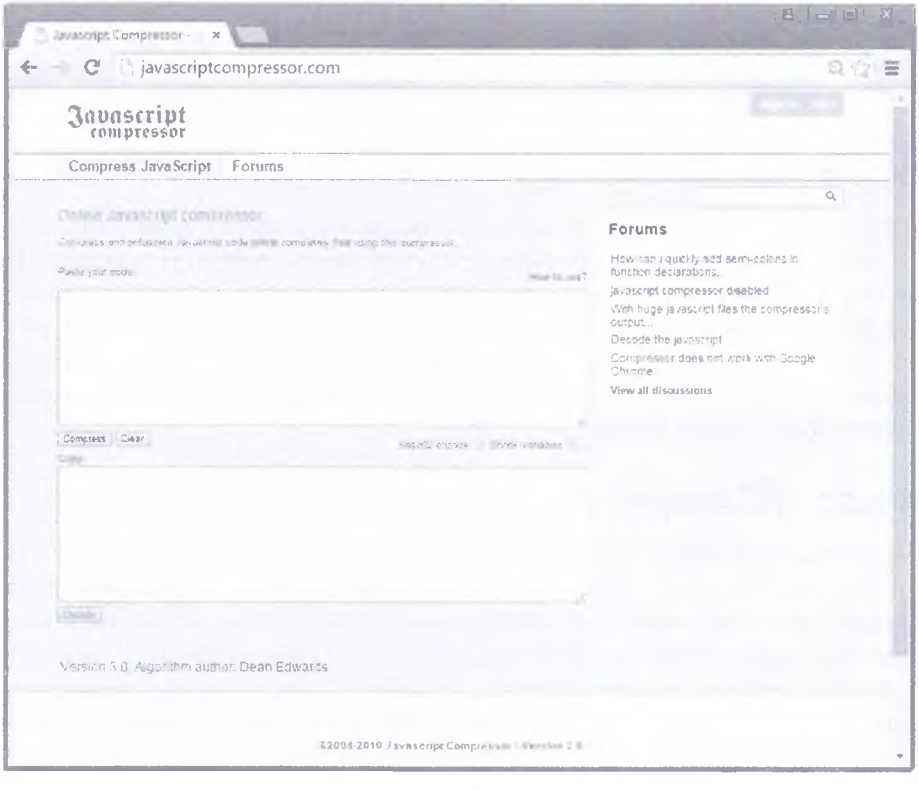


Рис. 21.4. Сервис `javascriptcompressor.com` уменьшает размеры файлов

jsbeautifier.org

JSBeautifier (рис. 21.5) — онлайновый инструмент, который приводит в порядок любой неряшливо оформленный код. Форматирование кода включает в себя следующие возможности:

- ✓ вставка новых строк;
- ✓ разрыв строк связанного кода;
- ✓ вставка пробелов перед условными операторами;
- ✓ применение стандартной схемы создания отступов по всему сценарию.

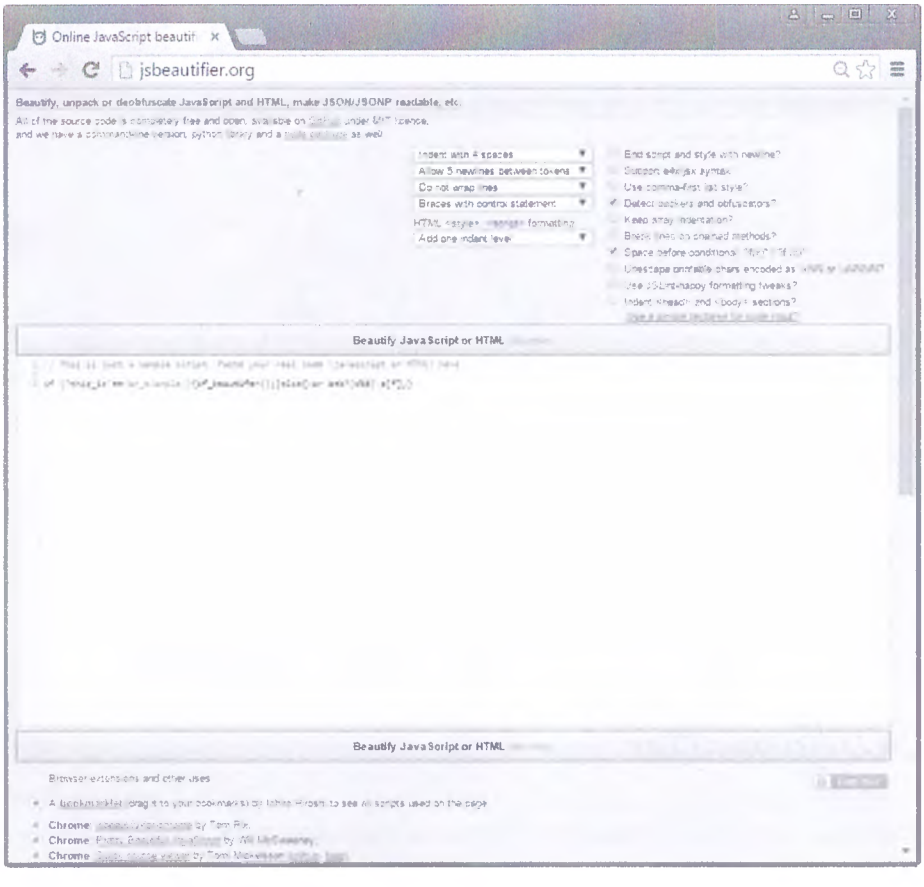


Рис. 21.5. Форматирование кода на сайте `http://jsbeautifier.org`

Генератор регулярных выражений JavaScript RegEx

Созданный компанией JavaScript Lab генератор регулярных выражений JavaScript Regex (`www.jslab.dk/tools.regex.php`) — это удобная форма, позволяющая строить регулярные выражения (рис. 21.6). Вам достаточно пощелкать на определенных кнопках, ввести текст для поиска совпадений, задать некоторые опции, и в нижней части окна отобразится требуемое регулярное выражение.

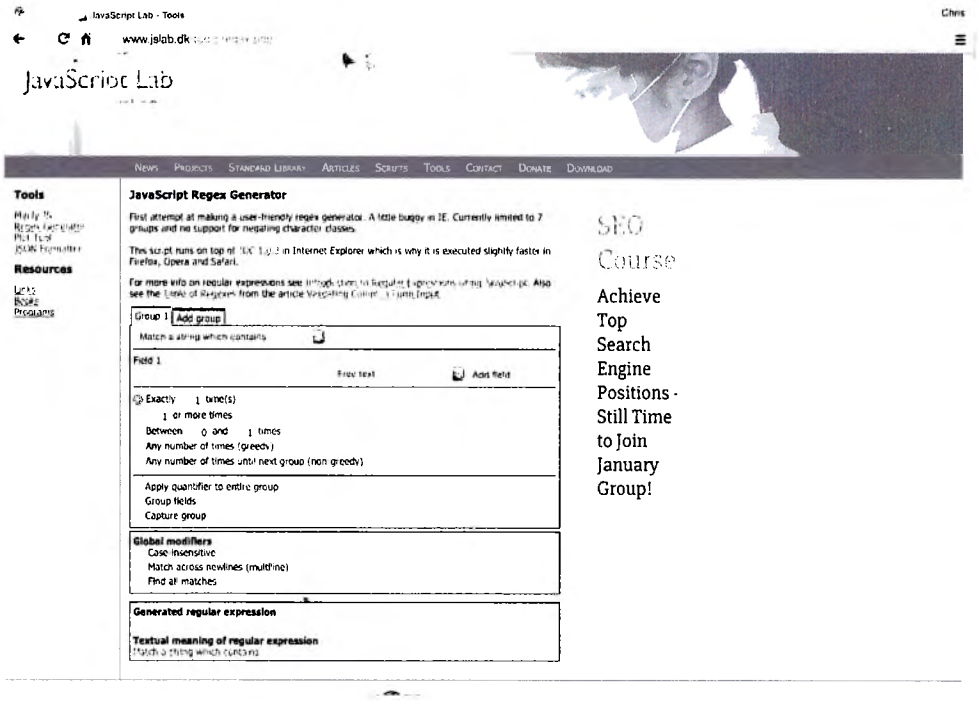


Рис. 21.6. Построение регулярных выражений одним щелчком

JSONformatter

JSONformatter (`http://jsonformatter.curiousconcept.com`) предоставляет окно для вставки неформатированного JSON-кода, например кода, полученного путем копирования из окна инструментального средства Chrome Developer Tools, форматирует код и выполняет проверку его допустимости (рис. 21.7).

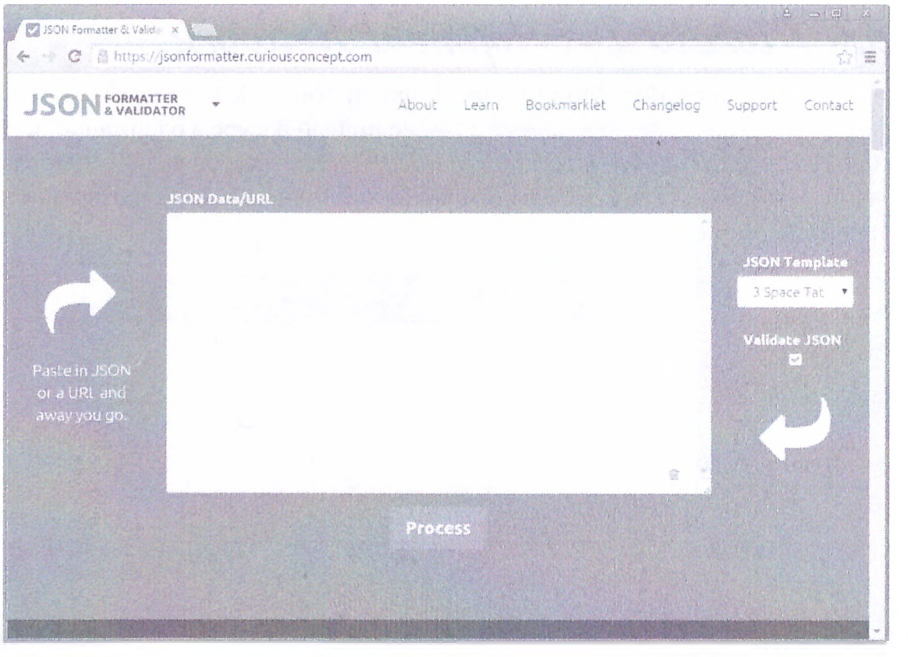


Рис. 21.7. Форматировщик JSON проверяет и упорядочивает JSON-данные

jshint.com

JShint (рис. 21.8) — это инструмент, облегчающий обнаружение ошибок и источников потенциальных проблем в JavaScript-коде. Кроме того, он предоставляет полезную информацию о коде JavaScript в процессе его написания.

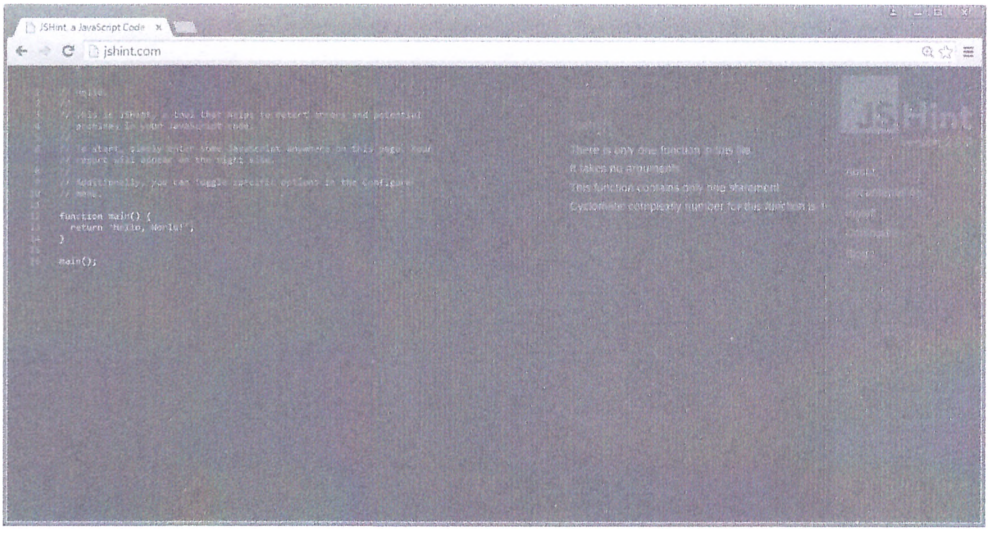


Рис. 21.8. Обнаружение проблем в коде по мере его написания с помощью средства JShint

Сайт Mozilla Development Network

Раздел JavaScript сайта Mozilla Development Network (`https://developer.mozilla.org/ru/docs/Web/JavaScript`) — солидный ресурс, содержащий информацию обо всем, что связано с JavaScript (рис. 21.09). Этот ресурс включает справочные материалы, учебники, статьи и демонстрационные примеры для программистов любого уровня.

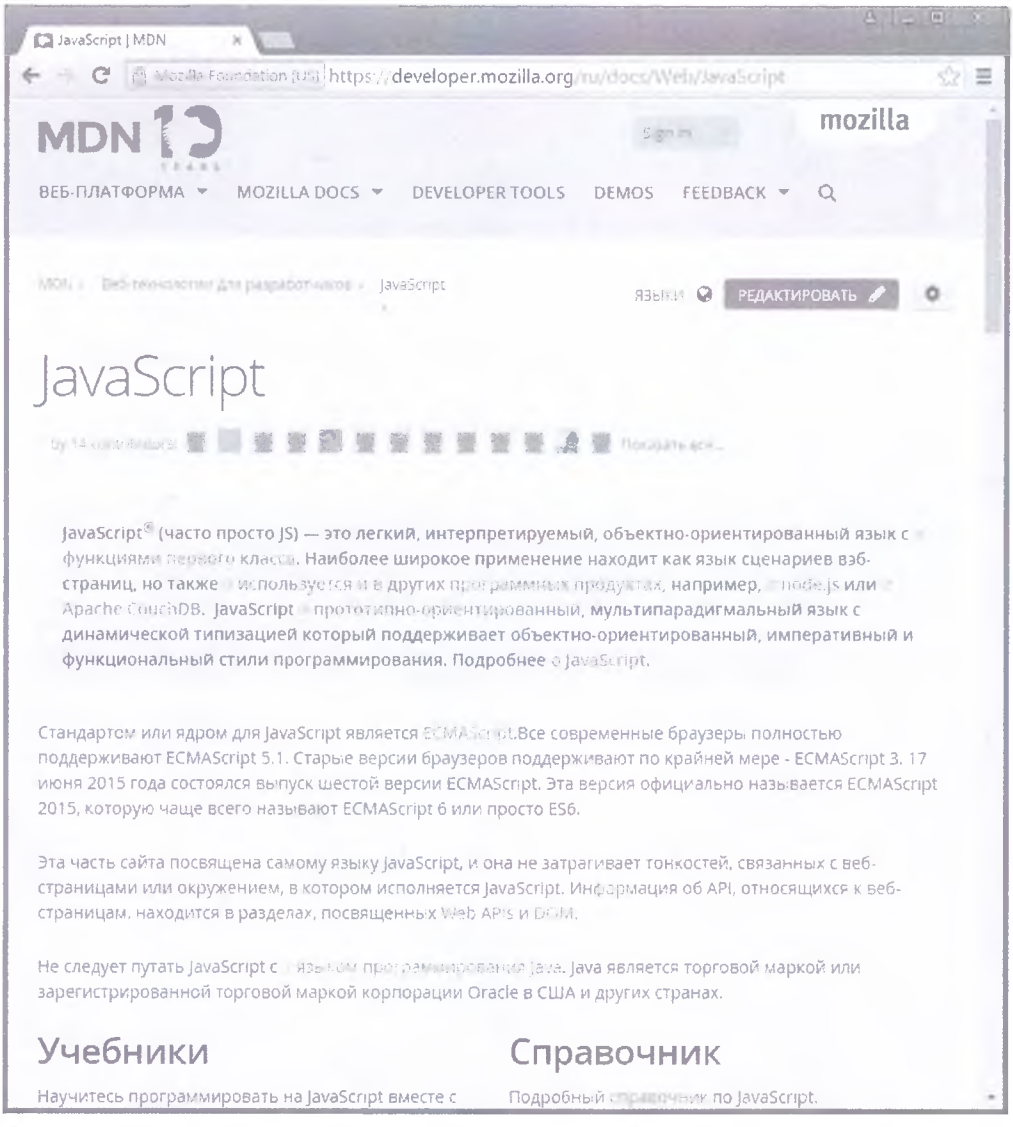


Рис. 21.9. Сайт Mozilla Development Network — один из лучших информационных ресурсов по JavaScript

Дуглас Крокфорд

Дуглас Крокфорд — герой для многих JavaScript-программистов. На его сайте (`http://javascript.crockford.com`) в свободном доступе предоставляется коллекция видеоклипов, охватывающих все аспекты JavaScript (рис. 21.10). Этот видеоматериал представляет огромную ценность для тех программистов, которые уже прошли начальный этап обучения и хотели бы повысить свой профессиональный уровень.

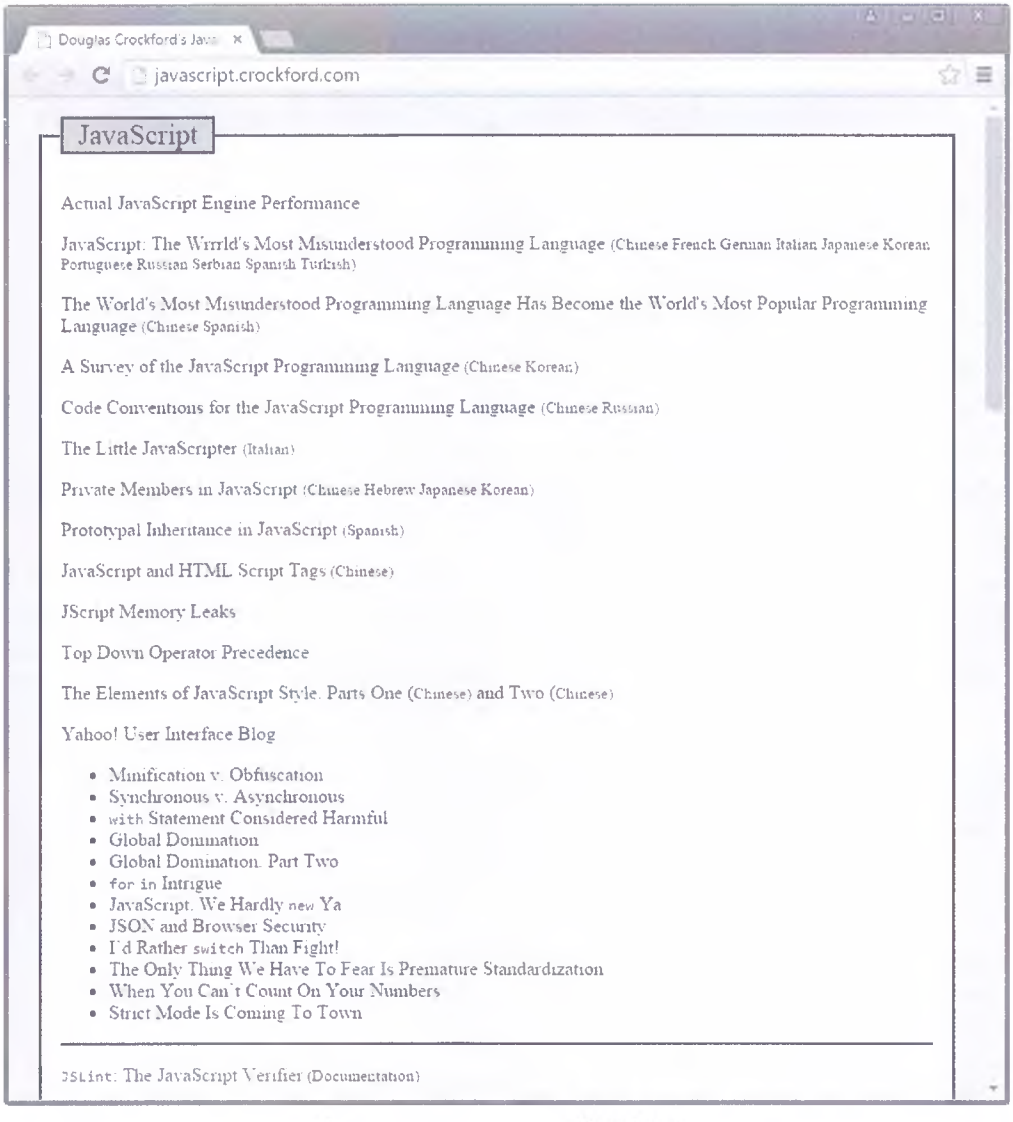


Рис. 21.10. Сайт Дугласа Крокфорда

Предметный указатель

П

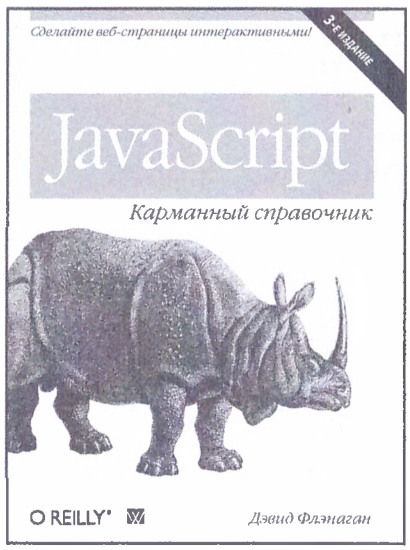
Р

С

Т

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

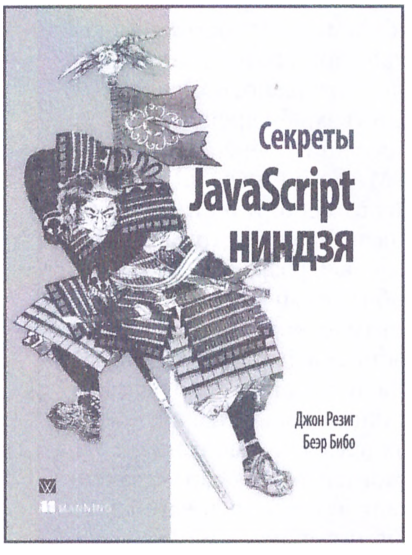
В главах 10–14 рассматриваются функциональные возможности языка наряду с моделью DOM и средствами поддержки HTML5.

ISBN 978-5-8459-1830-7

в продаже

СЕКРЕТЫ JAVASCRIPT НИНДЗЯ

ДЖОН РЕЗИГ
БЕЭР БИБО



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.

www.williamspublishing.com

ISBN 978-5-8459-1959-5 **в продаже**

JavaScript для профессионалов

Второе издание

Джон Резиг,
Расс Фергюсон,
Джон Пакстон

www.williamspublishing.com

Эта книга является незаменимым пособием для профессиональных разработчиков современных веб-приложений на JavaScript. Читатель найдет в ней все, что требуется знать о современном состоянии языка JavaScript, его достоинствах и недостатках, новых языковых средствах, внедренных в последних версиях стандарта ECMAScript, передовых приемах отладки и тестирования кода, а также инструментальных средствах разработки. Книга изобилует многочисленными практическими и подробно разбираемыми примерами кода, повторно используемых функций и классов, экономящих время разработчиков. Она помогает им овладеть практическими навыками написания динамических веб-приложений на высоком профессиональном уровне, а также повысить свою квалификацию.

Книга рассчитана на тех, кто интересуется разработкой веб-приложений и имеет опыт программирования на JavaScript.

ISBN 978-5-8459-2054-6 **в продаже**

JavaScript® для чайников®

Шпаргалка

Список селекторов jQuery, используемых в JavaScript-программах

Селектор	Выбираемые элементы
*	Все элементы
:animated	Все элементы выбранного набора, которые анимируются в данный момент
[*атрибут*|="*значение*"]	Элементы, имеющие указанный атрибут, значение которого равно заданному значению или начинается с заданного значения, за которым следует дефис (-)
[*атрибут**="*значение*"]	Элементы, имеющие указанный атрибут, значение которого содержит заданное значение
[*атрибут*~="*значение*"]	Элементы, имеющие указанный атрибут, значение которого содержит заданное значение, отделенное пробелами
[*атрибут*$="*значение*"]	Элементы, имеющие указанный атрибут, значение которого заканчивается заданным значением. Сравнение строк осуществляется с учетом регистра символов
[*атрибут*="*значение*"]	Элементы, имеющие указанный атрибут, значение которого в точности совпадает с заданным значением
[*атрибут*!="*значение*"]	Элементы, не имеющие указанного атрибута, либо имеющие его, но со значением, не совпадающим с заданным
[*атрибут*^="*значение*"]	Элементы, имеющие указанный атрибут, значение которого начинается с заданного значения
:button	Все элементы-кнопки
:checkbox	Все элементы-флажки
:checked	Все отмеченные элементы-флажки и переключатели
("P > C")	Все элементы *C*, являющиеся дочерними по отношению к элементам *P*
(".*класс*")	Все элементы заданного класса
:contains()	Все элементы, содержащие указанный текст
("A D")	Все элементы *D*, являющиеся потомками элементов *A*
:disabled	Элементы, находящиеся в неактивном состоянии
("*элемент*")	Элементы с указанным именем тега
:empty	Элементы, не имеющие дочерних элементов (включая текстовые узлы)
:enabled	Элементы, находящиеся в активном состоянии
:eq()	Элемент с указанным индексом в выбранном наборе (индексы отсчитываются от нуля)
:even	Четные элементы (индексы отсчитываются от нуля)
:file	Элементы файлового типа
:first-child	Элементы, являющиеся первыми дочерними элементами своих родителей
:first-of-type	Элементы, являющиеся первыми среди сестринских элементов с одним и тем же именем элемента
:first	Первый из подходящих элементов
:focus	Элемент, который в данное время получил фокус ввода
:gt()	Элементы выбранного набора, индексы которых превышают заданный индекс
[*атрибут*]	Элементы, имеющие заданный атрибут; значение атрибута несущественно
:has()	Элементы, содержащие по крайней мере один элемент, соответствующий указанному селектору

BESTSELLING
BOOK
SERIES

JavaScript® для чайников®

СЕРИЯ
КОМПЬЮТЕРНЫХ
КНИГ ОТ
ДИАЛЕКТИКИ

Шпаргалка

Селектор	**Выбираемые элементы**
:header	Элементы, являющиеся заголовками: h1, h2, h3 и т.д.
:hidden	Скрытые элементы
#id	Единственный элемент с указанным значением идентификатора (атрибута id)
:image	Элементы изображений
:input	Элементы input, textarea, select и button
:lang()	Элементы, для которых определен атрибут lang с заданным значением
:last-child	Элементы, являющиеся последними дочерними элементами своих родителей
:last-of-type	Элементы, являющиеся последними среди сестринских элементов с одним и тем же именем элемента
:last	Последний из подходящих элементов
:lt()	Элементы выбранного набора, индексы которых меньше заданного индекса
[*атрибут1*="*значение1*"] [*атрибут2*="*значение2*"]	Элементы, соответствующие всем заданным фильтрам атрибутов
(*селектор1*, *селектор2*... *селекторN*)	Объединенный результат применения всех указанных селекторов
(*предыдущий* + *следующий*)	Все следующие элементы указанного типа, которым предшествует элемент указанного типа
(*предыдущий* ~ *сестринский*)	Все сестринские элементы, имеющие общего родителя, которым предшествует элемент указанного типа и которые соответствуют фильтру *сестринский*
:not()	Элементы, не соответствующие заданному селектору
:nth-child()	Элементы, являющиеся n-ми дочерними элементами своих родителей
:nth-last-child()	Элементы, являющиеся *n*-ми, если считать с конца, дочерними элементами своих родителей
:nth-of-type()	Элементы, являющиеся *n*-ми среди сестринских элементов с одним и тем же именем элемента
:nth-last-of-type()	Элементы, являющиеся последними среди сестринских элементов с одним и тем же именем элемента
:odd	Нечетные элементы (индексы отсчитываются от нуля)
:only-child	Элементы, являющиеся единственными дочерними элементами своих родителей
:only-of-type	Элементы, не имеющие сестринских элементов с тем же именем элемента
:parent	Элементы, имеющие по крайней мере один дочерний узел (узел элемента или текста)
:password	Элементы, являющиеся паролями
:radio	Элементы-переключатели
:reset	Элементы типа reset
:root	Корневой элемент документа
:selected	Все выделенные элементы
:submit	Элементы типа submit
:target	Целевой элемент, указанный идентификатором фрагмента в URL-адресе документа
:text	Текстовые элементы
:visible	Видимые элементы

Шпаргалка

Список библиотек HTML5 API, используемых в JavaScript-программах

API	Описание
Ambient Light API	Предоставляет информацию об уровнях внешней освещенности, регистрируемых светочувствительным датчиком устройства
Battery Status API	Предоставляет информацию об уровне заряда батареи
Canvas 2D Context	Предоставляет средства рисования и манипулирования графикой в браузере
Clipboard API	Предоставляет доступ к функциональности операционной системы, обеспечивающей копирование, вырезание и вставку объектов
Contacts	Обеспечивает доступ к хранилищу информации о контактах пользователя
Drag and Drop	Поддерживает перетаскивание объектов в пределах окон браузера и между ними
File API	Предоставляет безопасный доступ к файловой системе устройства
Forms	Предоставляет программам доступ к новым типам данным, определенным в HTML5
Fullscreen API	Предоставляет возможность отображения веб-страницы в полноэкранном режиме без привлечения пользовательского интерфейса браузера
Gamepad API	Поддерживает ввод с USB-контроллеров игрового планшета (геймпада)
Geolocation	Предоставляет веб-приложениям доступ к данным географического местоположения пользовательского устройства
getUserMedia/Stream API	Предоставляет доступ к данным, поступающим от внешних устройств (например, веб-камеры)
History API	Позволяет программам манипулировать данными из истории просмотра браузера
HTML Microdata	Предоставляет возможность аннотировать контент машинными тегами
Indexed database	Создает простую клиентскую систему для управления базой данных в браузере
Internationalization API	Предоставляет средства форматирования и сравнения строк с учетом региональных особенностей
Offline apps	Позволяет программистам делать веб-приложения доступными в автономном режиме
Proximity API	Предоставляет информацию о расстоянии между устройством и объектом
Screen Orientation	Считывает состояние ориентации экрана (книжная или альбомная), позволяя программисту определить момент ее изменения и зафиксировать это состояние
Selection	Поддерживает выбор элементов в JavaScript с использованием селекторов CSS
Server-sent events	Позволяет серверу отсылать данные браузеру без предварительного получения запроса
User Timing API	Предоставляет программистам доступ к высокоточным временным отметкам для измерения производительности приложений
Vibration API	Позволяет управлять вибрацией устройства
Web Audio API	Программный интерфейс для обработки и синтеза звука

Шпаргалка

API	*Описание*
Web Messaging	Позволяет окнам браузера взаимодействовать через различные источники
Web Speech API	Предоставляет средства речевого ввода и преобразования текста в речь
Web storage	Позволяет сохранять пары "имя–значение" в браузере
Web sockets	Открывает сеанс двухсторонней связи между браузером и сервером
Web Workers	Позволяет выполнять сценарии JavaScript в фоновом режиме
XMLHttpRequest2	Улучшает технологию XMLHttpRequest, устраняя необходимость в обходе ограничений политики одного источника и позволяя работать с новыми средствами HTML5

Список зарезервированных слов JavaScript

abstract	final	public
boolean	finally	return
break	float	short
byte	for	static
case	function	super
catch	goto	switch
char	if	synchronized
class	implements	this
const	import	throw
continue	in	throws
debugger	instanceof	transient
default	int	true
delete	interface	try
do	long	typeof
double	native	var
else	new	void
enum	null	volatile
export	package	while
extends	private	swith
false	protected	